A Discipline of Multiprogramming:
A Programming Theory for Distributed Applications

Jayadev Misra

To My Parents

Sashibhusan and Shanty

Preface

For every complex problem, there is a solution that is simple, neat and wrong, so noted H. L. Mencken.[1] It is with great trepidation, therefore, that I propose a simple, and mostly neat, solution for designing distributed applications. The solution consists of an object-oriented programming model that imposes a strict discipline on the form of the constituent objects and interactions among them. Additionally, concerns of concurrency are eliminated from the model and introduced only during implementation. The programmers do not have to deal with concurrency explicitly, and that, I believe, is essential for effective concurrent programming.

There are several ways to gain productivity in programming, especially, multiprogramming. First, avoid programming; instead reuse components from a program library whenever possible. Second, employ automatic tools —program analyzers, static type-checkers, performance profilers— that enhance productivity. Third, adopt programming methodologies —structured programming, code walk-throughs, unit testing, rapid prototyping, etc.— that have been known to reduce design errors. The theories and design principles that contribute to productivity in specification, abstraction and modularization are developed in this book. It has been the guiding principle of my career in education and research to ensure that well-trained professionals are considerably more effective than inspired amateurs. The goal of this book is to widen the gap.

---

[1] Prejudices, second series (1920).

A programming model is introduced in chapters 2 and 3. Chapter 2 describes action systems, in which a set of actions is executed fairly in arbitrary order. These systems are relatively easy to analyze, partly because executions of different actions are never concurrent. Chapter 3 enhances the model in which a program consists of components, each an action system, and the components communicate through procedure calls. Each component resembles an object in object-oriented programming. Chapter 4 contains a number of examples that illustrate various aspects of the model and demonstrates that a few simple constructs are adequate for programming a variety of concurrent applications. One notable class of such applications is transaction processing [78]. Chapters 5 through 9 describe a logic for the analysis of programs and their compositions. The logic is quite effective, so that it is often cheaper to prove than to argue! That is, a well designed proof is no longer than a convincing informal argument. A notion of *compatibility* among pairs of actions is defined in chapter 10. A theorem proved in chapter 10 shows (roughly) that concurrent executions of compatible actions are equivalent to some serial executions of these actions; this theorem permits implementations in which compatible actions are executed concurrently, whereas the programming model simplifies the design task by restricting the program executions to the sequential ones. Chapter 11 shows an implementation scheme on a platform of message-communicating processors. Chapter 12 describes a logic for the object model of chapter 3.

Most of the chapters have a small amount of theory and a large number of examples. The examples help illustrate the application of the theory; additionally, the reader may appreciate the succinctness of the solutions (which have worked as reality checks in my research). The examples are mostly the standard ones from the literature: communications over bounded and unbounded channels, maintaining a database, implementing a caching strategy, mutual exclusions and synchronizations, resource allocation, etc. These are now standard design patterns in software construction. Many of them are so standard —mutual exclusion, for instance— that they have been integrated into operating systems and even into hardware. The reason for choosing these well known examples is to avoid the cost —pages of explanation, patience of the reader— of introducing new examples. The choice of examples makes it possible to compare solutions in this book with other solutions in the literature. However, the more important reason for choosing these examples is that they embody the kinds of problems that programmers typically encounter in practice.

The most important omission in this book is empirical support for some of the claims I like to have made on ease of programming and efficiency of program execution. Though my students and I have applied our model to a variety of examples, we cannot honestly assert (though we believe) that telephony software, for instance, will be simplified significantly by using our methods. Such a claim can be validated by a team of programmers working over an extended period, a facility we do not have. We have implemented

two versions of the programming system (using C^{++} and Java as the base languages), but we do not have an extensive set of performance data to evaluate the efficiency of the implementations.

The material in this book draws upon a large body of research in object-oriented programming, database transaction processing, communicating process theory, and action systems. I do not expect the reader to be familiar with all, or any, of these areas. In fact, very little background is assumed of the reader. Familiarity with basic concepts in concurrent programming, which are covered in a typical undergraduate-level course in operating systems, will help, though all concepts and examples are explained to the extent required to follow the arguments in the book. All of the formal material used in this book is developed here; little background beyond basic predicate calculus (which is summarized in the Appendix) is required to understand and appreciate the power of formal manipulations.

This book can be used in a graduate-level research course in computer science. Additionally, I hope researchers and computing practitioners will find the ideas worthy of serious consideration.

Acknowledgments

I owe a huge debt to my students, former students and associates: Will Adams, Flemming Andersen, Al Carruth, Ernie Cohen, David Goldschlag, Rajeev Joshi, Markus Kaltenbach, Edgar Knapp, Jacob Kornerup, Josyula R. Rao, Steve Roberts, Ambuj Singh, and Mark Staskauskas. Will Adams and Rajeev Joshi have contributed directly to this book through their Ph.D. works. Works of Ernie Cohen and Josyula R. Rao were instrumental in the development of the programming model. I am indebted to Edsger W. Dijkstra for being a colleague, friend, critic, and mentor, excelling in each of these roles. My interactions with C.A.R. Hoare have contributed to the refinement of the model; the idea of pre-procedure is inspired by a similar construct in CSP [91]. I am grateful to Lorenzo Alvisi, Don Batory, Robert Blumofe, Gruia-Catalin Roman, Edsger W. Dijkstra, Rutger M. Dijkstra, C.A.R. Hoare, Rajeev Joshi, Edgar Knapp, Rustan Leino, Pete Manolios, Peter McCann, Doug McIlroy, Greg Nelson, Vijaya Ramachandran, Harrick Vin, the PSP Research Group in Austin, the Eindhoven Tuesday Afternoon Club, especially Wim Feijen, and the Distributed Systems Reading Group at the Technische Universität München, especially Ingolf Krüger, for reading and commenting on portions of this manuscript. Ingolf Krüger developed the first C^{++}-based implementation; a Java-based language was designed by Joshi and Adams and implemented by Raymond Tse under the guidance of Lorenzo Alvisi and Calvin Lin. David Gries has been a colleague and friend for a very long time; I greatly appreciate his expenditure of time on this book.

The work reported here borrows heavily from UNITY [32], a programming model that K. Mani Chandy and I had developed earlier. I have always been inspired by my interactions with Chandy.

I appreciate that my family —Mamata, Amitav and Anuj— has endured the preparation of this book.

I am grateful for financial support over the years from an endowment funded by the O'Donnell Foundation of Dallas and an excellent research environment at the University of Texas at Austin. The National Science Foundation has supported this research through awards CCR–9803842, CCR–9707056, CCR–9504190, and CCR–9111912. Lucent Technologies, Naperville, funded part of the research in connection with applications in telephony; I am grateful to David Weiss for support and to Lalita Jagadeesan for technical collaboration.

Austin, Texas J. Misra
January 2001

Contents

1
A Discipline of Multiprogramming

1.1 Wide-Area Computing

The main software challenge in developing application programs during the 1960s and the 1970s was that the programs had to operate within limited resources, i.e., slow processors, small memories, and limited disk capacities. Application programming became far more widespread during the 1980s because of the falling prices of hardware (which meant that more processing power and storage were available for the same cost) and a better understanding of the application programming process. However, most applications still ran on mainframes or over a cluster of machines in a local-area network; truly distributed applications that ran over wide-area networks were few because of the latency and bandwidth limitations of long-haul communication. The 1990s saw great strides in broad-band communication, and the World Wide Web provides a giant repository of information. This combination promises development of a new generation of distributed applications, ranging from mundane office tasks —e.g., planning a meeting by reading the calendars of the participants— to real-time distributed control and coordination of hundreds of machines —e.g., as would be required in a recovery effort from an earthquake.[1]

The obvious problems in applications design that are related to the characteristics of wide-area communication are security and fault-tolerance.

[1] I am indebted to my colleague Harrick Vin for this example and extensive discussions on related topics.

These issues were present even when most computing was done on a single processor, but they have been magnified because messages can be intercepted more easily over a wide-area network, and it is more likely that some node will fail in a 1,000-node network. We contend that growth in applications programming is hindered *only slightly* by these technical problems; the crucial barrier is that *distributed application design is an extremely difficult task because it embodies many of the complexities associated with concurrent programming.*

The distributed applications we envisage have the structure that they collect data from a number of sources, compute for a while, and then distribute the results to certain destinations. This simple paradigm hides a multitude of issues. When should an application start executing— when invoked by a human, by another application, periodically, say, at midnight, or triggered by an event, say, upon detection of the failure of a communication link? How does an application ensure that the data it accesses during a computation is not altered by another concurrently executing application? How do communicating parties agree on the structure of the data being communicated? How are conflicts in a concurrent computation arbitrated? In short, the basic issues of concurrent computing, such as exclusive access to resources, deadlock, and starvation, and maintaining consistent copies of data, have to be revisited in the wide-area context.

One set of issues arises from the current structure of the World Wide Web. The Web sites are designed today under the assumption that their users are humans, not machines. Therefore, the sites are suitable for navigation by humans, and the browsers make it pleasant —by permitting clicks on hyper-links, for instance— for humans to visit related sites from a given site. The emphasis on human interaction has made it difficult, unfortunately, for machines to extract data from one or more sites, compute, and distribute the results to a number of users. For instance, given a database of news sites, it is not easy to "display all stories about cyclones published in the last 3 days". Given that professors in a department produce a grade sheet for each course they teach, it is currently a major effort to collate this information and produce the grade sheets for all students. Nor is it easy to arrange a meeting of professors all of whose calendars are available online.

Proposal for a programming model

There seems to be an obvious methodology for designing distributed applications: represent each device (computer, robot, a site in the World Wide Web) by an object and have the objects communicate by messages or by calling each others' methods. This representation maps conveniently to the underlying hardware, and it induces a natural partition on the problem that is amenable to stepwise refinement. We start with this model as the basis, and simplify and enhance it so that it is possible to address the concurrent programming issues.

The current view of wide-area programming typically requires a human being to invoke a method, and the method provides its results in a form suitable for human consumption. A program that runs each week to plan a meeting of a set of professors —by scanning their calendars, reserving a room for that time, and notifying the affected parties— is quite cumbersome to design today (see the example in section 1.2). Such programs that run autonomously based on certain conditions —once a week, whenever a grade is posted for a student, or when the stock market crashes— are called *actions* in this book. The coding of methods and actions are essentially identical, and we treat them similarly in the programming model.

We espouse a more elaborate view of methods (and actions) that is appropriate for wide-area computing. It may not always be possible for a method to be executed because the state of the object may not permit it. Such is the case for a *P*-method on a semaphore [58] when the semaphore value is zero, or a monitor [90] procedure that is called to remove an item from a buffer when the buffer is empty. The traditional approach then is to queue the caller, accept calls on other methods that may change the object state, and complete a queued call only when the object state permits it. Therefore, it is possible for a caller to be queued indefinitely.

We adopt a different approach: a call should be *accepted* by a method only if its completion is guaranteed, and *rejected* otherwise; a rejected caller may attempt its call in the future. Callers are not queued, and each caller is guaranteed a response from the called procedure in finite time.

The programming model proposed in this book and the associated theory have been christened *Seuss*. The major goal of Seuss is to simplify multiprogramming[2]. To this end, we separate the concern of concurrent implementation from the core program design problem. A program execution is understood as a single thread of control —sequential executions of actions that are chosen according to some scheduling policy— yet program implementation permits concurrent executions of multiple threads (i.e., actions). As a consequence, it is possible to reason about the properties of a program from its single execution thread, whereas an implementation may exploit the inherent concurrency for efficient execution. A central theorem establishes that multiple execution threads implement single execution threads; i.e., for any concurrent execution of actions there exists an equivalent serial execution of those actions.

The programming model is minimal; all well-known constructs of concurrent programming —process, message communication, synchronization, rendezvous, waiting, sharing, and mutual exclusion— are absent. However, the built-in primitives are powerful enough to encode all known communication and synchronization protocols succinctly. The fundamental concepts

[2]We use the terms "multiprogramming" and "concurrent programming" synonymously.

in the model are objects and procedures; a procedure is a method or an action. No specific communication or synchronization mechanism, except procedure call, is built in.

Seuss proposes a complete disentanglement of the sequential and concurrent aspects of programming. We expect large sections of concurrent programs to be designed, understood, and reasoned about as sequential programs. *A concurrent program merely orchestrates executions of its constituent sequential programs, by specifying the conditions under which each sequential program is to be executed.*

1.2 An Example: Planning a Meeting

To illustrate the intricacies of concurrent programming and motivate discussion of the programming model, we consider a small though realistic example.

1.2.1 Problem description

Professors in a university have to plan meetings from time to time. Each meeting involves a nonempty set P of professors; the meeting has to be held in one of a specified set R of rooms. A meeting can be held at time t provided that *all* members of P can meet at t and *some* room in R is free at t. Henceforth, time is a natural number and each meeting lasts one unit of time. The calendar of professor p can be retrieved by calling procedure $p.next$ with a time value as argument: $p.next(t)$ is the earliest time at or after t when p can meet. Similarly, for room r, $r.next(t)$ is the earliest time at or after t when r is free. Thus, $p.next(t) = t$ denotes that p can meet at t, and there is a similar interpretation of $r.next(t) = t$.

Our goal is to write a procedure *plan* that returns the earliest meeting time within an interval $[L, U)$, where the interval includes L and excludes U, given P and R as arguments; if no such meeting time exists, that fact is reported. Once a suitable meeting time and the associated room are determined, the calendars of the affected professors and the room are changed to reflect that they are busy at that time. To this end, each professor or room x has a procedure $x.reserve$; calling $x.reserve(t)$ reserves x for a meeting at t.

A simpler version of this problem appears in [32, section 1.4] and is also treated in sections 5.5.2 and 6.5.2 of this book. In these versions, room allocation is not a constraint. In the current version, professors impose a universal constraint —*all* professors in P have to meet at the scheduled time— and the rooms impose an existential constraint —*some* room in R should be free then.

1.2.2 Program development

Assume that rooms are represented by integers so that they can be numerically compared. Also, for any professor or room x, $x.next$ is ascending and monotonic; i.e., for all times s and t,

$$t \leq x.next(t), \text{ and}$$
$$s \leq t \;\Rightarrow\; x.next(s) \leq x.next(t)\ .$$

See section 5.5.2 for a discussion of these requirements.

Notation The notations for arithmetic and boolean expressions used in this example are explained in appendix A.2.1. In this section

$\langle \forall x :\ x \in P :\ t = x.next(t) \rangle$
  means that all professors in P can meet at t,
$\langle \exists y :\ y \in R :\ t = y.next(t) \rangle$
  means that some room in R is free at t,
$\langle max\ p :\ p \in P :\ p.next(t) \rangle$
  is the maximum over all p in P of $p.next(t)$, and
$\langle min\ y :\ y \in R \wedge t = y.next(t) :\ y \rangle$
  is the smallest (numbered) room in R that is free at t.
  The value of the expression is ∞ if no room is free at t. □

Define time t to be a *common meeting time* (abbreviated to *com*) if all professors in P can meet and some room in R is free at t. That is,

$$com(t) \;\equiv$$
$$\langle \forall x :\ x \in P :\ t = x.next(t) \rangle \;\wedge\; \langle \exists y :\ y \in R :\ t = y.next(t) \rangle.$$

Note that,

$$\langle \forall x :\ x \in P :\ t = x.next(t) \rangle \;\equiv\; (t = \langle max\ p :\ p \in P :\ p.next(t) \rangle).$$

Similarly,

$$\langle \exists y :\ y \in R :\ t = y.next(t) \rangle \equiv\; (t = \langle min\ r :\ r \in R :\ r.next(t) \rangle).$$

Therefore,

$$com(t) \;\equiv$$
$$t = \langle max\ p :\ p \in P :\ p.next(t) \rangle \;\wedge\; t = \langle min\ r :\ r \in R :\ r.next(t) \rangle.$$

In the following procedure, variable t is repeatedly assigned values of the expressions in the two given conjuncts of $com(t)$ (in a specific order, though any order would do) until $com(t)$ holds or t falls outside the interval $[L, U)$. If there is a common meeting time in $[L, U)$, then t is set to the earliest such time and r to a room in R that is free at t. If there is no such time in $[L, U)$, t is set to a value above the interval, i.e., $t \geq U$; the value of r is then irrelevant. We assert without proof that $L \leq t$ is an invariant of the main loop in procedure *plan* given next.

---

procedure $plan(P, R, L, U, t, r)$

$t := L$;
while $\neg com(t) \ \wedge \ t < U$ **do**
$t := \langle max\ p :\ p \in P :\ p.next(t)\rangle$;
$t := \langle min\ r :\ r \in R :\ r.next(t)\rangle$
enddo ;
$\{(L \leq t) \ \wedge \ (com(t) \ \vee \ t \geq U)\}$

if $t < U$ **then** $\{com(t) \ \wedge \ L \leq t < U\}$

{reserve the professors in P at t}
for $p \in P$ **do** $p.reserve(t)$ **endfor** ;

{find a room in R and reserve it at t}
$r := \langle min\ y :\ y \in R \ \wedge \ t = y.next(t) :\ y\rangle$;
$r.reserve(t)$

endif
end $\{plan\}$

1.2.3 Correctness and performance of plan

There are two ways to look at the correctness question: (1) *plan* is correct if none of the calendars (of the professors or the rooms) is changed during its execution by another program, and (2) *plan* is correct even when the calendars are changed during its execution. The first proposition, sequential correctness, is considerably easier to establish. For the current discussion, sequential correctness is not the central issue. There are well-known methods to establish such results; we refer the reader to sections 5.5.2 and 6.5.2 of this book for a thorough treatment of a variation of this problem. [3]

The second problem listed, correctness under concurrent execution, is very hard. Procedure *plan* may not work correctly if the calendar for some member of P or R is changed during its execution. In particular, concurrent executions of two instances of *plan* may reserve a room (or a professor) for two meetings simultaneously.

The problem is eliminated if each instance of *plan* gains exclusive access to the shared data, by explicitly locking the calendars of the members of P

[3] Correctness arguments can be based on the following facts: (1) any common meeting time in the interval $[L, U)$ is at least t (therefore, if *plan* returns such a time, t is the earliest common meeting time), and (2) if $\neg com(t) \wedge t < U$ holds, t will be increased eventually (therefore, either a common meeting time will be found or $t \geq U$ will hold).

and R before it commences execution. A more sophisticated strategy is to employ two-phase locking [20, 67]: all locks are acquired before any unlocking. Additionally, if the locks are acquired in a specific order, deadlock can be avoided. The programmer can introduce explicit locks into the code, or a compiler can insert them. Another possible protocol is as follows: each professor or room tentatively commits to a time whenever *next* is invoked and a commitment becomes permanent when *reserve* is invoked.

To execute several instances of *plan* concurrently, we can also exploit some of the properties of the program. For instance, if two instances of *plan* have disjoint sets of professors and disjoint sets of rooms or disjoint intervals $[L, U)$, their executions are non-interfering, and they can be executed concurrently. A more sophisticated scheme is to run exactly one iteration of the loop in each invocation of *plan*; if the iteration finds a common meeting time, then the rest of the procedure is executed to reserve the room and the professors and inform the caller; if no such time is found and $t < U$ after an iteration, then the call is rejected, i.e., the caller is asked to retry the call in the future. Thus, each call of *plan* locks the required data for only one iteration. Successive calls to *plan* may start with different calendars, and the requirement of the *earliest* meeting time may have to be replaced with *any* meeting time. However, such strategies are problem dependent; we cannot expect a program analyzer to deduce program properties and implement such strategies automatically.

1.3 Issues in Multiprogram Design

1.3.1 Concurrency is not a primary issue in design

We espouse the thesis that programmers should be concerned primarily with the problems they are solving and only secondarily with the implementation issues, such as concurrency. We have advocated this thesis for a number of years and demonstrated it in a number of examples in [32]. We continue to advocate that explicit concurrency considerations do not belong in program design, at least not in the early stages. A concurrent program should be designed as if each component in it will be executed in isolation; all other programs in the universe are suspended in favor of the executing component, and all state changes are attributable to this component alone.

The immediate consequence of this suggestion is that concurrent programming is now a vastly simpler task. Unfortunately, it is also a vastly impractical task because of severe degradation in performance. We examine these two issues next —correctness in this section and performance in section 1.3.3.

As we argued in section 1.2.3, correctness is much easier to establish if each component of a program is executed in isolation. In this book, the

unit of uninterrupted execution, called an *action*, is a sequential program.[4] If several actions have to be executed, they are executed in arbitrary, but serial, order; i.e., their internal steps are never interleaved. Thus, execution of an action completes before another is started.

Correctness of an individual action is established using traditional theories. An action is specified by a pair of predicates, its pre-condition and post-condition, and its correctness criterion is as follows: starting in a state where the pre-condition holds, execution of the action terminates in a state where the post-condition holds. (Termination is discussed later.) This aspect of programming and proof theory is in the domain of sequential programming, and we have little to say about it in this book. We are concerned largely with how to compose programs from objects and objects from procedures. We develop notations, methodology, and logic for designs of such programs. Correctness of a program can be deduced from the specifications of its constituent actions using some flavor of temporal logic [32, 118, 127, 128, 138, 139]; we develop an enhanced version of UNITY logic [32] in this book.

The constraint on executions of actions —execution of an action completes before another is started— has the consequence that "waiting" is now a meaningless concept. Since an action is executed alone, it cannot wait for another action to establish a condition for continuation of its execution. A process may wait neither to receive data along its input channel nor for a resource that it has requested to be granted; queuing up for a semaphore is a fruitless activity. Rendezvous-based communication that requires simultaneous participations of a sender and a receiver is outside our programming model. Our actions are all wait-free. Further, if an action is executed forever, it prevents execution of every other action. Therefore, execution of each action must be guaranteed to terminate (when started in an appropriate state). Termination guarantee is part of sequential correctness and is an obligation on the programmer. Our concern, therefore, is to develop a theory of programs consisting of wait-free, terminating actions.

1.3.2 Structuring through objects, not processes

The unit of abstraction in a typical concurrent program is a *process*. Processes are executed autonomously and concurrently, and they communicate with each other either through global shared variables or messages. Our model —a program is a set of wait-free, terminating actions— admits a different style of structuring, consisting of objects, and process communication is replaced by method call.

---

[4] An action can be a parallel program as long as its semantics can be specified by a pre-condition and a post-condition.

A program consists of a set of objects. Each object includes a set of *procedures*, where a procedure is either a method or an action. Actions and methods are similar; the only difference is that an action is executed autonomously, while a method is executed when it is called, as *p.next* and *r.next* are executed by being called from *plan*. The rule for action execution obeys a weak fairness condition: each action is executed infinitely often. (Therefore, a program execution is nonterminating, though each component action execution terminates.) Execution of a procedure —action or method— is strictly sequential: if a procedure calls a method of another object, the caller is suspended and resumes only on completion of the called method. Recall that completion of each method is guaranteed.

For the meeting planning problem, imagine that each *committee* of professors is represented by an object; this object may include an action that is executed periodically, say, at the start of each workweek to plan a meeting for that week. Procedure *plan* is a method that belongs to another object. Also, each professor and room is a separate object that includes the methods *next* and *reserve*. Execution of the action in *committee* initiates a call to *plan*, with professors in that committee and a set of appropriate rooms as arguments. Execution of *plan* calls on methods *next* and *reserve* of professor and room objects, as shown earlier. On completion of its execution, *plan* returns control to the calling action in *committee*. That action may then inform the members of the meeting time and the room (or that no meeting can be planned for that week).

Observe that there is no need to explicitly lock or unlock the calendars of the professors and rooms, because at most one instance of *plan* is executing at any moment. The program can be studied entirely as a sequential program, because concurrency aspects have been excluded during program design.

1.3.3 Implementation for efficient execution

The suggested execution strategy of one action execution at a time is only an illusion. The strategy makes it easier to design and understand programs, but it is totally impractical since it does not permit any concurrent execution; no two sites in the universe can have programs executing simultaneously.[5] What we want, ideally, is for the actions of a program to be executed concurrently for performance reasons, yet for humans to understand the program as if the actions are executed sequentially.

Two actions that are completely independent —i.e., no object is accessed or modified by both— can be executed simultaneously without causing interference. The notion of independence can be refined to allow concurrent

[5] Purists may argue that simultaneity is a meaningless concept in an Einsteinian universe.

executions of actions if their executions have the same effect as their serial executions in some order. In chapter 10, we define a binary relation, called *compatibility*, over the procedures and show that concurrent executions of compatible actions are equivalent to some serial executions of these actions.

Operations P and V on general semaphores are compatible and so are *put* and *get* over unbounded first-in–first-out channels. That is, whenever a call on *get* can be accepted, an execution of *put* before or after *get* has the same effect on the program state. However, operations *read* and *write* on a shared file are not compatible, as would be expected; the outcome of a *read* may depend on whether a *write* precedes or follows it. For the planning problem, $p.next$ and $q.next$ are compatible for all professors and rooms p and q (including $p = q$). However, $p.next$ and $p.reserve$ are not compatible because executing them in different order may yield different outcomes. Therefore, two invocations of *plan* cannot be executed concurrently if one may possibly call $p.next$ and the other $p.reserve$.

Programmers have been successful in writing concurrent programs because, we believe, most pairs of actions are compatible. A scheduler can be employed to ensure that only compatible actions are executed concurrently; see an implementation in chapter 11. The programmer need only specify the pairs of methods in each object that are compatible; an efficient algorithm determines compatibility for all pairs of procedures given this information. The programmer's specification may be incomplete; if no pairs are specified to be compatible, the program is still executed correctly but with a reduced amount of concurrency. The scheduler in chapter 11 effectively simulates acquisition and release of locks. The scheduler can be distributed. Other implementation schemes, inspired by database commit protocols, can also be developed.

1.3.4 Transformational and reactive procedures

What happens when a procedure calls a method to request a resource and the resource is unavailable, such as attempting to receive a message from a channel that is empty? The called method can return an exception code to denote that it cannot be executed successfully. However, this type of interaction is common enough in concurrent programming that we distinguish between methods that always *accept* calls (execute their codes and, possibly, return some values) and those that may *reject* a call (to denote that the method cannot be executed in the present state). The former are called *total methods* and the latter *partial methods*. This distinction plays a central role in the programming model as well as in the development of the theory of concurrent execution.

A procedure in traditional sequential programming —to sort an array of integers, for instance— is a total method in our model. A procedure such as a P operation on a semaphore or a *get* operation on a channel is a partial method, because P and *get* can cause a caller to wait. Since

our model does not admit waiting, partial methods reject a call whenever completion cannot be guaranteed. In fact, a rejection should happen as early as possible in the execution of an action. In our model, rejection takes place before any change in the caller's state, and rejection itself does not affect the caller's state. Therefore, the caller is oblivious to rejection. In database terminology, rejection is "abort", and abort, in general, requires a rollback of the system to a valid state. However, our model avoids this problem because a call is rejected *before* causing any state change that requires rollback.

A rejection represents a transient condition, whereas acceptance represents a stable condition. In traditional concurrent programming, if a process polls its incoming channel and finds it empty, it cannot assert that it is empty (and, hence, start a computation based on channel emptiness) because the condition may be falsified even before the start of the computation.

A total procedure represents a *transformational* program; a partial procedure, a *reactive* program, in the terminology of Manna and Pnueli [127]. We exploit the distinction between total and partial procedures to get a weaker definition of compatibility (i.e., more pairs of actions are compatible —hence, more pairs can be executed concurrently— than would be possible if all methods were regarded as total). See section 3.4.1 for a longer discussion on partial and total procedures.

1.4 Concluding Remarks

Most process control systems —e.g., telephony, avionics— are conveniently represented using actions. Even an operating system can be structured in this manner. Typical actions in an operating system may be for garbage collection, response to a device failure, and allocation of resources in response to a request. A process control system includes actions that receive and process data from external sources, update internal data structures, and detect dangerous operating conditions. Each of these actions may involve a large amount of computation, but at the level of program design it makes sense to regard each action as a unit and design a larger system based on the units.

Programming of individual actions is a much-studied subject in the arena of sequential programming. This book contributes little to that effort. The emphasis in this book is on the *compositions* of actions and objects. Composition is fundamental for designs of complex software systems. Our work addresses some of the issues in program composition, including specifications of interfaces, predictions of system properties from the component properties, and design principles for "safe" compositions of subsystems.

A programming model is incomplete without an appropriate theory to aid its user in the analysis of programs. This is particularly true for concurrent programs because they tend to be harder. An action is often designed by assuming that the starting state, i.e., its pre-condition, satisfies some invariant. The obligation of the action is to reestablish the invariant as a post-condition. Additionally, establishment of progress properties, such as that execution of each action achieves a certain goal —planning a meeting, for instance— requires a theory that is more general than the study of invariants. We propose such a theory in this book.

1.5 Bibliographic Notes

The programming model that most closely resembles the approach presented here is transaction processing. There is a vast amount of literature on that subject; we refer the reader to Gray and Reuter [78] for a comprehensive survey. Bernstein and Lewis [19] contains a thorough treatment of concurrency issues in database systems. See Broy [26] for another approach to designs of distributed applications. Feijen and van Gasteren [69] have developed a beautiful approach, based on the classic work of Owicki and Gries [145], for designs of multiprograms, and they illustrate the approach convincingly on a large number of examples. It is yet to be seen if their work will scale up for larger problems. Jackson [95] discusses a number of thought-provoking issues in specification and programming methodology.

2
Action Systems

In chapter 1 we suggested that a program be structured as a set of objects. Each object consists of actions and/or methods, where the actions are executed autonomously (following a specific execution rule) and the methods are executed when they are called. In this chapter, we consider a simpler version of this model; we eliminate the methods altogether, retaining only actions. The immediate consequence of this decision is that the objects can no longer communicate through procedure calls; we require the objects to communicate via shared variables. Actions from different objects can read/write into these variables. However, at most one action is executed at any time, so there is no possibility of concurrent write into a variable.

This is an appropriate model for programs where communications among components play a minor role; computations of a single component are of the primary interest. We have chosen to study this simpler model —called *action systems*— because many of the basic concepts of the general model can be explained within it. The simpler model suffices for many problems; we can express the solution to a problem as an action system and study its properties employing a simple logic, which we develop in chapters 5 to 9. We describe the general programming model in chapter 3 and a logic for it in chapter 12.

2.1 An Informal View of Action Systems

An *action system*, often called a *state transition system*, is a program that consists of a set of objects and shared (global) variables. Each object may have local variables. The program state is given by the values of its variables, local and global. Each object has one or more actions that may change the program state. The objects interact by reading and writing the shared variables. Interaction is not the primary subject of this chapter; therefore, in most cases we deal with a single object and study how its actions change the values of the variables.

We employ a neutral term, *box*, for an object. A program has a set of boxes. When the program consists of a single box, we refer to the box and program synonymously. Also, we use the terms "action system" and "program" synonymously in all cases.

The variable values at any point during a computation define the current state, and all possible combinations of variable values define the state space. An odometer in a car, for instance, may be regarded as a system with six variables, one for each position; each of these variables may, independently, assume a value between 0 and 9. Thus, the state space may be represented by six-digit numbers. For the six-digit odometer, there is usually a single action whose effect is to add 1 (modulo 10^6) to the current state (a mechanic may have access to another method that resets the odometer to 0).

Mathematically, an action is a binary relation over the state space. For any state, an action describes a set of successor states. If the successor set of state s has exactly one state, the effect of the action is to transform s to its unique successor; such is the case for the odometer, described above. If the successor set has more than one state, the current state is transformed to *any* of its successors; such an action is called *nondeterministic*. If the successor set is empty, the action is not *enabled* in the given state. If an action is executed in a state in which it is not enabled, its execution has no effect; i.e., the state does not change. (This rule can be modeled by including the pair (s, s) in the relation for an action that is not enabled at s.)

What constitutes an action is a methodological issue. In designing a sorting routine, for instance, we may make use of an action that exchanges a pair of data items, whereas in a spreadsheet program we may assume that sorting is a built-in primitive. A programming problem often specifies the set of available actions. In concurrent programming, a sequence of steps that may be executed without interruption is typically regarded as an action.

We focus attention on discrete action systems: "discrete" means that there are no continuously changing variables, such as flow in a pipe or voltage in an oscillating analog circuit. We do allow finite as well as infinite numbers of states. We assume that there is a finite number of actions, though our theory is largely applicable to infinite action systems, as well.

2.2 Syntax and Semantics of Action Systems

The notation used to describe an action system is mostly irrelevant for our purpose. Any reasonable notation may be used; we choose one to illustrate the concepts and show some examples.

An action system is given by the (1) specification of the state space and (2) specifications of the actions. We assume that a finite number of variables can define the state space. As an example,

boolean b;
integer x, y

defines a state space where each state is a triple that describes the values of b, x, and y.

The initial states are specified by declaring the initial values of some (or all) of the variables; e.g.,

boolean $b = \mathit{false}$;
integer $x, y = 0, 0$

The *initial condition* of a program is a predicate that holds at exactly the initial states. For the example, the initial condition is

$$\neg b \ \wedge \ x = 0 \ \wedge \ y = 0 \ .$$

An action is a guarded command [55]; it consists of a *guard* —a predicate over the state space— and a *command* —a prescription for the state change. We employ traditional notations for guards and commands, with $\rightarrow$ to separate the guard and command parts. Thus,

$$x < y \ \rightarrow \ x := x + 1; \ y := y - 1$$

is an action whose guard is $x < y$ and command is $x := x + 1; \ y := y - 1$. We use the symbol $[\!]$ to separate the codes of different actions. A missing guard should be taken to mean the guard *true*.

Requirement on actions Execution of an action, when started in a state where its guard holds, terminates. This has to be ensured by the programmer of the action. □

Execution rule

An execution of an action system starts in an initial state and consists of an infinite number of steps. In each step an arbitrary action is executed. Execution of an action is *ineffective* if the guard of the action does not hold when it is executed; otherwise, it is *effective*. Ineffective execution of an action is a *skip*; it does not change the state. Effective execution of an action consists of executing its body, which may change the state (it is guaranteed to terminate; see the requirement on actions). The executions

of a program are restricted by the following fairness condition: *each action is executed infinitely often in each execution.*

It may seem that an infinite execution is meaningless if the computation is guaranteed to terminate. A terminating computation continues to execute its actions, but no action execution has any effect; therefore, the final state repeats forever. The execution rule defines a logical view of the execution; in an implementation, once it is detected that a final state has been reached, the execution may be stopped and the resources released for other tasks. However, the logical view is convenient for developing a uniform treatment of terminating and nonterminating computations.

Variable types The basic types used for variables in this book are **integer**, **boolean**, and **nat** (for natural, i.e., non-negative integers). Enumerated type with values $\{a, b, c, d\}$, for instance, is written as **enum** $\{a, b, c, d\}$. We simply write **type** to denote a polymorphic type, when type information has no relevance to the discussion. The structured types used are **record**, **set**, **bag**, **array**, and **seq** (for sequence). For a structured variable the type of its elements is also specified, and for each array its bounds. We write $\langle\ \rangle$ for an empty sequence, and $\emptyset$ for both empty set and empty bag. □

2.3 Properties of Action Systems

A thorough treatment of program properties is given in chapters 5 and 6. Here, we describe two of the main concepts —*invariant* and *fixed point*— that are necessary for understanding the examples in this chapter. Progress properties —that a program eventually reaches a desired state— are described in detail in chapter 6; for the moment, we rely on the reader's intuition to establish progress properties.

2.3.1 Invariant

An invariant is a predicate that is initially *true* and is preserved by execution of each action. Therefore, an invariant is always *true* during an execution. (The states reached *during* an execution are the initial state and the state following the execution of each action. The states that are reached *during* execution of an action are invisible; we can observe the states only before and on completion of each action execution.) Formally, predicate p is an invariant if both of the following conditions hold.

initial condition $\Rightarrow\ p$
for each action of the form $g\ \rightarrow\ s,\ \{p \wedge g\}\ s\ \{p\}$

Here, $\{p \wedge g\}\ s\ \{p\}$ denotes that any execution of s started in a state that satisfies $p \wedge g$ terminates in a state that satisfies p; see appendix A.4.1 for details about this notation.

As an example, consider a program that consists of the following box only.

---

box *small*
  integer $x, y = 0, 0$;

    $x < y \ \rightarrow \ x := x + 1$
  $[\!]$ $y := max(x, y) + 1$
end $\{small\}$

We claim that $x \leq y$ is an invariant for this program. We have

initially $x = 0 \ \wedge \ y = 0$

which implies $x \leq y$. We can show that

$$\begin{array}{lll} \{x \leq y \ \wedge \ x < y\} & x := x + 1 & \{x \leq y\} \\ \{x \leq y\} & y := max(x, y) + 1 & \{x \leq y\} \end{array}$$

The notion of invariant is perhaps the most important foundational concept in this book. It is essential for writing specifications and designing programs.

2.3.2 fixed point

A fixed point of a program is a state that remains unchanged by execution of any action. Therefore, once a fixed point is reached, further execution of the program has no effect. The set of all fixed points is described by a predicate called *FP*.

It is possible to compute *FP* from the code of a program provided that we know the states left unchanged by each action. Consider a program whose action i is of the form $g_i \ \rightarrow \ s_i$. Let predicate b_i hold in exactly those states where the execution of s_i has no effect. Then

$$FP \ \equiv \ \langle \forall i :: \ g_i \Rightarrow b_i \rangle$$

Observe that *FP* holds in any state where all g_is are *false*.

It is easy to compute b_i if s_i is an assignment statement. For the assignment statement $x := e$ the corresponding predicate is $x = e$. That is, execution of $x := e$ has no effect exactly when $x = e$ holds prior to the execution. This observation may easily be extended to sequences of assignments and conditional statements; see section 5.3.2 for details. For program *small* of section 2.3.1, we compute[1]

[1] See appendix A.2.1 for an explanation of the proof format used here.

$$
\begin{array}{ll}
 & FP \\
\equiv & \{\text{from the definition of } FP\} \\
 & (x < y \Rightarrow x = x + 1) \ \wedge \ (y = max(x, y) + 1) \\
\equiv & \{\text{arithmetic and predicate calculus}\} \\
 & (x \geq y) \wedge (\mathit{false}) \\
\equiv & \{\text{predicate calculus}\} \\
 & \mathit{false}
\end{array}
$$

That is, each state of *small* can potentially be changed.

There is no direct method for computing the *FP* if the command portion of an action contains loops.

Most of the systems we consider in this book are never expected to reach a fixed point; they should run forever, so their *FP* should be *false*. In many cases, though, a box may reach a fixed point, but then a change in a shared variable by some other box may cause its *FP* to become *false*, and enable some of its actions to be executed effectively.

2.4 Examples

2.4.1 Finite state machine

Finite state machines are conveniently represented by action systems: the machine state can be encoded in a variable, and each state transition is an action. Alternatively, it may be possible to define a set of variables where the variable values encode the states and each transition affects only a small number of variables.

We show two different representations of a finite state machine that accepts binary strings that have an even number of zeroes and an odd number of ones. A pictorial representation of the machine is given in Fig. 2.1. In this figure, the initial state is a and state c is the only accepting state.

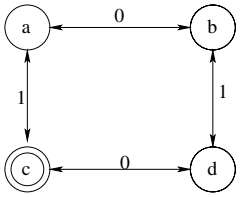


Figure 2.1: Finite state machine accepting even number of 0's and odd 1's

A box *FSM1* that represents this finite state machine follows. Variable *state* assumes one of the values a, b, c, and d. Variable x holds the next binary digit to be scanned. Some external box E stores a value into x after

FSM1 has scanned the digit; E is usually called the *environment* of *FSM1*. The following protocol is used by E and *FSM1* to read/write into x. The value of x is ϕ when there is no value to be scanned; in this case box E may store a binary digit in x. Box *FSM1* reads a value from x if $x \neq \phi$ and then it sets x to ϕ.

---

box *FSM1*
  enum $\{a, b, c, d\}$ $state = a$;
  enum $\{0, 1, \phi\}$ x;

  $x = 0 \ \rightarrow$ **if** $state = a$ **then** $state := b$
    elseif $state = b$ **then** $state := a$
    elseif $state = c$ **then** $state := d$
    else $\{state = d\}$ $state := c$
    endif ; $x := \phi$

  $[\!]$ $x = 1 \ \rightarrow$ **if** $state = a$ **then** $state := c$
    elseif $state = b$ **then** $state := d$
    elseif $state = c$ **then** $state := a$
    else $\{state = d\}$ $state := b$
    endif ; $x := \phi$
end $\{$*FSM1*$\}$

Box *FSM1* reaches a fixed point when $x \neq 0 \ \wedge \ x \neq 1$, i.e., $x = \phi$. Then *FSM1* is merely waiting for input from its environment.

In the following box, we encode the state by two boolean variables $p0$ and $p1$, where $p0$ is *true* iff the number of scanned 0's is even; $p1$ is similarly defined. Thus, states a, b, c, d are encoded by the following values of $p0, p1$, respectively: $(true, true), (false, true), (true, false), (false, false)$. Note that the resulting box is considerably simpler because of the choice of variables that represent the states.

---

box *FSM2*
  boolean $p0, p1 = true, true$;
  enum$\{0, 1, \phi\}$ x;

  $x = 0 \ \rightarrow \ p0 := \neg p0;\ x := \phi$
  $[\!]$ $x = 1 \ \rightarrow \ p1 := \neg p1;\ x := \phi$
end $\{$*FSM2*$\}$

Let $n0$ and $n1$ denote, respectively, the number of 0's and 1's scanned. Variables $n0$ and $n1$ are *auxiliary* variables that can be introduced into

FSM2: initially, both of these variables are 0; $n0$ is incremented in the first action and $n1$ in the second. It can be shown that

invariant $p0 \equiv even(n0)$
invariant $p1 \equiv even(n1)$

2.4.2 Odometer

We consider a three-digit odometer whose state is described by the values of the variables $d0$, $d1$, and $d2$ ($d0$ is the least significant and $d2$ the most significant digit). An external process, the environment of the odometer, sets variable $c0$ to *true* to signify that the odometer should be incremented. The odometer is incremented eventually if $c0$ remains *true*, and then $c0$ is set to *false* (to denote that the incrementation has been completed).

In the first design, we have a single action that increments the odometer when $c0$ is found to be *true*.

---

box *Odometer1*
  enum(0..9) $d0, d1, d2 = 0, 0, 0$;
  boolean $c0$;

  $c0 \ \rightarrow\ c0 := false;$
  $d0 := (d0 + 1)\ mod\ 10;$
  if $d0 = 0$ **then** $d1 := (d1 + 1)\ mod\ 10;$
  if $d1 = 0$ **then** $d2 := (d2 + 1)\ mod\ 10$ **endif**
  endif
end {*Odometer1*}

Observe that if $c0$ becomes *true*, from the fairness condition, the odometer will be incremented and $c0$ set to *false*.

There is a deficiency in our modeling of a physical odometer as an action system. We cannot guarantee that the odometer will be incremented within a very short time of $c0$ being set to *true*; the guarantee that the odometer is incremented eventually may have little value in practice if several miles elapse before an incrementation. We discuss this issue in some detail in chapter 6.

A note on the notation We have not distinguished variable $c0$ from variables $d0$, $d1$, and $d2$ syntactically, even though the latter variables are local to the box (i.e., they cannot be changed by an external action) whereas $c0$ can be changed by an external action. We introduce a syntactic distinction in section 8.2.1. □

The next design treats incrementation of each digit as a separate action. Analogous to $c0$, we introduce booleans $c1$ and $c2$ that signify if $d1$ and $d2$, respectively, are to be incremented. These variables, which are merely carries from the previous digits, are local to the *Odometer2* box. Variable $c0$ remains *true* until all the digits have been incremented appropriately; that is, $c0$ is set to *false* only when there is no carry to the next digit or after the incrementation of $d2$. In the following box we label the three actions, and we use these labels in further discussions.

---

box *Odometer2*
  enum(0..9) $d0, d1, d2 = 0, 0, 0$;
  boolean $c0$;
  boolean $c1, c2 = \mathit{false}, \mathit{false}$;

  α:: $c0 \wedge \neg c1 \wedge \neg c2 \ \rightarrow$
      $d0 := (d0 + 1) \ \mathit{mod}\ 10$; $c1 := (d0 = 0)$; $c0 := c1$
  [] β:: $c1 \ \rightarrow$
      $d1 := (d1 + 1) \ \mathit{mod}\ 10$; $c1 := \mathit{false}$; $c2 := (d1 = 0)$; $c0 := c2$
  [] γ:: $c2 \ \rightarrow$
      $d2 := (d2 + 1) \ \mathit{mod}\ 10$; $c2 := \mathit{false}$; $c0 := \mathit{false}$
end {*Odometer2*}

As before, we would like to show that once $c0$ holds, the triple $(d0, d1, d2)$ is incremented eventually. Also, we would like to show that as long as $c0$ remains *false* the state is unchanged. These claims are obviously true for the first box, *Odometer1*; but they are not so obvious for *Odometer2* because, with the decoupling of the actions, it seems plausible that β may be executed before α when $c0$ becomes *true*, for instance.

For *Odometer2*, we claim the following invariant. The proof of this invariant is left to the reader.

invariant $\langle (c1 \vee c2) \Rightarrow c0 \rangle \ \wedge \ \langle \neg(c1 \wedge c2) \rangle$

Given this invariant, we see that if $\neg c0$ holds, then $\neg c1 \wedge \neg c2$ holds as well; hence, no action execution has any effect. (The following computation of *FP* confirms this fact.) Therefore, as long as $c0$ remains *false*, the odometer is unchanged. Whenever $c0$ holds, exactly one action execution has some effect: α if $\neg c1 \wedge \neg c2$ holds, β if $c1 \wedge \neg c2$ holds, and γ if $\neg c1 \wedge c2$ holds; the remaining possibility, $c1 \wedge c2$, is ruled out by the second conjunct in the invariant. For the progress proof, use an operational argument to conclude that once $c0 \wedge \neg c1 \wedge \neg c2$ holds, the odometer is incremented eventually and $\neg c0 \wedge \neg c1 \wedge \neg c2$ is established.

We compute the *FP* for this box as follows. The command portion of α has an assignment $d0 := (d0 + 1) \ \mathit{mod}\ 10$; setting its left and right side equal yields *false* (similarly for β and γ). The computed *FP* is

$$
\begin{array}{ll}
 & (c0 \wedge \neg c1 \wedge \neg c2 \ \Rightarrow \ \mathit{false}) \ \wedge \ (c1 \ \Rightarrow \ \mathit{false}) \ \wedge \ (c2 \ \Rightarrow \ \mathit{false}) \\
\equiv & \{\text{predicate calculus}\} \\
 & \neg(c0 \wedge \neg c1 \wedge \neg c2) \ \wedge \ \neg c1 \ \wedge \ \neg c2 \\
\equiv & \{\text{predicate calculus}\} \\
 & \neg c0 \ \wedge \ \neg c1 \ \wedge \ \neg c2
\end{array}
$$

The first conjunct in the invariant implies that $\neg c0 \ \Rightarrow \ (\neg c1 \wedge \neg c2)$. Therefore, the *FP* is equivalent to $\neg c0$; i.e., once $\neg c0$ holds, the state remains unchanged by the actions of *Odometer2*.

2.4.3 Greatest common divisor

Action systems are typically used to describe never-ending computations, as in an operating system or a telephone switch. To illustrate the generality of action systems, we show a small combinatorial problem, computation of the greatest common divisor (*gcd*) using repeated subtraction. This example also illustrates the use of invariants and fixed points. Another combinatorial example, computations of shortest paths in a graph, is given in section 2.4.6.

Let M and N be two positive integers whose *gcd* is to be computed. The following scheme is well known.

---

box *GCD*
  integer $m, n = M, N;$

$$
\begin{array}{ll}
 & m > n \ \rightarrow \ m, n := m - n, n \\
[\!] & n > m \ \rightarrow \ m, n := m, n - m
\end{array}
$$

end $\{GCD\}$

We show that the following predicate P is an invariant of box *GCD*:

$$P :: \ m > 0 \ \wedge \ n > 0 \ \wedge \ gcd(m, n) = gcd(M, N) \ .$$

Initially, P holds because $m, n = M, N$ and M and N are positive integers. Next, we show

$$
\begin{array}{lll}
\{P \ \wedge \ m > n\} & m, n := m - n, n & \{P\} \\
\{P \ \wedge \ n > m\} & m, n := m, n - m & \{P\}
\end{array}
$$

These assertions can be proved from the following properties of *gcd* (which we do not prove here). For positive integers x and y:

$$
\begin{array}{l}
gcd(x, y) = gcd(y, x) \\
x > y \ \Rightarrow \ gcd(x, y) = gcd(x - y, y)
\end{array}
$$

Next, we show that $m = gcd(M, N)$ holds at any fixed point reached by the box *GCD*. First, compute the *FP* for this box.

$$
\begin{array}{ll}
 & FP \\
\equiv & \{\text{from the definition of } FP\} \\
 & \quad \langle (m > n) \;\Rightarrow\; (m, n = m - n, n) \rangle \;\wedge \\
 & \quad \langle (n > m) \;\Rightarrow\; (m, n = m, n - m) \rangle \\
\equiv & \{\text{Simplify}\} \\
 & \quad \langle (m > n) \Rightarrow (n = 0) \rangle \;\wedge\; \langle (n > m) \Rightarrow (m = 0) \rangle
\end{array}
$$

Any fixed point reached by the box satisfies the invariant and this FP; hence, at a reachable fixed point

$$
\begin{array}{ll}
 & P \;\wedge\; \langle (m > n) \Rightarrow (n = 0) \rangle \;\wedge\; \langle (n > m) \Rightarrow (m = 0) \rangle \\
\Rightarrow & \{P \;\Rightarrow\; n > 0. \text{ And } m > n \;\Rightarrow\; n = 0. \text{ So } m \leq n. \\
 & \quad \text{Similarly, } n \leq m\} \\
 & \quad m \leq n \;\wedge\; n \leq m \;\wedge\; gcd(m, n) = gcd(M, N) \\
\Rightarrow & \{\text{arithmetic}\} \\
 & \quad m = n \;\wedge\; gcd(m, n) = gcd(M, N) \\
\Rightarrow & \{gcd(x, x) = x, \text{ for any positive integer } x\} \\
 & \quad m = gcd(M, N)
\end{array}
$$

The remaining proof obligation is that every execution of GCD eventually reaches a fixed point. This result does not follow from anything we have proved so far: if we replace $m - n$ with $m + n$ and $n - m$ with $n + m$, all the proof steps remain valid, yet the box will never reach a fixed point. Since we have not developed a theory of progress, we provide an operational argument to justify that a fixed point will be reached. Observe that if $m \neq n$, execution of one of the actions changes m or n, thus decreasing $m + n$, whereas the other action has no effect. From the fairness rule that each action is eventually executed, we conclude that $m + n$ will be decreased eventually if $m \neq n$. Since both m and n are always positive (see the invariant), $m + n$ can be decreased a finite number of times only. Hence, within finite time $m = n$, and this implies FP.

2.4.4 Merging sorted sequences

This example demonstrates that message-communicating processes may be represented easily as action systems. We design a box that merges the data received along three input channels. Each channel carries an increasing sequence of positive integers; the output of the box is an increasing sequence that includes all (and only) the received values, and this sequence is sent along an output channel. Since the output sequence is increasing, no value appears more than once in the output channel, even though the same value may appear in different input channels. This box is used as part of a larger example in section 4.5.

The shared variables in this example are channels. A channel is an unbounded sequence; an empty channel is denoted by the empty sequence, $\langle\rangle$. Sending value x along channel c has the same effect as

$$c := c +\!\!+ x$$

where $+\!\!+$ is the concatenation operator. Receiving a value from c into v is effected by

$$c \neq \langle\rangle \;\rightarrow\; v, c := c.head, c.tail$$

In this example, some external box appends values to the input channels, and box *Merge*, shown below, removes values from these channels. Dually, *Merge* appends values to the output channel, and some external box receives those values. The protocol shown here guarantees that the channels are first-in–first-out (fifo).

The algorithm used in *Merge* is as follows. The input channels are called f, g, and h, and the output channel, *out*. Each input channel has an integer variable associated with it —vf, vg, and vh with f, g, and h, respectively— that holds the last value read from the channel that has not yet been output; in case all values read from a channel have been output, the corresponding variable value is 0 (recall that the channels carry only positive integers). A value is read from channel f and stored in vf provided that $vf = 0$ and the channel is nonempty; similarly for the other channels. A value is output only if vf, vg, and vh are all nonzero; in that case, the smallest of these values is output, and vf, vg, vh are appropriately modified.

---

box *Merge*
  seq f, g, h, out;
  integer $vf, vg, vh = 0, 0, 0$;
  integer m;

  $vf = 0 \;\wedge\; f \neq \langle\rangle \;\rightarrow\; vf,\; f \;:=\; f.head,\; f.tail$
  $[\!]\; vg = 0 \;\wedge\; g \neq \langle\rangle \;\rightarrow\; vg,\; g \;:=\; g.head,\; g.tail$
  $[\!]\; vh = 0 \;\wedge\; h \neq \langle\rangle \;\rightarrow\; vh,\; h \;:=\; h.head,\; h.tail$
  $[\!]\; vf \neq 0 \;\wedge\; vg \neq 0 \;\wedge\; vh \neq 0 \;\rightarrow$
    $m := min(vf, vg, vh);\; out := out +\!\!+ m;$
    if $m = vf$ **then** $vf := 0$ **endif** ;
    if $m = vg$ **then** $vg := 0$ **endif** ;
    if $m = vh$ **then** $vh := 0$ **endif**
end {*Merge*}

Box *Merge* expects a never-ending stream of values along each input channel. In case a channel carries a finite number of values, some of the values from the other channels may never be output (for instance, if f carries some values and g and h are permanently empty). In that case, each finite sequence should be terminated by a special end marker, say ∞, and the box should be modified to ignore that channel after receiving the special value.

The properties of *Merge* that are of interest are as follows.

1. Each value in *out* is from f, g, or h.

2. *out* is a strictly increasing sequence.

3. Each value from f, g, and h appears eventually in *out*.

The first two properties can be stated as invariants of *Merge* and the last one is a progress property.

2.4.5 Mutual exclusion

Mutual exclusion is a classic problem in concurrent computing. We treat the problem here not because of its intrinsic difficulty or its central place in concurrent computing but as an illustration of refinement in action systems.

Two or more processes each have a section of code called the *critical section*, and it is required that at most one process execute its critical section at any time. Therefore, if two processes attempt to execute their critical sections simultaneously, then one of them will be forced to wait at least until the other has completed execution of its critical section. Additionally, a reasonable progress requirement is that some process eventually executes its critical section if there are processes waiting to enter their critical sections. A stronger progress requirement is that every waiting process eventually be allowed to enter its critical section.

The *Merge* example of section 2.4.4 is part of a *loosely coupled* system, where the components —boxes that write into the input channels of *Merge* and read from its output channel, and the *Merge* box itself— can be developed and understood without detailed understanding of the other components. These components interact only through the shared channels, and such interactions are easy to understand. The thesis in this book is that all large programs should be loosely coupled. In contrast to *Merge*, a solution to the mutual exclusion problem is usually *tightly coupled*; such a solution is difficult to understand by examining the code of each process in isolation. The shared variables are manipulated in an intricate manner, and it is preferable to study the program, consisting of all its components, in its entirety.

In this section, we develop a mutual exclusion algorithm due to Peterson [151]. We start with a high-level solution that is loosely coupled. Next, we refine this solution, implementing a complex shared data structure using elementary data structures. Ultimately, we represent Peterson's solution as a single action system. To show the power of refinement, we derive a second mutual exclusion algorithm from the same high-level program.

Mutual exclusion using a shared queue

The mutual exclusion problem is easily solved if there is a shared queue that the processes can access in an exclusive manner (it seems paradoxical to solve the mutual exclusion problem using a facility that already implements mutual exclusion in access to a data structure; we eliminate this paradox during refinement). A process that attempts to enter its critical section appends its process-id to the back of the queue; the process at the head of the queue enters its critical section, and upon completion removes its id from the queue. The correctness of this solution is easy to see: the process that is executing its critical section is at the head of the queue; hence, at most one process can execute its critical section at any time. Further, a process attempting to execute its critical section will eventually do so if every process completes its critical section in finite time. This follows from (1) the process at the head of the queue enters its critical section, completes it, and then removes the head item of the queue; (2) hence, every item in the queue eventually becomes the head item, so every queued process eventually enters its critical section; (3) a process attempting to enter its critical section appends its id to the queue and, from (2), enters its critical section eventually.

We write a concurrent program using traditional notation. Here, q is the shared queue. The algorithm is described for two processes u and v; we write their ids as "u" and "v", respectively. Each assignment statement is atomic. A guarded command is executed as follows. A process checks the guard from time to time, and the command is executed (atomically) only if the guard holds. The process waits as long as the guard does not hold.

---

program *MutualExclusion*
  seq $q = \langle\rangle$ {initially q is empty};

process u

loop
  noncritical section;
  $q := q +\!\!+$ "u";
  $q.head =$"u" $\rightarrow$ *skip*;
  critical section;
  $q := q.tail$
end

process v

loop
  noncritical section;
  $q := q +\!\!+$ "v";
  $q.head =$"v" $\rightarrow$ *skip*;
  critical section;
  $q := q.tail$
end

end {*MutualExclusion*}

Implementing the shared queue: Peterson's algorithm

Queue q, defined in the previous box, takes on five possible values when it is shared between two processes: $\langle\rangle$, "u", "v", "$u\ v$", "$v\ u$" (here "$u\ v$" represents the queue that has "u" as the head item followed by "v"). Therefore, we need at least three boolean variables to represent the queue. Let boolean variables u and v be *true* when the corresponding id is in the queue. Then it remains to distinguish between the two queue values, "$u\ v$" and "$v\ u$". We introduce a boolean variable *turn* that is *true* when $q =$ "$v\ u$" and *false* when $q =$ "$u\ v$"; the value of *turn* in other cases is irrelevant. The operations on q can now be written in terms of the operations on u, v, and *turn* as follows.

$q = \langle\rangle$	is	$u, v\ = \mathit{false}, \mathit{false}$
$q := q +\!\!+$ "u"	is	$u, \mathit{turn} := \mathit{true}, \mathit{true}$
$q := q +\!\!+$ "v"	is	$v, \mathit{turn} := \mathit{true}, \mathit{false}$
$q.\mathit{head} =$"u"	is	$\neg v\ \vee\ \neg \mathit{turn}$
$q.\mathit{head} =$"v"	is	$\neg u\ \vee\ \mathit{turn}$
$q := q.\mathit{tail}$	is	$u := \mathit{false}$ {in process u}
$q := q.\mathit{tail}$	is	$v := \mathit{false}$ {in process v}

To see the transformation for $q := q +\!\!+$ "u", note that q becomes "u" or "$v\ u$" as a result of appending "u" to it. In the first case, *turn*'s value is irrelevant and in the second case, *turn* has to be set to *true*; therefore, we set *turn* to *true* in both cases in addition to setting u to *true*. The test $q.\mathit{head} =$"u", given that u is in q, is equivalent to $q =$"u" $\vee\ q =$"$u\ v$", i.e., $\neg v\ \vee\ \neg \mathit{turn}$. Applying the given transformations, we obtain the following program.

---

program *MutualExclusionRefined*
  boolean $u, v = \mathit{false}, \mathit{false}$;

process u

loop
  noncritical section;
  u,*turn*:=*true*, *true*;
  $\neg v\ \vee\ \neg \mathit{turn}\ \rightarrow\ \mathit{skip}$;
  critical section;
  $u := \mathit{false}$
end

process v

loop
  noncritical section;
  v,*turn*:=*true*, *false*;
  $\neg u\ \vee \mathit{turn}\ \rightarrow\ \mathit{skip}$;
  critical section;
  $v := \mathit{false}$
end

end {*MutualExclusionRefined*}

From multiple assignments to single assignments The preceding algorithm is almost identical to Peterson's two-process mutual exclusion algorithm. The remaining step is to decouple the assignments to u and *turn* in process u (and similarly v and *turn* in process v). We can show that (see Misra [134, note 13]) it is safe to replace

u,*turn* := *true*, *true*
by
u := *true*; *turn* := *true* □

Note Switching the order of the two assignments for either process makes the program incorrect. To see this, suppose that processes u and v have the following codes.

process u:: u := *true*; *turn* := *true*
process v:: *turn* := *false*; v := *true*

Consider an execution in which process u sets u to *true*, process v sets *turn* to *false*, and u then sets *turn* to *true*. Now u enters its critical section ($\neg v$ holds); then, process v sets v to *true* and enters its critical section (because *turn* holds), thus violating mutual exclusion. □

Peterson's algorithm as an action system

It is easy to translate the two-process mutual exclusion program into an action system. First, we rewrite the program using two explicit program counters —m for process u and n for process v— that take on integer values between 0 and 3.

---

program *MutualExclusionRefined1*
boolean u, v = *false*, *false*;
integer $m, n = 0, 0$;

process u	process v
loop	**loop**
noncritical section;	noncritical section;
u, m := *true*, 1; *turn*, m := *true*, 2;	v, n := *true*, 1, *turn*, n := *false*, 2;
$\neg v \vee \neg turn \rightarrow skip$;	$\neg u \vee turn \rightarrow skip$;
{enter critical section} $m := 3$;	{enter critical section} $n := 3$;
critical section;	critical section;
u, m := *false*, 0	v, n := *false*, 0
end	**end**

end {*MutualExclusionRefined1*}

We translate this program to the action system shown below. In the translation, we introduce predicates $u.h$ and $v.h$, which are controlled by external boxes. Predicate $u.h$ is set to *true* to denote that process u is waiting to enter its critical section, and it is set to *false* while process u is in its critical section; $v.h$ is manipulated similarly.

The fact that every critical section is eventually completed is simulated by setting m to 0 sometime after it becomes 3 (similarly for n).

---

program *mutex*
  boolean $u, v = \mathit{false}, \mathit{false}$;
  integer $m, n = 0, 0$;

    {process u's box}
  $u.h \ \wedge \ m = 0 \ \rightarrow \ u, m := \mathit{true}, 1$
$[\!]$ $m = 1 \ \rightarrow \ \mathit{turn}, m := \mathit{true}, 2$
$[\!]$ $m = 2 \ \wedge \ (\neg v \ \vee \ \neg \mathit{turn}) \ \rightarrow \ m := 3$
$[\!]$ $m = 3 \ \rightarrow \ u, m := \mathit{false}, 0$

    {process v's box}
$[\!]$ $v.h \ \wedge \ n = 0 \ \rightarrow \ v, n := \mathit{true}, 1$
$[\!]$ $n = 1 \ \rightarrow \ \mathit{turn}, n := \mathit{false}, 2$
$[\!]$ $n = 2 \ \wedge \ (\neg u \ \vee \ \mathit{turn}) \ \rightarrow \ n := 3$
$[\!]$ $n = 3 \ \rightarrow \ v, n := \mathit{false}, 0$
end{*mutex*}

Proof of mutual exclusion

We constructed program *mutex* through a series of transformations starting from the program that used a shared queue. Since *mutex* is a correct refinement of a correct mutual exclusion algorithm it also enforces mutual exclusion. That is, m and n cannot both be 3 simultaneously:

invariant $\neg(m = 3 \ \wedge \ n = 3)$

This fact cannot be proved directly from the program text; we prove invariants (I1) and (I2), given below, from which this fact can be deduced.

invariant $\langle m \neq 0 \ \equiv \ u \rangle \ \wedge \ \langle (m = 3) \ \Rightarrow \ (\neg v \vee \neg \mathit{turn}) \rangle$ (I1)
invariant $\langle n \neq 0 \ \equiv \ v \rangle \ \wedge \ \langle (n = 3) \ \Rightarrow \ (\neg u \vee \mathit{turn}) \rangle$ (I2)

Invariants (I1) and (I2) can be proved by showing that they hold initially and that every action preserves the truth of each of these predicates. The proof is straightforward, and we leave it to the reader. Given (I1) and (I2), we conclude from their conjunction that both processes cannot be in their critical sections simultaneously, as follows.

• Proof of $\neg(m = 3 \ \wedge \ n = 3)$:

$$
\begin{array}{ll}
 & m = 3 \ \wedge \ n = 3 \\
\Rightarrow & \{\text{From (I1), } m = 3 \ \Rightarrow \ u \wedge (\neg v \vee \neg turn), \text{ similarly from (I2)}\} \\
 & u \ \wedge \ (\neg v \vee \neg turn) \ \wedge \ v \ \wedge \ (\neg u \vee turn) \\
\Rightarrow & \{\text{simplify}\} \\
 & u \ \wedge \ \neg turn \ \wedge \ v \ \wedge \ turn \\
\Rightarrow & \{\text{predicate calculus}\} \\
 & false
\end{array}
$$

Using the methods of chapter 6, we can show that once $u.h$ holds, $m = 3$ holds eventually (and similarly for process v); i.e., every process waiting to enter its critical section will do so eventually.

A new two-process mutual exclusion algorithm

We suggest a different implementation of shared queue q. As before, let u and v be *true* whenever the corresponding process-id is in the queue. We introduce a boolean variable p that is *true* when the queue has v as its head (i.e., the queue is "v" or "v u") and *false* when the queue has u as its head; the value of p is immaterial if the queue is empty.

Note that p has different values when $q =$"v u" and when $q =$"u v". Thus, all five possible queue values are distinguished by u, v, and p. Also, note that p is more defined than *turn* of the previous algorithm: when the queue has two elements, the two variable values match; when the queue has one element, *turn*'s value is irrelevant, though p's value is determined; for empty queue both variable values are irrelevant.

The tests and assignments in both processes are transformed as follows.

$q = \langle\rangle$	is	$u, v \ = \mathit{false}, \mathit{false}$
$q := q +\!\!+$ "u"	is	$p, u := v, \mathit{true}$
$q := q +\!\!+$ "v"	is	$p, v := \neg u, \mathit{true}$
$q.head =$"u"	is	$\neg p$
$q.head =$"v"	is	p
$q := q.tail$	is	$p, u := \mathit{true}, \mathit{false}$ {in process u}
$q := q.tail$	is	$p, v := \mathit{false}, \mathit{false}$ {in process v}

Appending "u" to q in $q := q +\!\!+$ "u" causes q to become either "u" or "v u". In either case, u is to be set to *true*. In the first case, p is set to *true* and in the second to *false*; in either case p acquires the value of v. The test $q.head =$"u" is $\neg p$, from the definition of p. Removing the head item of q in process u causes q to become either "v" or $\langle\rangle$; in either case, p can be set to *true*. The other transformations are similarly justified.

After applying all the transformations, we obtain the program shown next.

---

program *MutualExclusionRefined2*
  boolean $u, v = \mathit{false}, \mathit{false}$;
  boolean p;

process u

loop
  noncritical section;
  $p, u := v, \mathit{true}$;
  $\neg p \ \rightarrow \ \mathit{skip}$;
  critical section;
  $p, u := \mathit{true}, \mathit{false}$
end

process v

loop
  noncritical section;
  $p, v := \neg u, \mathit{true}$;
  $p \ \rightarrow \ \mathit{skip}$;
  critical section;
  $p, v := \mathit{false}, \mathit{false}$
end

end {*MutualExclusionRefined2*}

This program has the advantage over Peterson's that exactly one boolean variable has to be checked in the guarded command. Unfortunately, the program requires assignments of the form $p := v$ and $p := \neg u$, naming shared variables on both sides of an assignment, which are difficult to implement as atomic actions.

The multiple assignment statements can be replaced by the following sequences of single assignments; see [134, note 13] and also see the note on page 28.

$p, u := v, \mathit{true}$ by $u := \mathit{true}$; $p := v$
$p, v := \neg u, \mathit{true}$ by $v := \mathit{true}$; $p := \neg u$

2.4.6 Shortest path

Dijkstra's shortest path algorithm [56] has by now become a classic (the cited paper is officially designated "classic" by the Citation Index Service). Typical descriptions (and derivations) of this algorithm start by postulating that the shortest paths be enumerated in the order of increasing distances from the source. In this section, we present a derivation that is quite different in character. We view the problem as the computation of a "greatest solution" of a set of equations. We prescribe an action system whose implementation results in Dijkstra's algorithm.

The bulk of the work in our derivation is in designing the appropriate heuristics that guarantee termination (i.e., reaching a fixed point); this is in contrast to traditional derivations, where most of the effort is directed toward postulating and maintaining the appropriate invariant.

The shortest path problem

Given is a finite directed graph that has (1) a source node, henceforth, designated by s, and (2) for each edge (i, j) a non-negative real number, w_{ij}, called its *length*. The length of a path is the sum of the edge lengths along the path. It is required to compute the shortest path, i.e., a path of minimum length, from s to every node. Henceforth, *shortest path to a node* means the shortest path from s to that node, and *distance* to a node is the length of the shortest path. The distance to a reachable node from s is a non-negative real number and the distance to an unreachable node is ∞. For the moment, assume that every node in the graph is reachable from s; the general case, where some of the nodes are unreachable, is taken up on page 34. We restrict ourselves to computing the distances to all nodes; a minor modification of this algorithm can be used to compute the shortest paths.

Equations for distances

Let D_k denote the distance to node k; this is a non-negative real number since all nodes are reachable from s. Note that $D_s = 0$. Call i a *predecessor* of k if there is an edge (i, k) in the graph. For node k, $k \neq s$, whose only predecessors are i and j,

$$D_k = min(D_i + w_{ik},\ D_j + w_{jk})$$

This is because the shortest path to k passes through either i or j, and any initial segment of a shortest path is a shortest path to the corresponding node. Therefore, D —where D is a vector, with the nodes ordered in some fixed manner— is the unique solution for the unknowns d in the following equations E.

$$\begin{aligned} \text{E::}\quad & d_s = 0 \\ & \langle \forall j :\ j \neq s :\ d_j = (min\ i :\ i \text{ is a predecessor of } j :\ d_i + w_{ij}) \rangle \end{aligned}$$

Since all nodes are assumed to be reachable from s, every j, $j \neq s$, has a predecessor.

Inequalities for distances; relaxing the equations

An equation of the form

$$d_k = min(d_i + w_{ik},\ d_j + w_{jk})$$

implies that $d_k \leq d_i + w_{ik}$, and $d_k \leq d_j + w_{jk}$. We convert E into a set of such inequalities, one for each edge. Let F be the system of inequalities so constructed along with the equation $d_s = 0$.

$$\begin{aligned} \text{F::}\quad & d_s = 0 \\ & \langle \forall (i, j) :\ (i, j) \text{ is an edge} :\ d_j \leq d_i + w_{ij} \rangle \end{aligned}$$

It is clear that any solution of E —the only solution of E is D— is a solution of F. However, F may have many more solutions; for instance, a vector of all zeroes is a solution of F.

The distance vector is the greatest solution of F

Define a partial order $\sqsubseteq$ over vectors as follows. For vectors u and v,

$$u \sqsubseteq v \;\equiv\; \langle \forall i :: \; u_i \leq v_i \rangle$$

Theorem (GS) Distance vector D is the greatest solution of F. That is, D is a solution of F and for any solution d, $d \sqsubseteq D$.
Proof: As stated earlier, D is a solution of F. We prove that $d \sqsubseteq D$ for any solution d of F. Let h_j be the number of edges in the shortest path to node j; if there are several shortest paths to j, then one with the fewest edges determines h_j. Since every node is reachable from s, h_j is defined for all j. First, we prove the following proposition by induction on natural numbers.

$$\begin{array}{ll} \text{H::} & \langle \forall n :: \\ & \quad (\forall j :: \; h_j = n \;\Rightarrow\; d_j \leq D_j) \\ & \rangle \end{array}$$

Case $n = 0$**:** We have to show that $\langle \forall j :: \; h_j = 0 \Rightarrow d_j \leq D_j \rangle$. From the definition of h,

$$\begin{array}{ll} & (h_j = 0) \;\Rightarrow\; (j = s) \\ \Rightarrow & \{\text{from F, } (j = s) \Rightarrow (d_j = 0). \text{ Also, } (j = s) \Rightarrow (D_j = 0)\} \\ & (h_j = 0) \;\Rightarrow\; (d_j = 0 \;\wedge\; D_j = 0) \\ \Rightarrow & \{\text{arithmetic}\} \\ & (h_j = 0) \;\Rightarrow\; (d_j \leq D_j) \end{array}$$

Case $n + 1$**:** Assume for all k, $h_k = n \;\Rightarrow\; d_k \leq D_k$. We show that $h_j = (n+1) \;\Rightarrow\; d_j \leq D_j$. From equations E and the definition of h, node j has a predecessor i such that $D_j = D_i + w_{ij}$ and $h_j = h_i + 1$. Since $h_j = (n+1)$ we have $h_i = n$.

$$\begin{array}{ll} & d_j \\ \leq & \{d \text{ is a solution of F; hence } d_j \leq d_i + w_{ij}\} \\ & d_i + w_{ij} \\ \leq & \{h_i = n \;\Rightarrow\; d_i \leq D_i, \text{ from induction hypothesis}\} \\ & D_i + w_{ij} \\ = & \{D_j = D_i + w_{ij}\} \\ & D_j \end{array}$$

Now, for every node j there is some n such that $h_j = n$; hence, from H, $d_j \leq D_j$, for every j.

Unreachable nodes So far our treatment has assumed that all nodes are reachable from s. Under that assumption there is a unique solution to E, which is the distance to the nodes. If there are unreachable nodes, there are several solutions to E: for instance, let u and v be distinct nodes, different from s, that are each other's predecessor, they have no other predecessors, and the lengths of the two edges, (u, v) and (v, u), are both zero. Then E yields the equations $d_u = d_v$ and $d_v = d_u$, permitting these variables to be set arbitrarily.

It can be shown that D is the greatest solution of E in that case, and theorem (GS) is still valid. The proof of H $\Rightarrow \langle \forall j :: \ d_j \leq D_j \rangle$ (in the proof of GS) will have a case distinction for reachable and unreachable nodes. □

Computing the distances

We first suggest a naive method for obtaining the greatest solution of F. The inequality corresponding to edge (i, j) is $d_j \leq d_i + w_{ij}$. This inequality is equivalent to the equation $d_j = min(d_j, d_i + w_{ij})$. Convert this equation to the following action for edge (i, j):

$$S_{ij}:: \ d_j := min(d_j, \ d_i + w_{ij}).$$

Note that the only effect of executing S_{ij} is possibly to decrease d_j.

The execution strategy is to start in a state where $d_s = 0$ and $d_j = \infty$ for all j, $j \neq s$, and to execute an arbitrary action in each step, ensuring that every action is executed eventually. We show that eventually the distances are computed, i.e., $d = D$.

fixed point, invariant

Execution of $d_j := min(d_j, d_i + w_{ij})$ has no effect iff $d_j = min(d_j, d_i + w_{ij})$, or $d_j \leq d_i + w_{ij}$, i.e., a fixed point is a solution of F. From theorem (GS), $d \sqsubseteq D$ at any fixed point.

Below, we show that $D \sqsubseteq d$ is an invariant of the proposed execution. Coupled with $d \sqsubseteq D$ at a fixed point, we have $d = D$ at any fixed point reached by this execution (recall that $\sqsubseteq$ is a partial order). In the next section we address the question of reaching a fixed point.

We can derive that **initially** $D \sqsubseteq d$ because, **initially** $d_s = 0$ and **initially** $d_j = \infty$, for all j, $j \neq s$. We show that execution of any action

$$S_{ij}:: \ d_j := min(d_j, \ d_i + w_{ij})$$

preserves $D \sqsubseteq d$. Execution of S_{ij} affects only d_j; therefore, it is sufficient to show that $D_j \leq d_j$ is a post-condition of S_{ij} given that $D \sqsubseteq d$ is a pre-condition. Applying the axiom of assignment (see appendix A.4.1), we have to show that $D_j \leq min(d_j, \ d_i + w_{ij})$ is a pre-condition of S_{ij}. Prior to execution of S_{ij},

$$
\begin{array}{ll}
 & D_j \\
= & \{\text{From } D \sqsubseteq d\text{: } D_j \leq d_j\} \\
 & min(d_j, D_j) \\
\leq & \{D \text{ is a solution of F; hence } D_j \leq D_i + w_{ij}\} \\
 & min(d_j,\ D_i + w_{ij}) \\
\leq & \{\text{From } D \sqsubseteq d\text{: } D_i \leq d_i\} \\
 & min(d_j,\ d_i + w_{ij})
\end{array}
$$

Reaching a fixed point

It can be shown that picking an arbitrary action for execution, as long as every action is executed eventually, reaches a fixed point in a finite number of steps. However, this strategy is wasteful because it may consecutively repeat execution of an action even though such executions have no effect.

Define the *measure* of action S_{ij} to be d_i. Call an action *active* if its measure has changed since its last execution; the action is *idle* otherwise. More formally, initially all actions are active. An action becomes idle by being executed; an idle action becomes active only if its measure changes. Therefore, idle action S_{jk} may become active as a result of executing S_{ij}, for some i, if it changes d_j.

It follows that (1) execution of an idle action does not change the program state; (2) therefore, if all actions are idle, the program state is a fixed point; (3) execution of S_{ij} can activate an idle action of the form S_{jk} provided that $d_j > d_i + w_{ij}$ holds prior to the execution of S_{jk}, because the measure d_j of S_{jk} changes only under this condition.

We propose that only active actions be picked for execution. Such a computation reaches a fixed point. We propose next a refinement of this strategy and prove its correctness.

Refinement of the execution strategy: BF strategy

Dijkstra's algorithm is the implementation of the following breadth-first strategy. We show that this strategy reaches a fixed point.

BF strategy Pick an active action of smallest measure (among all active actions) for execution.

Observation BF strategy has the following properties.

1. An idle action remains idle.

2. The following proposition C is invariant:

   C:: measure of any idle action $\leq$ measure of any active action.

Proof: Proposition C holds initially because there is no idle action.

Let S_{ij} be an active action of smallest measure, chosen for execution in a step. First, we show that all idle actions remain idle. We need consider only idle actions of the form S_{jk} because execution of S_{ij} can change only d_j, and thus possibly make S_{jk} active. Prior to S_{ij}'s execution, the measure d_j of S_{jk} is at most the measure d_i of S_{ij}, from invariant C. Therefore, execution of S_{ij},

$$S_{ij}::\ d_j := min(d_j,\ d_i + w_{ij})$$

does not change d_j, leaving S_{jk} idle.

Now we prove that C holds after execution of S_{ij}. Before execution of S_{ij}, the measure of any idle action $\leq d_i$ (from C and that S_{ij} was active). Execution of S_{ij} does not change the measure of any idle action (see preceding paragraph), and it makes S_{ij} idle. Therefore, after execution of S_{ij}, the highest measure for any idle action is d_i. The lowest measure for any active action before execution of S_{ij} was d_i. Execution of S_{ij} may change the measure d_j for an active action of the form S_{jk} to $d_i + w_{ij}$. Hence, every active action's measure $\geq d_i$, thus preserving C.

Since each step increases the number of idle actions (the active action chosen for execution becomes idle), a fixed point is reached eventually.

Implementation of the BF strategy

We show that the BF strategy can be implemented in $O(n^2)$ time, where n is the number of nodes.

Let S_{ij} be an active action of smallest measure. Then, from the definition of measure, any active action S_{ik} also has the smallest measure, because both these measures are equal to d_i. Further, execution of S_{ij} leaves an active S_{ik} with the smallest measure active: execution of S_{ij} can possibly change d_j to $d_i + w_{ij}$, which is at least d_i, the measure of S_{ik}. Therefore, we propose that once an active action S_{ij} of smallest measure is identified, then S_{ik} for all k be executed (if S_{ik} is idle its execution has no effect).

The implementation strategy is (1) find i such that S_{ij} is an active action of smallest measure and (2) execute S_{ik} for all k.

Call (1,2) above a *superstep* with node i. Such a superstep makes S_{ik} for all k idle, and they remain idle forever. Call node i idle if S_{ik} for all k are idle; i is active otherwise. The proposed implementation strategy guarantees that if S_{ij} is chosen in a superstep, i is active, and following the superstep i is permanently idle.

A superstep may be implemented in $O(n)$ time. Associate a label, idle or active, with each node; initially all nodes are active. Scan the list of d-values to locate an active node i such that d_i is lowest among all active nodes; this is an $O(n)$ computation. Then execute S_{ik} for all k and mark i idle; this is again an $O(n)$ computation. Since i remains idle afterward, there are exactly n supersteps before all nodes (and actions) become idle. Hence, the entire algorithm is implemented in $O(n^2)$ time.

2.5 Concluding Remarks

This chapter discussed program components, i.e., boxes, whose interactions with their environments are minimal. Typically, the environment sets the values of certain shared variables, simulating a message send or a method call; a box reads the shared variable values and resets them to simulate return of values and indicate the completion of its computation. The emphasis in this chapter has been on the computational aspects, for which invariants and fixed points provide the logical foundation.

Programming with shared variables is error-prone unless considerable care is exercised, say, by restricting the shared variables to behave like message sequences (i.e., channels). The general programming model we develop in the next chapter includes method calls as the sole means of communication.

2.6 Bibliographic Notes

For theories of action systems see Back and Kurki-Suonio [14, 15], Chandy and Misra [32], Lamport [118], and Lynch and Tuttle [124]. The execution rule for action systems proposed here is from [32]. Meseguer [130] has extensively developed the theory and practice of term rewriting systems in a manner akin to action systems, and he reports impressive performance numbers. Action systems have been used effectively in designs of large-scale software in industry; see Pizzarello [152], and Creveuil and Roman [50].

The mutual exclusion problem was introduced by Dijkstra [57]. Program *MutualExclusionRefined* on page 28 closely resembles the algorithm from Peterson [151]. The refinements of mutual exclusion algorithms shown in this chapter are from Misra [134, note 13]; Dappert-Farquhar [52] reports an error in this note and its correction. The shortest path algorithm in section 2.4.6 is from Dijkstra [56]. The development of the shortest path algorithm given in this chapter is from Misra [140]; portions of that paper are reprinted here with permission from Elsevier Science.

3

An Object-Oriented View of Action Systems

3.1 Introduction

Action systems are used in chapter 2 to represent a message communicating process (*Merge*), a fragment of an operating system (mutual exclusion), a process controller (*odometer*), and even solutions to combinatorial problems (gcd, shortest path). The syntax and semantics of action systems are sparse, yet we developed succinct programs for several well-known problems. This chapter extends the programming model of chapter 2 to make it easier to describe process interactions. Additionally, we address the issue of program composition in some detail.

The typical mode of process interaction in chapter 2 is through shared variables. For instance, in the odometer example of section 2.4.2, a boolean variable $c0$ is shared by the odometer with its environment. The environment sets $c0$ to *true* to indicate that the value of the odometer should be incremented, and the odometer program sets $c0$ to *false* on completion of the incrementation. In this chapter, we eliminate shared variables altogether, replacing them with "remote procedure calls". Thus, the odometer program will include a procedure for incrementation that can be called by its environment. Fortunately, a procedure is quite similar to an action, so we need to extend the model of action systems only minimally to accommodate procedure calls. We designate certain actions in a box to be *methods*, and methods can be executed only when called. A box resembles an object in object-oriented programming, though it may include *actions* —which are executed autonomously, as before— in addition to the methods.

There is a fundamental issue related to method calls. In wide-area computing, in particular, a caller should not be made to wait indefinitely (i.e., for an unbounded number of computation steps); if there is no guarantee that the method can be executed successfully to completion, the call should be *rejected*. This is in contrast to current practices in concurrent computing where a process that requests a resource is made to wait, perhaps forever, if the resource is not available. Guaranteed termination of a method call is an essential requirement of our theory; we insist that execution of every action terminate, and this requirement can be met only if every method called by an action is also guaranteed to terminate.

This model of programming is called *Seuss*. Seuss is basically an action system with a minimal amount of additional machinery to facilitate interactions among its components. Our examples in this chapter and the next attest to the effectiveness of this model of programming on a variety of computing problems.

Overview of the model

We use the **box** construct from chapter 2 that plays the role of an object. As before, a program consists of a set of boxes, though there are no shared variables. Typically, a user defines generic boxes, called **cat**s (*cat* is short for *category*), and creates several boxes from each cat through instantiation. A cat is similar to a class; a box is similar to a class instance.

The state of a box is given by the values of its variables. The variables are local to the box. Therefore, their values can be changed only by the steps taken within the box. Each box includes a set of **procedures**, where a procedure is either an **action** or a **method**. A method is called by a procedure of another box; method call is the only mechanism for interactions among boxes. An action is not called like a traditional procedure; it is executed from time to time under the following fairness rule: each action is eventually executed. Both actions and methods can change the state (values of the variables) of their own box and possibly of other boxes by calling their methods. A method may have parameters; an action does not have any parameter.

A method may *accept* or *reject* a call made upon it. If the state of the box does not permit a method to be executed —for instance, a *get* method on a channel cannot be executed if the channel is empty— then the call is rejected. Otherwise, the call is accepted. Some methods accept every call; such methods are called *total* methods. A method that may reject a call is called a *partial* method. Similarly, we have total and partial actions.

3.2 Seuss Syntax

In this section we introduce a notation for writing programs. The notation is intended for implementation on top of a variety of host languages, each of which should provide facilities for sequential programming. Therefore, no commitment has been made in the syntax about the sequential programming aspects.

Notational Conventions

The notation is described using BNF. All nonterminal identifiers are in Roman and all terminal identifiers are in boldface type. The traditional meta symbols of BNF, ::= { } [ ] ( ), are used, along with $\vee$ to stand for alternation (the usual symbol for alternation, "|", is a terminal symbol in our notation). The special symbols used as terminals are | // ; : :: $\rightarrow$ in the syntax given below. A syntactic unit enclosed within "{" and "}" in a production may be instantiated zero or more times, and a unit within "[" and "]" may be instantiated zero or one time. In the right-hand side of a production, $(p \vee q)$ denotes that a choice is to be made between the syntactic units p and q in instantiating this production; the parentheses, "(" and ")" , are omitted when no confusion can arise. Text enclosed within "{" and "}" in a program is to be treated as a comment. The traditional keywords of sequential programming appear in small boldface, e.g, **if**, **then**, **else**, **endif**, **do**, **while**, and **enddo**.

3.2.1 Program

program ::= **program** program-id {cat $\vee$ box} **end**
cat ::= **cat** cat-name [parameters]: {variable} {procedure} **end**
box ::= **box** box-id [parameters]: cat-id [arguments]

A program consists of a set of cats and boxes. The declaration of a cat or box includes its name and possibly parameters. The ids of programs, cats, and boxes are simple identifiers. The parameters of a cat or box can be ordinary variables, cats or boxes (see program *MutualExclusion1* on page 47 for an example of cat declaration with parameters). A cat consists of zero or more variable declarations followed by procedure declarations. A box is an instance of a cat. We adopt the convention that several boxes may be instantiated under one "box" declaration. Variables are declared and initialized in a cat as in traditional programming languages.

Example

We use a single running example to illustrate the syntax of Seuss. A ubiquitous concept in multiprogramming is a *Semaphore*. The skeletal program given below includes a definition of *Semaphore* as a cat and two instances of *Semaphore*, boxes s and t. Cat *user* describes a group of users that execute

their critical sections only if they hold both semaphores, s and t; there are three instances of *user*.

---

program *MutualExclusion*
  cat *Semaphore*
    nat $n = 1$ {initially, the semaphore value is 1};
    {The procedures of *Semaphore* are to be included here}
  end {*Semaphore*}

  box s, t : *Semaphore*

  cat *user*
    boolean $hs, ht = false, false$;
    {hs is *true* when *user* holds s. Similarly, ht.}
    {The procedures of *user* are to be included here}
  end {*user*}

  box u, v, w : *user*
end {*MutualExclusion*}

3.2.2 Procedure

procedure ::= partial-procedure ∨ total-procedure
partial-procedure ::= **partial** partial-method ∨ partial-action
total-procedure ::= **total** total-method ∨ total-action
partial-method ::= **method** head :: partial-body
partial-action ::= **action** [label] :: partial-body
total-method ::= **method** head :: total-body
total-action ::= **action** [label] :: total-body

A procedure is either **partial** or **total**, and it is either a **method** or an **action**. Thus, there are four possible headings identifying each procedure. Each method has a head and a body. The head is similar to the form used in typical imperative languages; it has a procedure name followed by a list of parameters and their types. The labels are optional for actions; they have no effect on program execution.

The convention for parameter-passing is call by value-result. First, the arguments of the procedure call are stored into the parameters; the parameters are treated as local variables of the called procedure. Upon termination of the execution of the called procedure the parameter values are assigned to the arguments.

Example (continued)

We add the procedure names to the previous skeletal program.

---

program *MutualExclusion*
  cat *Semaphore*
    nat $n = 1$ {initially, the semaphore value is 1};
    partial method P:: {Body of P goes here}
    total method V:: {Body of V goes here}
  end {*Semaphore*}

  box s, t : *Semaphore*

  cat *user*
    boolean $hs, ht = false, false$;
    partial action $s.acquire$:: {acquire s and set hs to *true*.}
    partial action $t.acquire$:: {acquire t and set ht to *true*.}
    partial action *execute*::
      {execute body if both hs, ht are *true*. Then set hs, ht to *false*.}
  end {*user*}

  box u, v, w : *user*
end {*MutualExclusion*}

3.2.3 Procedure body

A procedure body has different forms for partial and total procedures. In this book, a total-body is any sequential program; it may include calls upon total methods of other boxes. The partial-body is defined by:

partial-body ::= alternative {(| alternative) ∨ (⊬ alternative)}
alternative ::= pre-condition [; pre-procedure] → total-body
pre-condition ::= predicate
pre-procedure ::= partial-method-call

The body of a partial procedure consists of one or more alternatives; we defer discussion of | and ⊬ to section 3.2.4. Each alternative has a pre-condition, an optional pre-procedure and a total-body. A pre-condition is a predicate that may name only the procedure parameters and the local variables of the box in which the procedure appears. A pre-procedure is a partial method of some other box.

Next, we show examples of partial procedures that have only single alternatives. A discussion of multiple alternatives appears in section 3.2.4, and examples are given in section 3.2.5.

Example (continued)

Below, we include code for each procedure body in *MutualExclusion*. The partial actions $s.acquire$ and $t.acquire$ in *user* call on partial methods $s.P$ and $t.P$ as pre-procedures. Partial action *execute* in *user* calls the total methods $s.V$ and $t.V$ in its body. Partial action P in *Semaphore* has no pre-procedure.

---

program *MutualExclusion*
  cat *Semaphore*
    nat $n = 1$ {initially, the semaphore value is 1};
    partial method P:: $n > 0 \ \rightarrow \ n := n - 1$
    total method V:: $n := n + 1$
  end {*Semaphore*}

  box s, t : *Semaphore*

  cat *user*
    boolean $hs, ht = false, false$;
    partial action $s.acquire$:: $\neg hs$; $s.P \ \rightarrow \ hs := true$
    partial action $t.acquire$:: $\neg ht$; $t.P \ \rightarrow \ ht := true$
    partial action *execute*:: $hs \ \wedge \ ht \ \rightarrow$
        critical section; $s.V$; $t.V$; $hs := false$; $ht := false$
  end {*user*}

  box u, v, w : *user*
end {*MutualExclusion*}

The operational semantics of Seuss programs are described in section 3.3. The program given above may become deadlocked, i.e., it may not allow any user to enter its critical section, because one may have acquired s and another t. This problem may be avoided by first acquiring s and then t; i.e., by changing the pre-condition of $t.acquire$ to $hs \wedge \neg ht$.

3.2.4 Multiple alternatives

Each alternative in a partial procedure is *positive* or *negative*: the first alternative is always positive; an alternative preceded by | is positive and one preceded by ⧸ is negative. For each partial procedure at most one of its alternatives holds in any state; i.e., the pre-conditions of the alternatives are pairwise disjoint.

The programmer must ensure that a negative alternative never modifies the value of any argument of its call.

The effect of a procedure call is described in detail in section 3.3. In short, a procedure none of whose alternatives has a *true* pre-condition rejects the call. If an alternative has a *true* pre-condition (there can be at most one, because the alternatives have disjoint pre-conditions), then its pre-procedure is called; if the pre-procedure rejects, the alternative (and this procedure) rejects; and if the pre-procedure accepts, the body of the alternative is executed. The distinction between positive and negative alternative plays a role only after the execution of the body; the call is rejected if it is a negative alternative and accepted if the alternative is positive.

A negative alternative allows the state of the called box to be changed while the caller's state remains unchanged (this is why we required that it should not modify the value of any argument). This is a powerful mechanism to devise starvation-free solutions, such as in implementing a strong semaphore (section 4.9.2).

3.2.5 Examples of alternatives

Use of positive alternatives

Cat *multiplexor* includes method *get*, which returns items from channels $in1$ and $in2$ alternately, starting with $in1$. Variable c is 1 if the next item is to be retrieved from $in1$; $c = 2$ otherwise.

---

cat *multiplexor*
  enum $\{1, 2\}$ $c = 1$;

  partial method *get*(x: type)::
    $c = 1$; $in1.get(x)$ $\rightarrow$ $c := 2$
  | $c = 2$; $in2.get(x)$ $\rightarrow$ $c := 1$
end {*multiplexor*}

The program below attempts to avoid the use of alternatives. Variable y's value is the next item to be returned, or ϕ when it holds no item.

---

cat *multiplexor1*
  enum $\{1, 2\}$ $c = 1$;
  type $y = \phi$;

  partial method *get*(x: type):: $y \neq \phi$ $\rightarrow$ $x, y := y, \phi$
  partial action *get*1:: $c = 1$ $\wedge$ $y = \phi$; $in1.get(y)$ $\rightarrow$ $c := 2$
  partial action *get*2:: $c = 2$ $\wedge$ $y = \phi$; $in2.get(y)$ $\rightarrow$ $c := 1$
end {*multiplexor1*}

The execution of *multiplexor1* does not quite match that of *multiplexor*, because *multiplexor1* implements a one-item look-ahead. In general, alternatives cannot be eliminated. Also, the alternatives in a partial action cannot generally be split into separate actions.

Use of negative alternatives: strong semaphore

A group of users share a semaphore. Each user calls method P persistently in the following sense: if its call is rejected, it calls again. Also, every user that is granted the semaphore relinquishes it eventually. We implement a strong binary semaphore, which guarantees that each caller is eventually granted the semaphore.

The implementation is as follows. Each caller passes its id as an argument to P. A negative alternative is used to record the ids of the callers to P whose calls have been rejected, so that they can be granted the semaphore in the same sequence in which they called P. The program uses the following variables:

q: the sequence of users each of whose last call on P was rejected
avail: a boolean whose value is "the semaphore is available"

---

cat *StrongSemaphore*
  seq(id) $q = \langle\rangle$;
  boolean $avail = true$;

  partial method $P(i\text{: id}) ::$
    $avail \ \wedge \ i = q.head \ \rightarrow \ avail, q := false, q.tail$
    $\not\downarrow \ i \notin q \ \rightarrow \ q := q +\!\!+ i$ {i is appended to q}

  total method $V ::$ $avail := true$
end {*StrongSemaphore*}

The reader can argue operationally that (1) at most one caller is granted the semaphore at any time, and (2) if each caller is persistent, the solution is starvation-free: each caller is eventually granted the semaphore, provided that each caller that is granted the semaphore relinquishes it eventually. Several variations of semaphores are treated in section 4.9.

Note: *StrongSemaphore* rejects the first call on P even though $q = \langle\rangle$. Modify the solution so that P accepts the call in this case. □

Mutual exclusion using negative alternatives

In the example of section 3.2.3, a user acquires semaphores s and t to enter its critical section. Seuss prohibits calling multiple partial methods from a procedure. Thus, it is illegal to write in box *user*

partial action *execute*:: *true* ; $s.P$; $t.P$ $\rightarrow$ *critical section*; $s.V$; $t.V$

This is clearly the intent, though, and we show how to simulate this using negative alternatives. We introduce a cat, *MultiSemaphore*, whose partial method *PP* accepts only if it holds both s and t. Total method VV releases both semaphores. A user calls *MultiSemaphore.PP* repeatedly to enter its critical section and *MultiSemaphore.VV* to release both semaphores. The instance u' of *MultiSemaphore* acts on u's behalf; similarly, v' and w'.

Method *PP* in *MultiSemaphore* first acquires s and then t to avoid deadlock. It rejects a call, using a negative alternative, even if it acquires s. After acquiring s it accepts a call only if it can acquire t.

---

program *MutualExclusion1*
  cat *Semaphore*
    nat $n = 1$ {initially, the semaphore value is 1};
    partial method *P*:: $n > 0$ $\rightarrow$ $n := n - 1$
    total method *V*:: $n := n + 1$
  end {*Semaphore*}

  box s, t : *Semaphore*

  cat *MultiSemaphore*
    boolean $hs, ht = false, false$;
    partial method *PP*::
        $hs \wedge \neg ht$; $t.P$ $\rightarrow$ $ht := true$
      $\not\vert$ $\neg hs$; $s.P$ $\rightarrow$ $hs := true$
    total method *VV*:: $s.V$; $t.V$; $hs := false$; $ht := false$
  end {*MultiSemaphore*}

  cat *user*(*ms* : *MultiSemaphore*)
    partial action *execute*::
      true ; $ms.PP$ $\rightarrow$ *critical section*; $ms.VV$
  end {*user*}

  box u', v', w' : *MultiSemaphore*
  box u : *user*(u'), v : *user*(v'), w : *user*(w')
end {*MutualExclusion1*}

Exercise Devise a general strategy that simulates the effect of having multiple partial procedures in a guard. Use the ideas from the solution given above for mutual exclusion with two semaphores. □

3.2.6 Constraints on programs

Procedure call A total-body can call only total methods; a partial method cannot be called by a total body. A partial method can appear only as a pre-procedure in an alternative of a partial procedure. The syntax specifies that an alternative can have at most one pre-procedure. □

Partial order on boxes Every procedure p should impose a partial order $\geq_p$ over the boxes of the program in the following sense: during an execution of p a procedure of box b can call a procedure of box b' provided that $b >_p b'$ (i.e., $b \geq_p b' \ \wedge \ b \neq b'$), that is, a procedure calls a procedure of a lower box in the order defined by $\geq_p$. See section 10.2.3 for a formal description of this condition. In the example of section 3.2.3, *user* boxes (u, v, w) call upon *Semaphore* boxes (s and t), but not conversely. □

In most cases, all procedures impose the same partial order over the boxes, as is the case in the example of section 3.2.3. However, there are important exceptions for which we allow different procedures to impose different partial orders over boxes; see section 3.4.4.

A consequence of the requirement of partial order is that if some procedure of a box is being executed, then no procedure of that box is called; therefore, at most one procedure from any box is being executed at any moment.

Termination condition Execution of each total body (the body part of any action, total or partial) terminates; the programmer has to prove that this requirement is met by the program. □

The termination condition can be proved by induction on the "level" of a procedure. First, show that any procedure that calls no other procedure terminates whenever it accepts a call. Next, show that execution of a procedure terminates assuming that executions of all procedures it calls terminate.

3.3 Seuss Semantics (Operational)

At run time, a program consists of a set of boxes; their states are initialized at the beginning of the execution. There are two different execution styles for a program. In a *tight execution*, one action is executed at a time. There

is no notion of concurrent execution; each action completes before the next action is started. In a *loose execution*, actions may be executed concurrently.

The programmer understands a program by reasoning about its tight executions only. We have developed a logic for this reasoning. An implementation may choose a loose execution for a program to maximize resource utilization. Loose execution is described in chapter 10, and the correspondence between loose and tight executions is established in that chapter. An implementation for loose execution is suggested in chapter 11.

3.3.1 Tight execution

A tight execution consists of an infinite number of steps; in each step, an action of a box is chosen and executed as described below in section 3.3.2. The choice of action to be executed in a step is arbitrary except for the following fairness constraint: each action of each box is chosen eventually.

Observe that methods are executed only when they are called from other methods or actions, though actions are executed autonomously (and eventually).

3.3.2 Procedure execution

A method is executed when it is called. To simplify the description, imagine that an action is called by a *scheduler*. Then the distinction between a method and an action vanishes; each procedure is executed when called.

A procedure *accepts* or *rejects* a call. A total procedure always accepts calls. A partial procedure may accept or reject a call, as described below. A partial procedure that has a single (positive) alternative is of the form:

partial method $g(x, y)$:: p; $h(u, v) \rightarrow S$

Execution of g can be described by the following rules.

---

if $\neg p$ **then** *reject*
else {p holds} call h with arguments (u, v);
  if h *rejects* **then** *reject*
  else {h *accepts*}
    execute S;
    return *accept* code and parameter values (x, y) to the caller of g
  endif
endif

As stated earlier, the programmer must ensure that execution of each total procedure terminates. It can then be shown that execution of any partial procedure g terminates by using induction on the partial order induced by $\geq_g$ (see section 3.2.6).

The caller is oblivious to rejection, because then its body is not executed and its state remains unchanged. If all alternatives in a program are positive, then the state does not change for any box in case of rejection; only acceptance may cause some box state to change. This is because if any procedure rejects during execution of an action, the entire action rejects. If any procedure accepts —a procedure that has no pre-procedure accepts first, followed by acceptances by its callers in the reverse order of calls— then the entire action accepts. This execution strategy meets the *commit* requirement in database protocols, where a transaction is either executed to completion or not executed at all.

We have described execution of a partial procedure that has a single (positive) alternative. If a procedure has several alternatives, positive and negative, the following execution strategy is adopted. Recall that the pre-conditions of the alternatives are disjoint.

---

if pre-conditions of all alternatives are *false* **then** *reject*
else {pre-condition of exactly one alternative f holds}
  if f is a positive alternative **then** execute as described earlier
  else {f is a negative alternative}
    execute f as a positive alternative except on completion of f:
      reject the call
  endif
endif

Execution of a negative alternative always results in rejection. The caller is still oblivious to rejection, because the caller's body is not executed and its state remains unchanged. However, a called method may change the state of its own box even when it rejects a call, by executing a negative alternative.

For a partial action the effect of execution is identical for positive and negative alternatives because the scheduler does not discriminate between acceptance and rejection of an action. Therefore, partial actions have no negative alternatives.

Effective Execution

For the action systems of chapter 2, execution of an action is *effective* if its guard —which consists only of a pre-condition— holds when it is executed. The definition of effective execution is more involved in the presence of pre-procedures in guards, and particularly with negative alternatives. Now, we discuss when a "call" to a procedure is effective.

Calls to total procedures and accepted calls to partial procedures are always effective; a rejected call may or may not be effective. The general rule for partial procedures is: a call is *ineffective* iff its execution calls a

partial procedure whose alternatives' pre-conditions are all *false*. It can be shown that an accepted call or any call that causes a state change is effective. Therefore, a call that is rejected from a negative alternative may be effective if the body of the alternative is executed. An ineffective execution causes no state change; not every effective execution causes a state change because a procedure may have *skip* as its body.

3.4 Discussion

3.4.1 Total vs. partial procedures

It may seem that total and partial procedures are interchangeable. A total procedure f can be coded as the partial procedure $true \rightarrow f$. Also, a partial procedure can be coded as a total procedure where the outcome of the call —acceptance or rejection— is coded explicitly as an argument that can be tested by the caller.

But the distinction between total and partial procedures is fundamental. Total procedures model terminating computations, i.e., *transformational* aspects of programming, and partial procedures model potentially nonterminating computations, or *reactive* aspects of programming [127]. In this view, a P operation on a semaphore is modeled by a partial procedure —because it may never terminate— whereas a V operation is a total procedure.

The distinction between total and partial procedures is important for concurrent implementation. We show in chapter 10 that two threads can be executed concurrently given that (1) total procedures in different threads commute and (2) partial procedures in each thread *semicommute* with total procedures in the other thread (there is no requirement on the partial procedures of different threads). This condition allows a richer set of concurrent computations, because not all procedures are required to commute.

Total procedure

The body of a total procedure is a wait-free program. A total procedure can be assigned a meaning based only on its inputs and outputs; if the procedure is started in a state that satisfies the input specification, it terminates eventually in a state that satisfies the output specification. Procedures to sort a list, to find a minimum spanning tree in a graph and to send a job to an unbounded print queue are examples of total procedures. A total procedure need not be deterministic; e.g., *any* minimum spanning tree can be returned by the procedure. Further, a total procedure need not be implemented on a single processor; e.g., the list may be sorted by a sorting network [18]. Data parallel programs and other synchronous computation schemes are usually total procedures. A total procedure may even be a

multiprogram in our model that admits of asynchronous execution provided that it is guaranteed to terminate, and its effect can be understood only through its inputs and outputs; therefore, such a procedure never waits to receive input, for instance. An example of a total procedure that interacts with its environment is one that sends jobs to a print queue (without waiting); the jobs may be processed by the environment while the procedure continues its execution. All total procedures shown in this book are sequential programs.

A total procedure may call only total procedures. When a total procedure is called, its execution may (1) terminate normally, (2) continue forever, or (3) fail. Nontermination of a total procedure is the result of a programming error. We require (see Termination condition in section 3.2.6) the programmer to establish that the procedure is invoked only in those states where its execution is finite.

A failure is also caused by a programming error; it occurs when a procedure is invoked in a state in which it should not be invoked, for instance, if the computation requires a number to be divided by 0 or a natural number to be reduced below 0. Failure is a general programming issue, not just an issue in Seuss or multiprogramming. We interpret failure to mean that the resulting state is arbitrary; any step taken in a failed state results in a failed state. Typically, a hardware or software trap aborts the program when a failure occurs.

Example Consider V operation on a binary semaphore. If the semaphore value is 0, execution of V changes it to 1. What happens if the semaphore value is 1 prior to execution of V? There are at least four possible implementations: (1) the operation is interpreted as a *skip* (i.e., the semaphore value remains 1 and the operation terminates), (2) the operation fails, i.e., it changes the semaphore value arbitrarily to either 0 or 1, (3) the operation waits for the semaphore value to become 0 and then it changes it to 1, and (4) the operation never terminates. If we adopt interpretations (1) or (2), we may regard the V operation as a total procedure. With interpretation (3), the operation is viewed as a partial procedure. We insist that the V operation be so implemented that possibility (4) does not arise. In this book, we treat V as a total procedure with meaning (1). □

Partial procedure

Each execution of a partial procedure terminates although the procedure may be called over and over (possibly infinitely often). For instance, in traditional programming, the caller of P on a semaphore waits as long as the semaphore value is 0 (typically, it is placed in a queue to wait for the semaphore to become free). In our model, each call to P terminates, either accepted or rejected. For an action $c; P \rightarrow S$, body S is executed only if P accepts the call; if P rejects the call, S is not executed, and the state of

the caller's box does not change (thus preserving c). To simulate waiting, execution of this procedure is attempted repeatedly as long as c holds.

A partial procedure can call at most one partial procedure, and then as a pre-procedure only. Thus, it is illegal to write

partial action *execute*:: *true* ; $s.P$; $t.P$ $\rightarrow$ ...

The reason for this requirement is that we want to view an action as a unit that is either executed to completion or not executed at all. In the example above, if $s.P$ accepts and $t.P$ rejects, then *execute* should reject and roll back the effect of $s.P$. Rollback requires an elaborate implementation, and that is why we have decided to avoid rollback. Additionally, we allow a partial procedure to appear only as a pre-procedure; i.e., the call on a partial procedure is made *before* the calling procedure has changed its box state; again, this policy avoids rollback in case the pre-procedure rejects the call.

3.4.2 Tight vs. loose execution

In a tight execution, an action is completed before another action is started. This allows a program execution to be understood as a single thread of control, avoiding the complexity of reasoning about interleaved executions of the action bodies. Each procedure, total or partial, may be understood from its text alone, using the semantics of the procedures it calls without consideration of interference by other procedures. A simple temporal logic, such as UNITY logic [32], is suitable for deducing properties of a program in this execution model.

An implementation need not be restricted to a single thread as long as it achieves the same effect as a single-thread execution. In chapter 11, we show how implementations may exploit the structures of Seuss programs (and user-supplied directives) to run concurrent threads of actions with a fine grain of interleaving; these loose executions preserve the semantics of tight execution.

A consequence of having a single thread in a tight execution is that the notion of waiting has to be abandoned, because a thread can afford to wait only if there is another thread whose execution can establish a state in which the first thread can resume execution. Rendezvous-based interactions [92, 132], which require at least two threads of control to be meaningful, do not appear in this model of computation. We have replaced waiting by the refusal of a procedure to execute, i.e., by rejection.

3.4.3 The Seuss programming methodology

In the Seuss model, we view a multiprogram as a set of *actions*, where each action deals with one aspect of the system functionality, and *execution of*

an action is wait-free. Additionally, we specify the conditions under which an action is to be executed.

Seuss partitions the multiprogramming world into (1) programming of action bodies whose executions are wait-free, and (2) specification of the conditions for orchestrating executions of the action bodies. Different theories and programming methodologies are appropriate for these two tasks. In particular, if the action bodies are sequential programs, traditional sequential programming methodologies may be adopted for their developments. The orchestration of the actions has to employ some multiprogramming theory, but it is largely independent of the action bodies. Seuss addresses the design aspects of multiprograms only —i.e., how to combine actions— not the designs of the action bodies.

Seuss severely restricts the amount of control available to the programmer at the multiprogramming level. The component actions of a program can be executed only through infinite repetitions. In particular, sequencing of two actions has to be implemented explicitly. Such loss of flexibility is to be expected when controlling larger abstractions. For an analogy, observe that machine language offers complete control over all aspects of a machine operation: the instructions may be treated as data, data types may be ignored entirely and control flow may be altered arbitrarily. Such flexibility is appropriate when a piece of code is very short; then the human eye can follow arbitrary jumps, and "mistreatment" of data can be explained away in a comment. Flowcharts are particularly useful in unraveling intent in a short and tangled piece of code. However, at higher levels control structures for sequential programs are typically limited to sequential composition, alternation, and repetition; arbitrary jumps have nearly vanished from all high-level programming. Flowcharts are of limited value at this level of programming, because intricate manipulations are dangerous when attempted at a higher level, and prudent programmers limit themselves to appropriate programming methodologies to avoid such dangers. We expect the structuring operators to be even simpler at the multiprogramming level. That is why we propose that the component actions of a multiprogram be executed using only a form of repeated nondeterministic selection.

3.4.4 Partial order on boxes

A partial order on the boxes of a program is imposed by each procedure. This is in contrast to the usual views of process networks, in which the processes communicate by messages or through a shared store. Typically, such a network is not regarded as being partially ordered. For instance, suppose that process P sends messages over channel cp to process Q and Q sends over cq to P. The processes are viewed as nodes in a cycle where the edges (channels), cp and cq, are directed between P and Q. Similar remarks apply to processes communicating through shared memory. We view communication media (message channels and memory) as boxes. Therefore, we

represent this system as a set of four boxes —P, Q, cp, and cq— with the procedures (*send* and *receive*) in cp and cq being called from P and Q but not vice versa. The direction of message flow is immaterial in this hierarchy. A partial order is extremely useful in deducing properties by induction on the "levels" of the procedures.

The restriction that procedure calls be made along a partial order implies that a partial procedure at a lowest level is of the form $p \rightarrow S$, where the pre-procedure is absent. A total procedure at a lowest level calls no procedure.

We have introduced some flexibility by permitting each procedure to impose a partial order over the boxes rather than have a single partial order that is obeyed by all procedure calls. To see why this flexibility is essential, consider a *user* who sets an alarm clock (*AlarmClock*) to ring at a specified time, and *AlarmClock* notifies the *user* by ringing at the specified time. Let the *user* and the *AlarmClock* be coded as boxes; an action *set* in *user* calls a procedure in *AlarmClock* to set the clock, and a procedure *tick* in *AlarmClock* calls *WakeUp* in *user* to wake up the user at the specified time. So there is no fixed order of these two boxes: *set* orders *user* higher than *AlarmClock*, and *tick* orders them in the opposite manner. See section 4.4 for a more detailed treatment of this problem.

3.5 Concluding Remarks

The programming model described in this chapter incorporates ideas from transaction processing [78], serializability and atomicity in databases [20], notions of objects [131], communicating sequential processes [92], i/o automata [124, 125], and temporal logic of actions [118]. A partial procedure is similar to a database (nested) transaction that may commit or abort; the procedure commits (to execute) if its pre-condition holds and its pre-procedure commits, and it aborts otherwise. A typical abort of a database transaction requires a rollback to a valid state. In Seuss, a rejected call does not require a rollback (though the call may change the program state).

The form of a partial procedure is inspired by communicating sequential processes [92]; a pre-procedure is a receive operation in that model. A box may also be viewed as a monitor, see Hoare [90]; see also Brinch Hansen [23, 24] and Dijkstra [59] for concepts similar to monitor. However, unlike monitor procedures Seuss procedures never suspend their executions; execution of a procedure always completes either by accepting or rejecting the call. The *wait* and *signal* operations of monitors have been eliminated in Seuss in favor of an optimizing scheduler; see section 11.7.

Seuss is an outgrowth of our earlier work on UNITY [32], which models action systems. The UNITY commands were particularly simple —

assignments to program variables— and the model allowed few programming abstractions besides asynchronous compositions of programs.

Seuss is an attempt to build a compositional model of multiprogramming, retaining some of the advantages of UNITY. An action is similar to a statement, though we expect actions to be much larger in size. We have added more structure in Seuss, by distinguishing between total and partial procedures, grouping the actions within boxes, and allowing procedure calls among them. Executing actions as indivisible units would extract a heavy penalty in performance; so we have developed a theory that permits interleaved executions of the actions. Programs in UNITY interact by operating on a shared data space; however, Seuss boxes have no shared data, and they interact only through procedure calls. As in UNITY, issues of deadlock, starvation, progress (liveness), etc. can be treated by making assertions about the sequence of states in every execution. Also, as in UNITY, program termination is not a basic concept. A program reaches a *fixed point* when the pre-conditions of all actions are *false*; further execution of the program does not change its state, and an implementation may then terminate the execution. This book contains an enhanced version of UNITY logic that is applicable to Seuss programs.

3.6 Bibliographic Notes

The first paragraph of the previous section contains most of the general references on which this work is based. A preliminary version of this model appears in Misra [141]. Browne [25] and Newton and Browne [144] have developed and implemented parallel programming models in which parallel and sequential aspects of computing are separated, as in Seuss. No attention is paid in this book to the development of total procedure bodies; we refer the reader to Hehner [87] for a treatment of sequential programming methodology. A calculus of objects is introduced in Abadi and Cardelli [1] that develops a number of important ideas dealing with the semantics of objects and their typing rules.

A practical programming language based on Seuss needs (1) a module structure to associate scopes with names; (2) features for importing and exporting cats and boxes; and (3) features for a program to interact with programs written in other notations, particularly for input and output. These issues have been addressed in two implementations of Seuss on C^{++}, see Krüger [109], and Java, see Alvisi et al. [8] and Joshi [98]. Particularly important for practical programming are libraries; these implementations support the building of libraries as modules that export cats.

4
Small Examples

A number of small examples are treated in this chapter. The goal is to show that typical multiprogramming examples from the literature have succinct representations in Seuss. Additionally, the small number of features of Seuss is adequate for solving many well-known problems: communications over bounded and unbounded channels, maintaining a database, implementing a caching strategy, mutual exclusions and synchronizations, and resource allocation. We show a number of variations of some of these examples, implementing different progress guarantees, for instance.

Notational conventions

Single instance of a cat

A cat C with only one instance is abbreviated by declaring a box b with the body of C appended to it. This eliminates explicit introduction of cat name C.

Quantification

A notation for quantified expressions is described in appendix A.2.1. Here, we extend the notation to permit constructions of a (bounded) number of alternatives of a procedure and a (bounded) number of procedures. In all cases, the form of a quantification is as follows: $\langle \otimes x : \ q(x) : \ e(x) \rangle$, where $\otimes$ is any commutative, associative binary operator, x is the *bound* variable (or a list of bound variables), $q(x)$ is a predicate that determines the *range* of the bound variables, and $e(x)$ is an expression called the *body*. We extend

the operator $\otimes$ to $[\!]$, $|$, and $/\!/$, and the body to include alternatives and procedures. Only actions, not methods, may be quantified. Two examples are shown below.

$\langle [\!]\ i : 0 \leq i < N :$
  partial action $get(i)$:: $c.i;\ sem[i].P\ \rightarrow\ B.i$
$\rangle$

partial action::
  $\langle |\ i : 0 \leq i < N : b.i\ \rightarrow\ pos.i$
  $/\!/\ i : N \leq i < 2 \times N : b.i\ \rightarrow\ neg.i$
  $\rangle$

The first **partial action** declaration creates N partial actions, named $get(i)$, $0 \leq i < N$. Action $get(i)$ includes i as a parameter in its pre-condition $c.i$, pre-procedure $sem[i]$ and body $B.i$. The last **partial action** declaration creates N positive and N negative alternatives. Each alternative may include i as a parameter in its pre-condition, pre-procedure, and body.

4.1 Channels

4.1.1 Unbounded fifo channel

An unbounded first-in–first-out (fifo) channel is a cat that has two methods: total method *put* (i.e., send) appends an element to the end of the message sequence, and partial method *get* (i.e., receive) removes and returns the head element of the message sequence, provided that it is nonempty. The channel is represented below by variable r that is the sequence of values sent but not yet received. We define a polymorphic version of the channel where the message type is left unspecified.

---

cat *FifoChannel*(type)
  seq(type) $r = \langle\rangle$ {r is initially empty};
  partial method $get(x$: type) :: $r \neq \langle\rangle\ \rightarrow\ x, r := r.head, r.tail$
  total method $put(x$: type) :: $r := r +\!\!+ x$ {append x to r}
end {*FifoChannel*}

As an application of *FifoChannel*, shown below is a box whose partial action, *transfer*, copies the elements of *in* to *out*, both being boxes of *FifoChannel* of integer. Since *transfer* is executed repeatedly, every element of *in* is eventually transferred to *out* (assuming that no other box removes items from *in*).

---

box *copy*
  integer x;
  partial action *transfer*:: *true*; $in.get(x) \rightarrow out.put(x)$
end {*copy*}

The following box is similar to *copy* except that it includes two partial actions, to read from either $in1$ or $in2$ and output to *out*; boxes $in1$, $in2$, and *out* are instances of *FifoChannel*. Since the two actions in *merge* are executed infinitely often, this box implements a fair merge of $in1$ and $in2$ (again, we assume that no other box removes items from $in1$ or $in2$).

---

box *merge*
  integer x;
  partial action *transfer1*:: *true*; $in1.get(x) \rightarrow out.put(x)$
  partial action *transfer2*:: *true*; $in2.get(x) \rightarrow out.put(x)$
end {*merge*}

4.1.2 Bounded fifo channel

We show a bounded fifo channel of size N, $N > 0$, below. Here both procedures *put* and *get* are partial. The messages are kept in a circular buffer, b. Let $\oplus$ denote addition mod N; f is the index of the oldest message, r is the index of the youngest message $\oplus$ 1, and k is the total number of messages in the channel.

---

cat *bch*(N: nat, type)
  array$[0..N-1]$**(type)** b;
  enum $(0..N)$ $f, r, k = 0, 0, 0$ {initially the channel is empty};
  partial method *put*(x: type)::
      $k < N \rightarrow r, b[r], k := r \oplus 1, x, k+1$

  partial method *get*(x: type)::
      $k > 0 \rightarrow f, x, k := f \oplus 1, b[f], k-1$
end {*bch*}

As a special case, consider a channel that can hold at most one message, i.e., a bounded channel with $N = 1$. The sender writes into the channel only when the channel is empty, and writing makes the channel full. The receiver reads and removes from the channel only when it is full, thereby making the channel empty. Thus, the sender receives an acknowledgment

(that its previous output has been received) when it is able to write into the channel. As before, both *get* and *put* are partial methods. Variable w holds the contents of the channel, if any, and *full* is *true* iff w contains data.

---

cat *word*(type)
  type w;
  boolean *full* = *false* {the channel is initially empty};
  partial method *put*(x: type):: $\neg$*full* $\rightarrow$ w, *full* := x, *true*
  partial method *get*(x: type):: *full* $\rightarrow$ x, *full* := w, *false*
end {*word*}

As an application of bounded channels, consider the following variation of an example from Hoare [91]. A *multiplexor* process receives a stream of messages from 10 different *consoles*. It acknowledges each message it receives and sends the received message along a single output channel. A *console* may terminate the stream by sending a special end-of-stream (*eos*) message.

The solution in [91] uses rendezvous-based communication that eliminates the need for acknowledging receipt of a message. We achieve a similar effect by requiring that a *console* and the *multiplexor* communicate over a bounded channel of size 1, i.e., a *word*. Then each *console* is assured that its last message has been received if it is able to send another message. This is slightly inferior to rendezvous-based communications where the buffer size is zero and each communication is instantly acknowledged.

The *multiplexor* and *console* i, $0 \leq i \leq 9$, communicate via $c[i]$, where $c[i]$ is a *word*. The *multiplexor* sends its outputs along a *FifoChannel* called *out*. Variable $more[i]$ is *true* iff the *multiplexor* has not yet received an *eos* message from channel i.

---

box *multiplexor*
  type m;
  array[0..9]**(boolean)** *more* = *true*;

  $\langle [\!] i : 0 \leq i \leq 9 :$
    partial action::
      $more[i]$; $c[i].get(m)$ $\rightarrow$ $out.put(m)$; $more[i] := (m \neq eos)$
  $\rangle$
end {*multiplexor*}

There is no restriction on the order in which the partial actions in the *multiplexor* are executed. The fairness constraint ensures that any message sent by a *console* is eventually received and output by the *multiplexor*.

4.1.3 Unordered channel

The fifo channel guarantees that the order of delivery of messages is the same as the order in which they were put into the channel. Next, we consider an unordered channel in which *any* message from the channel is returned in response to a call on *get*, provided that the channel is nonempty. The channel is implemented as a bag, and *get* is implemented as a nondeterministic operation. We write $x :\in b$ to denote that x is assigned any value from bag b, provided that b is nonempty. The usual notation for set operations is used for bags in the following example.

---

cat *uch*(type)
  bag(type) $b = \emptyset$ {initially b is empty};

  partial method *get*(x: type) :: $b \neq \emptyset \ \rightarrow \ x :\in b;\ b := b - \{x\}$

  total method *put*(x: type) :: $b := b \cup \{x\}$
end {*uch*}

This implementation does not guarantee that every message will eventually be delivered, even if messages are removed from the bag an unbounded number of times. Such a guarantee is, of course, established by the fifo channel. We propose a solution below that implements this additional guarantee. In this solution every message has an *index* —a natural number— and variable t is at most the smallest index. A message is assigned an index strictly exceeding t whenever it is put in the channel; the indices of different messages need not be distinct. The *get* method removes any message with the smallest index and assigns to t the index of the removed message.

---

cat *nch*(type)
  bag(*index***: nat,** *msg***: type)** $b = \emptyset$ {initially b is empty};
  nat $t = 0$;
  nat s;
  type m;

  partial method *get*(x: type)::
    $b \neq \emptyset \ \rightarrow$ remove pair (s, m) with minimum index s from b;
    $t, x := s, m$

  total method *put*(x: type)::
    $b := b \cup \{(s, x)\}$, where s is any natural number, $s > t$
end {*nch*}

Now we show that every message is removed eventually given that there are an unbounded number of calls on *get*. For a message with index i we show that the pair $(i-t, p)$, where p is the number of messages in the bag with index t, decreases lexicographically with each accepting execution of *get*, and it never increases. Hence eventually $i = t$ and $p = 0$, implying that this message has been removed. An execution of *put* does not affect i, t, or p, because the added message receives an index higher than t; thus, $(i-t, p)$ does not change. A *get* either increases t, thus decreasing $i-t$, or keeps t the same and decreases p, thus decreasing $(i-t, p)$.

In section 7.4.1 we show how to assign an arbitrary natural number to n so that any natural number is a possible value of n. Let s be assigned the value $t + n + 1$ in method *put*; then any value that exceeds t is a possible value of s. The proposed solution is *maximal* in the sense that any possible order of removal of items from b can be implemented in *nch* with appropriate choices for the values assigned to s. This claim is proved in section 7.5.

4.1.4 Task dispatcher

A task *dispatcher* interacts with a set of *client*s and *server*s. A *client* generates a sequence of tasks each with a *priority* between 0 and N. A *server* requests the *dispatcher* for a task whenever it is idle. A *server* can process tasks of priority p or lower, for some p. The *dispatcher* responds to a request from a *server* by sending it a task that the *server* can process.

A *dispatcher* is nothing but a glorified channel; it has two methods, *put* and *get*. A *client* calls *put*, with a task and its priority as arguments, to deposit the task in the channel. A *server* calls *get* with some priority p for the *dispatcher* to send it a task of priority p or lower, if such a task exists.

Below, $r[i]$ is a queue of tasks of priority i that are pending. Method *get* returns a task of the highest priority that a server can process.

---

cat *dispatcher*
  array$[0..N]$ **seq(task)** $r = \langle\rangle$;
  enum $(0..N)$ i;

  partial method *get*(x: task, $p : 0..N$)::
    {get a task of priority p or lower, as close to p as possible}
    $(\exists j :: \ 0 \le j \le p \ \wedge \ r[j] \neq \langle\rangle) \ \rightarrow$
      $i := (max \ j : \ 0 \le j \le p \ \wedge \ r[j] \neq \langle\rangle : \ j)$;
      $x, \ r[i] \ := \ r[i].head, \ r[i].tail$

  total method *put*(x: task, $p : 0..N$):: $\ r[p] \ := \ r[p] +\!\!+ x$
end {*dispatcher*}

There is no guarantee that a task will be removed eventually, even though *servers* that can process that task call *dispatcher* repeatedly. This modification of *dispatcher* is left to the reader; use the ideas of section 4.1.3.

4.1.5 Disk head scheduler

The problem of coding a disk head scheduler is as follows. Several *users* desire access to specific tracks on a disk. A *user* submits an access request and then waits until the request is served. A *filter* process accepts requests from the *users*; it imposes an order in which the requests are to be served. The disk is controlled by a *server* process that can serve one access request at a time; the *server* calls filter to receive the next request to be served.

Structure of users and server

The *filter* may be viewed as a channel between the *users* and the *server*. A *user* submits a request by executing the partial action

$$b;\ \mathit{filter}.\mathit{put}(i, r) \ \rightarrow \ \{\text{continue execution}\}$$

Here, b is the condition under which the user makes an access request, the identity of the *user* is i, and r specifies the request (track number, the locations from or into which the data is to be transferred). The *user* is expected to let b remain *true* as long as its request has not been served; therefore, once b becomes *true*, procedure $\mathit{filter}.\mathit{put}(i, r)$ is called repeatedly until it accepts.

The *server* executes a partial action:

$$\mathit{true}; \mathit{filter}.\mathit{get}(r) \ \rightarrow \ \{\text{serve request } r\}$$

to receive the next request from the *filter*. Henceforth, a pair (i, r), where i is a process-id and r is a request, is called a *submission*.

Structure of filter

The code of box *filter* is shown below. The local variables are ps, a set of submissions that have already been served, and rq, a queue of submissions that are yet to be served, arranged according to some ranking protocol. Method *get* (called by the *server*) accepts a call provided that $rq \neq \langle\rangle$; i.e., there is some submission that is yet to be served. In that case, the pair (i, r) at the head of rq is removed and r is returned to the *server*; additionally, (i, r) is added to ps since the submission will be served immediately as part of the action that calls *filter.get*.

Method *put* accepts a call with argument (i, r) only if this submission has already been served, i.e., $(i, r) \in ps$; otherwise, the call is rejected. This is justified given the structure of a *user* program (see the previous paragraph). When a *user* calls *put* for the first time with a fresh submission the submission is queued in rq and the call is rejected. Subsequent calls by

the same *user* are rejected until the submission is served. The solution can handle multiple requests from the same *user*.

---

box *filter*
  seq(id, request) $rq = \langle\rangle$;
  set(id, request) $ps = \emptyset$;

  partial method *get*(r: request)::
    $rq \neq \langle\rangle \;\rightarrow\; (i, r) := rq.head;\; rq := rq.tail;\; ps := ps \cup \{(i, r)\}$

  partial method *put*(i: id, r: request)::
    $(i, r) \in ps \;\rightarrow\; ps := ps - \{(i, r)\}$
  $\not|\;\; (i, r) \notin ps \;\wedge\; (i, r) \notin rq \;\rightarrow\;$ add (i, r) to rq
end {*filter*}

This solution should be contrasted with the ones in [94, 75]. Our code is considerably shorter because process synchronization requirements can be stated very succinctly. Note particularly the manner in which the negative alternative is used in method *put* to ensure that a call from the *user* is accepted only *after* the submission has been processed by the *server*.

4.1.6 Faulty channel

Cat *FaultyChannel*, which simulates message loss, duplication, and out-of-order delivery in a channel, is shown in this section. Such a channel has the usual methods, *put* and *get*, by which the senders and receivers interact with it. The channel may lose messages, it may duplicate any message an unbounded (though finite) number of times, and it may permute the order of messages.

We implement a faulty channel using a bag b, as in *uch* of section 4.1.3, to simulate out-of-order delivery. To simulate message loss and duplication, we associate a natural number n with each message that is added to b; n denotes the number of times that the message is to be delivered. If $n = 0$ for a message, the message is immediately discarded, and if $n > 0$, the message is added n times to b.

The faulty channel, as described, can provide no guarantee of any message transmission at all because all messages may be lost; clearly, no useful device can be built out of such a channel. The next requirement we add is that a message that is *put* repeatedly is eventually delivered, provided that the receiver calls *get* over and over. We implement this requirement by insisting that n become nonzero periodically in method *put*. In section 7.2.1 we show a box that returns a natural number with this property. Below, we use the notation defined for the unordered channel in section 4.1.3: write $x :\in b$ to mean that x is to be assigned an arbitrary value from b.

```
cat FaultyChannel(type)
  bag(type) b = ∅ {initially b is empty};

  partial method get(x: type)::
      {remove one item from the bag if the bag is nonempty}
      b ≠ ∅ → x :∈ b; b := b − {x}

  total method put(x: type)::
      {add x to the bag some random number of times}
      Let n be a fair natural number (see section 7.2.1);
      while n ≠ 0 do
        b := b ∪ {x}; n := n − 1
      enddo
end {FaultyChannel}
```

Now we argue that if *put* is called repeatedly with argument m and *get* is called repeatedly, then m will eventually be returned as a result of *get*. Repeated execution of *put* establishes $n > 0$ eventually, so message m is eventually added to b. Let c be the number of elements of b that differ from m at this point. Whenever a message other than m is returned in a call to *get*, the value of c decreases. Also, no message other than m is added to b by *put*; hence, c does not increase. Therefore, messages other than m can be delivered only a finite number of times before m is returned as the result of a call to *get*.

This fault model of a channel is assumed in the alternating bit protocol [159], without the possibility of message reordering.

It can be shown that the proposed solution is maximal (see section 7.6). That is, it can display any possible behavior of the faulty channel. This is a necessity if this program is to be used as a simulator for a faulty channel.

4.2 A Simple Database

We use a simple database example to illustrate the use of some of the cats introduced so far.

Cat *DataBase* has total methods *insert*, *delete*, and *query* that act upon a stored database D whose elements are of some specified type. Each of these procedures stores one of three possible results in parameter r: *eff* (effective), *error* (error), or *ineff* (ineffective). An *insert* of x has the outcome *eff* if x is not in D (prior to the operation) and there is enough room to add x to D. In this case, x is added to D. The outcome of *insert* is *error* if x is not in D and there is not enough room to insert x; and the outcome is *ineff* if x is already in D. A *delete* of x has an outcome *eff* if x is in D

prior to the operation, and then x is removed from D; the outcome is *ineff*, otherwise. A *query* for x has an outcome *eff* if x is in D; the outcome is *ineff*, otherwise.

---

type outcome = (*eff*, *error*, *ineff*)
cat *DataBase*(type)
  set(type) $D = \emptyset$ {the database is initially empty};

  total method *insert*(x: type, r: outcome)::
    if $x \in D$ **then** $r :=$ *ineff*
      elseif there is room to add x **then** $r :=$ *eff*; add x to D
      else $r :=$ *error*
    endif

  total method *delete*(x: type, r: outcome)::
    if $x \in D$ **then** $r :=$ *eff*; remove x from D
      else $r :=$ *ineff*
    endif

  total method *query*(x: type, r: outcome)::
    if $x \in D$ **then** $r :=$ *eff*
      else $r :=$ *ineff*
    endif
end {*DataBase*}

We create one instance of *DataBase* with item-type "element".

box *store* : *DataBase*(element)

Now consider two users, each of whom sends a stream of requests for operations on *store*. Requests are directed to a box *multiplexor*. The *multiplexor* accepts requests in arbitrary order. An operation's result is a boolean; it is *true* iff the operation was effective. The *multiplexor* also outputs a log of the effective *insert* and *delete* operations, from which the database can be reconstructed.

First, consider the communications between a user and the *multiplexor*. We implement the communication of requests by using *word*, described in section 4.1.2. We create two instances of *word* —*xreq* and *yreq*— for the two users to send requests to the *multiplexor*, by the following declaration.

type request =
  record *op* : (*insert*, *delete*, *query*), n: element **endrecord**
box *xreq*, *yreq* : *word*(request)

Box *multiplexor* creates a log of the effective *insert* and *delete* operations by sending the sequence of effective requests over an unbounded fifo

channel (see section 4.1.1). We create a single box, *log*, for this purpose. Also, we create two instances of *FifoChannel* —*xrep* and *yrep*— for the *multiplexor* to communicate with the two users, sending them the results of their requests.

box log : *FifoChannel*(request)
box $xrep, yrep$: *FifoChannel*(boolean)

Box *multiplexor* consists of two partial actions to read from the channels *xreq* and *yreq*.

```
box multiplexor
  request req;
  outcome r;

  partial action :: true; xreq.get(req) →
    if req.op = insert then store.insert(req.n, r);
      if r = eff then log.put(req) endif
    elseif req.op = delete then store.delete(req.n, r);
      if r = eff then log.put(req) endif
    else {req.op = query} store.query(req.n, r)
    endif ;
    xrep.put(r = eff)

  partial action :: true; yreq.get(req) →
    if req.op = insert then store.insert(req.n, r);
      if r = eff then log.put(req) endif
    elseif req.op = delete then store.delete(req.n, r);
      if r = eff then log.put(req) endif
    else {req.op = query} store.query(req.n, r)
    endif ;
    yrep.put(r = eff)
end {multiplexor}
```

The trail of effective requests may alternatively be stored in a *DataBase* instead of being sent on a fifo channel. In that case, declare *log* to be a box of type *DataBase*(request), and let *multiplexor* insert the effective requests into *log*. This implementation of *DataBase* does not save the sequence in which the data are inserted. Therefore, *multiplexor* will have to add a sequence number explicitly to each effective request before inserting it into *log*.

4.3 Management of Multilevel Memory: Lazy Caching

We show an algorithm called Lazy Caching [6] that maintains consistency among multiple caches. Our description uses different terminology from the original paper; in particular, we have reduced the number of procedures by at least half by using partial procedures. We refer the reader to the original paper for the proof of correctness as well as an extensive discussion of cache consistency.

The entire system consists of a main *memory*, several *processors*, and their associated *caches*. We encode memory, processor, and cache as three separate cats. There is only one instance of memory, but there may be several processors, and there is a cache corresponding to each processor. Additionally, there are two first-in–first-out channels, *in* and *out*, corresponding to each cache.

The memory has two total methods, *Mread* and *Mwrite*; for location a and value d, procedure $Mread(d, a)$ assigns the value at a to d, and $Mwrite(d, a)$ assigns d to a.

A *LazyCache* is interposed between a processor and the main memory. A processor calls $read(d, a)$ and $write(d, a)$ of its *LazyCache* to read and write, respectively, from location a. Each instance of *LazyCache* includes a local variable C that represents the cache memory, which is a subset of the main memory; $C(a)$ denotes the value at location a in C. Each *LazyCache* has two integer variables, *lenout* and $leninT$; the former is the number of items in *out* and the latter is the number of *true* items in *in* (see below).

The clever idea in lazy caching is to let a processor continue after a $write(d, a)$; the pair (d, a) is simply appended to *out*. A *read* is delayed until all previous writes by this processor have been processed.

Let u be a *LazyCache*; fifo channels *in* and *out* of u have the following structure. Channel *out* includes pairs of the form (d, a) where d is a value and a is a location. These are the pending write operations on the main memory by u. Only u may apply *put* and *get* on its *out* channel. Channel *in* has triples of the form (d, a, tag) where d is a value, a is a location, and *tag* is a boolean. These are the pending updates to the cache due to writes by u and other *LazyCaches*. Any *LazyCache* may apply *put* on *in* of cache u, though only u may apply *get* on its own *in* channel; a *true* tag signifies that the item was *put* by u as part of a *write*.

A *LazyCache* has two partial actions —*conin* and *conout*— which consume items from *in* and *out*, respectively. Action *conin* removes the head entry (d, a, tag) of *in* and assigns d to cache location a (if the cache does not have a as a location, some entry is removed from the cache to make room for it; this aspect of the algorithm is not shown in the program below). Action *conout* removes the head entry (d, a) of *out*, writes d into location a of the main memory and notifies the other processors —by appending

(d, a, false) to the *in* channel of their caches— that they should update their caches; also, (d, a, true) is appended to the *in* channel of its own cache.

---

cat *LazyCache*
  set(value, location) C;
  integer $\mathit{lenout} = 0$ {number of items in *out*};
  integer $\mathit{leninT} = 0$ {number of items with *true* tag in *in*};

  partial method $\mathit{read}(d, a)$::
    $\mathit{lenout} = 0 \ \wedge \ \mathit{leninT} = 0 \ \wedge \ a \in C \ \rightarrow \ d := C(a)$
  $\not$ $a \notin C \ \rightarrow \ \mathit{Mread}(d, a); \ \mathit{in.put}(d, a, \mathit{false})$

  total method $\mathit{write}(d, a)$:: $\mathit{out.put}(d, a); \ \mathit{lenout} := \mathit{lenout} + 1$

  partial action *conin*::
    $\mathit{true}; \ \mathit{in.get}(d, a, \mathit{tag}) \ \rightarrow$
      $C(a) := d$;
      if *tag* **then** $\mathit{leninT} := \mathit{leninT} - 1$ **endif**

  partial action *conout*::
    $\mathit{true}; \ \mathit{out.get}(d, a) \ \rightarrow$
      $\mathit{lenout} := \mathit{lenout} - 1$;
      $\mathit{Mwrite}(d, a)$;
      $\mathit{in.put}(d, a, \mathit{true})$;
      $\mathit{leninT} := \mathit{leninT} + 1$
      {append (d, a, false) to the *in* channel of all other caches}
      $\langle \forall p : \ p$ is *in* channel of another cache : $\ p.\mathit{put}(d, a, \mathit{false}) \rangle$
end {*LazyCache*}

4.4 Real-Time Controller; Discrete-Event Simulation

A set of users, encoded by boxes $\mathit{user}[0..N]$, communicate with an alarm clock, *AlarmClock*, in the following way. Action *set* in $\mathit{user}[u]$ calls method *set* in *AlarmClock* with argument d; procedure *tick* in *AlarmClock* wakes up $\mathit{user}[u]$ —by calling method *WakeUp*— after d clock ticks.

In section 3.2.6 we imposed the constraint that there is a partial order over boxes so that a procedure may call another procedure only if the latter is from a lower box. The purpose of this example is to illustrate that fixed partial orders do not suffice, because there is no way to order $\mathit{user}[u]$ and *AlarmClock*: *set* in $\mathit{user}[u]$ calls *set* in *AlarmClock*, and *tick* in *AlarmClock* calls *WakeUp* in $\mathit{user}[u]$. That is why we allow each procedure to impose

its own partial order; *set* in *user*[*u*] ranks *user*[*u*] higher than *AlarmClock*, and *tick* ranks them in the opposite order.

In *user*[*u*], boolean variable *sleep* is *true* in the interval after execution of *set* and before execution of *WakeUp*. Predicate *c* in action *set* is the condition under which the user sleeps. The exact value of *d*, the length of the interval for which the user sleeps, is irrelevant for this example; we assume *d* is set by some other procedure of *user*[*u*] that is not shown here.

```
box user[u: id]
  boolean sleep = false;

  partial action set::
      integer d;
      c ∧ ¬sleep → AlarmClock.set(u, d); sleep := true

  total method WakeUp:: sleep := false
end {user}
```

Next, we design the *AlarmClock*. It has a local integer variable *time*, and it includes a method *tick*, which is called periodically to advance *time*. Whenever *time* is advanced, any wake-up calls scheduled then are executed. In the following solution, we have adopted a simple implementation of the event list: for each time *i*, *event*[*i*] is a list of users that have to be woken up at *i*.

```
cat AlarmClock
  integer time = 0;
  array[0..M] seq(id) event = ⟨⟩;

  total method set(u, d)::
    enum (0..N) j;
    j := time + d; event[j] := event[j] ++ u

  total method tick::
    time := time + 1;
    while event[time] ≠ ⟨⟩ do
      u := event[time].head;
      user[u].WakeUp;
      event[time] := event[time].tail
    enddo
end {AlarmClock}
```

4.4.1 Discrete-event simulation

A particularly simple view of discrete-event simulation is that method *tick* is called to advance *time* as soon as all the wake-up calls have been processed for the current value of *time*. Thus, *time* represents the value of a virtual clock. The counterpart of *AlarmClock* in simulation is an event-list manager. Such a box includes methods to enqueue an event for future processing and dequeue scheduled events; also, it includes an action to process the next scheduled event and advance *time*. Note that *time* need not be incremented in steps of 1; instead, it can be advanced to the point where the next event is scheduled.

4.5 Example of a Process Network

It is required to compute the sequence of integers of the form $2^i \times 3^j \times 5^k$ in increasing order, for all natural numbers i, j, k. This example illustrates how a network of processes may be employed to solve a combinatorial problem.

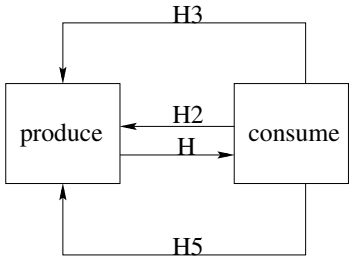


Figure 4.1: Network to compute $2^i \times 3^j \times 5^k$

The computation strategy is as follows. Let H denote the sequence to be computed. Then,

$$H = \langle 1 \rangle +\!\!+ \mathit{merge}(2 \times H,\ 3 \times H,\ 5 \times H),$$

where function *merge* merges its argument sequences, each of which is increasing, to form an increasing sequence (*merge* drops the duplicates from its arguments). This equation has a unique solution in H.

The equation is implemented by the network shown in Fig. 4.1. Box *produce* receives $2 \times H,\ 3 \times H,\ 5 \times H$ along *FifoChannel*s $H2, H3, H5$, respectively, and it produces the desired sequence along H by merging its inputs; initially, H has the integer 1 in it. Box *consume* removes items from H; for a removed item h it sends $2 \times h,\ 3 \times h,\ 5 \times h$ along the *FifoChannel*s $H2, H3, H5$, respectively.

Program *Hamming* is shown below. Box *produce* is same as program *Merge* of section 2.4.4, except for a change in channel names. This box

has three variables, $h2, h3, h5$, where $h2$ is the last number received along $H2$ that is yet to be sent along H; if all numbers received have already been output, then $h2$ is 0; $h3, h5$ have similar meanings. Box *produce* has two kinds of partial actions, *read* and *write*. A *read* action, say *read*2, receives the next value from $H2$ provided that $h2 = 0$. Procedure *write* outputs along H the smallest of $h2, h3$, and $h5$ when they are all nonzero, and updates them appropriately. The computation is started by having 1 in channel H, initially; this is accomplished by executing method $put(1)$ in H. Box *consume* contains a single action that reads an input h from H and outputs $2 \times h,\ 3 \times h,\ 5 \times h$ along *FifoChannel*s $H2, H3, H5$, respectively.

---

program *Hamming*
  box $H2, H3, H5$: *FifoChannel*(integer)
  box H: *FifoChannel*(integer) **init** $put(1)$

  box *produce*
    nat $h2, h3, h5 = 0, 0, 0;$

    partial action *read2*:: $h2 = 0;\ H2.get(h2)\ \rightarrow\ skip$
    partial action *read3*:: $h3 = 0;\ H3.get(h3)\ \rightarrow\ skip$
    partial action *read5*:: $h5 = 0;\ H5.get(h5)\ \rightarrow\ skip$

    partial action *write*::
      nat f;
      $h2 \neq 0\ \wedge\ h3 \neq 0\ \wedge\ h5 \neq 0\ \rightarrow$
        $f := min(h2, h3, h5);\ H.put(f);$
        if $f = h2$ **then** $h2 := 0;$
        if $f = h3$ **then** $h3 := 0;$
        if $f = h5$ **then** $h5 := 0$
  end {*produce*}

  box *consume*
    nat h;
    partial action::
    true; $H.get(h)\ \rightarrow$
      $H2.put(2 \times h);$
      $H3.put(3 \times h);$
      $H5.put(5 \times h)$
  end {*consume*}

end {*Hamming*}

4.6 Broadcast

We show a cat that implements broadcast-style message communication. A set of *writers* attempt to broadcast a sequence of values to a set of *readers*. A new value can be broadcast only if all previous values have been read by all *readers*. Cat *broadcast* synchronizes the reads and writes as follows.

The value to be broadcast is stored in variable v, n is the number of *readers* that have read v, and N is the total number of *readers*. Both *read* and *write* are partial methods.

The pre-condition for *read* is that this particular *reader* has not read the current value of v. To implement the pre-condition for reading, we associate a sequence number with the value of v. It is sufficient to have a one-bit sequence number, a boolean variable t, as in the alternating bit protocol for communication over a faulty channel [159]. A *reader* calls *read* with a boolean argument s that is the last sequence number read by this *reader*. If s and t match, the *reader* has already read this value, so the call on *read* is rejected. If s and t differ, the *reader* is allowed to read the value and both s and n are updated. It is easy to show that n equals the number of *readers* whose s-value equals the cat's t-value. Initially, the local variable s for each *reader* is *true* and v contains no unread value.

Writing is permitted only when all *readers* have read the value of v, i.e., $n = N$. The boolean sequence number t is reversed whenever a new value is written to v.

---

cat *broadcast*(type)
  type v;
  enum $(0..N)$ $n = N$;
  boolean $t = true$;

  partial method *read*(s: boolean, x: type)::
    $s \neq t \ \rightarrow \ s, x, n \ := \ t, v, n + 1$

  partial method *write*(x: type):: $n = N \ \rightarrow \ t, v, n \ := \ \neg t, x, 0$
end {*broadcast*}

Ticket

Variable s used by a *reader* encodes its state. However, this variable is never read or set by the *reader*. We call such a variable a *ticket* in this book; tickets are similar to "cookies" used by several Web browsers. In secure communication, a process is sometime given an encrypted message by a trustworthy server that it is supposed to send to another party. The message behaves like a ticket because its contents cannot be examined or changed by the process.

4.7 Barrier Synchronization

A group of concurrently executing *user* processes perform their computations in a sequence of phases. A *user* may start executing its phase $p+1$ only after all *users* have completed their phase p, $p \geq 0$. Each *user* has a variable k, the highest phase that this *user* has completed; **initially** $k = 0$ for all *users*. Partial method *sync* of cat *barrier* is called by each *user* with argument k in order for it to advance to phase $k+1$. The protocol for a *user* is shown below.

---

box *user*
  nat $k = 0$;
  partial action ::
    true; *barrier.sync*(k) $\rightarrow$ do next phase
      {Effective execution of *barrier.sync*(k) increases k by 1.}
end {*user*}

In the solution below, cat *barrier* has a variable p, which is the highest phase that all users have completed. If a *user* calls *sync* with argument k, the call should be accepted provided that $k = p$, i.e., all *users* have completed phase k. To update p, we need to know how many *users* have completed phase p; we accomplish that as follows.

Let the number of *users* be N. Let n be the number of *users* that have not yet started their phase $(p+1)$. Then, n is decremented whenever *sync* accepts a call. If n becomes 0, then all *users* have completed phase p, so p is set to $p+1$ and n to N.

---

box *barrier*
  enum $(0..N)$ $n = N$;
  nat $p = 0$;

  partial method *sync*(k: nat)::
    $k = p$ $\rightarrow$ $k, n := k+1, n-1$;
      if $n = 0$ **then** $p, n := p+1, N$ **endif**
end {*barrier*}

We leave it to the reader to show that $p \leq k \leq p+1$ is an invariant of this program, for k of any *user*. Therefore, $k = p$ may be evaluated by comparing the lowest bits of k and p. Note that incrementation of a variable —k and p in particular— causes its lowest bit to be inverted. We rewrite the solution using the booleans s and t that represent the lowest bits of k and p, respectively.

---

box *user1*
  boolean $s = true$;
  partial action ::
    $true$; $barrier.sync(s)$ $\rightarrow$ do next phase
end {*user1*}

---

box *barrier1*
  enum $(0..N)$ $n = N$;
  boolean $t = true$;

  partial method $sync(s$: boolean)::
    $s = t$ $\rightarrow$ $s, n := \neg s, n - 1$;
      if $n = 0$ **then** $t, n := \neg t, N$ **endif**
end {*barrier1*}

Note Variables k and s are tickets; s encodes the state of a *user*. No process other than the *barrier* process may change the values of these variables. □

4.8 Readers and Writers

We consider the classic readers-writers problem [49] in which a common resource —say, a file— is shared among a set of *readers* and *writers*. Any number of *readers* can have simultaneous access to the file, but a *writer* needs exclusive access. There are two partial methods, *StartRead* and *StartWrite*, by which a *reader* and a *writer*, respectively, gain access to the resource. The *readers* and *writers* release the resource by calling total methods *EndRead* and *EndWrite*. Reading and writing are finite, so each accepted *StartRead* is eventually followed by an *EndRead* and a *StartWrite* by *EndWrite*. Let nr and nw be the number of active *readers* and *writers*.

---

box *ReaderWriter*
  integer $nr, nw = 0, 0$;
  partial method *StartRead* :: $nw = 0$ $\rightarrow$ $nr := nr + 1$
  partial method *StartWrite*:: $nr = 0$ $\wedge$ $nw = 0$ $\rightarrow$ $nw := 1$
  total method *EndRead* :: $nr := nr - 1$
  total method *EndWrite* :: $nw := 0$
end {*ReaderWriter*}

4.8.1 Guaranteed progress for writers

The solution given above can make no guarantee of progress for either the *readers* or the *writers*. Our next solution guarantees that *readers* do not permanently overtake *writers*: if there is a waiting *writer*, some *writer* eventually gains access to the resource. The strategy is to reject calls on *StartRead* if some *writer* is attempting to execute *StartWrite*. Boolean variable *WriteWait* is set to *true* whenever a call on *StartWrite* is rejected because there are active *readers*. We do not discuss the correctness of this solution because a more general case is treated next. However, we note that *writers* may permanently overtake the *readers* in this solution.

---

box *ReaderWriter1*
  integer $nr, nw = 0, 0$;
  boolean *WriteWait* = *false*;

  partial method *StartRead* ::
    $nw = 0 \ \wedge \ \neg$ *WriteWait* $\rightarrow$ $nr := nr + 1$

  partial method *StartWrite* ::
    $nr = 0 \ \wedge \ nw = 0 \rightarrow$ $nw := 1$; *WriteWait* := *false*
  $\not$ $nr \neq 0$ $\rightarrow$ *WriteWait* := *true*

  total method *EndRead* :: $nr := nr - 1$

  total method *EndWrite* :: $nw := 0$
end {*ReaderWriter1*}

4.8.2 Guaranteed progress for readers and writers

The next solution guarantees progress for both *readers* and *writers*; it is similar to the previous solution — introduce a boolean variable *ReadWait* analogous to *WriteWait*. However, the analysis is considerably more complicated in this case. We outline an operational argument for the progress guarantees; a formal proof appears in section 12.2.2.

We argue that if *WriteWait* is ever *true*, it will eventually be falsified, ensuring that a call upon *StartWrite* will accept eventually. Similarly, if *ReadWait* is ever *true*, it will eventually be falsified. To prove the first claim, consider the state in which *WriteWait* is set to *true* (initially *WriteWait* is *false*). Since $nr \neq 0$ is a pre-condition for such an assignment, a read operation is under way, so $nw = 0$. Once $\neg$ *WriteWait* holds, no further call on *StartRead* will be accepted, and successive calls on *EndRead* will eventually establish $nr = 0$. Also, as long as there are no active *writers*,

ReadWait will remain *false* because *ReadWait* is set to *true* only when $nw \neq 0$. Thus, eventually,

$$nr = 0 \ \wedge \ nw = 0 \ \wedge \ \neg \mathit{ReadWait} \ \wedge \ \mathit{WriteWait}$$

will hold. No method other than *StartWrite* can be executed effectively in this state: none of the guards of the alternatives of *StartRead* holds; no call on *EndRead* or *EndWrite* will be made because no read or write operation is under way, from $nr = 0 \wedge nw = 0$. Therefore, a call on *StartWrite* will be accepted, which will falsify *WriteWait*. The argument for eventual falsification of *ReadWait* is similar.

---

box *ReaderWriter2*
  integer $nr, nw = 0, 0$;
  boolean $\mathit{WriteWait}, \mathit{ReadWait} = \mathit{false}, \mathit{false}$;

  partial method *StartRead* ::
    $nw = 0 \ \wedge \ \neg \mathit{WriteWait} \ \rightarrow \ nr := nr + 1; \ \mathit{ReadWait} := \mathit{false}$
  $\not\mid$ $nw \neq 0 \ \rightarrow \ \mathit{ReadWait} := \mathit{true}$

  partial method *StartWrite* ::
    $nr = 0 \wedge nw = 0 \wedge \neg \mathit{ReadWait} \rightarrow \ nw := 1; \ \mathit{WriteWait} := \mathit{false}$
  $\not\mid$ $nr \neq 0 \ \rightarrow \ \mathit{WriteWait} := \mathit{true}$

  total method *EndRead* :: $nr := nr - 1$

  total method *EndWrite* :: $nw := 0$
end {*ReaderWriter2*}

A number of additional facts about this program are worth noting. *ReadWait* is never set to *true* if there are active *readers*, and *WriteWait* is never set to *true* if there is an active *writer*. This is essential. Otherwise, consider the scenario in which there are several active *readers*, and a *writer* calls *StartWrite*, setting *WriteWait* to *true*, and then, a *reader* calls *StartRead* setting *ReadWait* to *true*. In this state, both *ReadWait* and *WriteWait* are *true* and they will remain *true* forever, preventing *StartRead* and *StartWrite* from ever accepting a call.

4.8.3 Starvation-freedom for writers

Our final variation guarantees absence of starvation for the *writers* but no progress guarantees for the *readers*. We identify a *writer* by including its process-id as an argument in the call to *StartWrite*. A queue of *writer* ids, *wq*, is maintained and *StartWrite* accepts a call only if $nr = 0 \wedge nw = 0$

holds and the caller is at the head of the queue. The test on variable *WriteWait* is replaced by a test on the queue length. In the following, "pid" stands for the type of process-id.

---

box *ReaderWriter3*
  integer $nr, nw = 0, 0$;
  seq(pid) $wq = \langle\rangle$ {wq is initially empty};

  partial method *StartRead* ::
    $nw = 0 \ \wedge \ wq = \langle\rangle \ \rightarrow \ nr := nr + 1$

  partial method $StartWrite(i$: pid) ::
    $nr = 0 \ \wedge \ nw = 0 \ \wedge \ i = wq.head \rightarrow \ nw := 1; \ wq := wq.tail$
  $\not| \ i \notin wq \qquad\qquad\qquad\qquad\quad \rightarrow \ wq := wq +\!\!+ i$

  total method *EndRead* :: $nr := nr - 1$
  total method *EndWrite* :: $nw := 0$
end {*ReaderWriter3*}

A solution that guarantees absence of starvation for both *readers* and *writers* is slightly more involved. One strategy is to create a single queue in which the list of *reader* and *writer* ids are kept for the calls that have been rejected; subsequent calls are accepted in order of appearance in this queue. Consecutive *readers* in the queue are permitted to have simultaneous access to the resource.

4.9 Semaphore

A binary semaphore, often called a *lock*, is typically associated with a resource such as a file, device, or communication channel [58]. A process has exclusive access to a resource only when it *acquires* (or *holds*) the corresponding semaphore. A process acquires a semaphore by completing a P operation, and it releases the semaphore by executing a V. We regard P as a partial method and V as a total method.

Traditionally, a semaphore is *weak* or *strong* depending on the guarantees made about the eventual success (i.e., acceptance) of the individual calls on P. For a weak semaphore, no guarantee can be made about a particular process acquiring the semaphore no matter how many times it attempts P, though it can be asserted that a call on P by some process is accepted if the semaphore is available. Thus, a specific process may be *starved*; it is never granted the semaphore even though another process may hold it arbitrarily many times. A strong semaphore avoids individual (process) starvation; if

the semaphore is available infinitely often, it is eventually acquired by each process attempting the P operation. We discuss both types of semaphores and show several subtle variations in their implementations.

In section 4.9.3, we introduce *snoopy semaphore*, a new kind of semaphore. Unlike a typical semaphore that is first acquired and then released after its associated resource has been used, the holder of a snoopy semaphore releases the semaphore only if some other process requests it. This is a useful strategy if there is low contention for the resource, because a process may then use the resource as long as it is not required by the other processes.

We restrict ourselves to binary semaphores in all cases; extensions to general semaphores are straightforward; see section 3.2.3 for an example of a general weak semaphore.

4.9.1 Weak semaphore

In the following, *semaphore* is a weak binary semaphore, and *user* shows how a semaphore s is called.

---

cat *semaphore*
  boolean *avail* = *true*; {initially the semaphore is available}
  partial method P :: *avail* $\rightarrow$ *avail* := *false*
  total method V :: *avail* := *true*
end {*semaphore*}

---

box *user*
  partial action :: c; $s.P$ $\rightarrow$ use s's resource; $s.V$
  {other actions of the box}
end {*user*}

Usually, once pre-condition c in *user* becomes *true*, it remains *true* until the process acquires the semaphore. However, there is no requirement in Seuss that c will remain *true* as described. This feature can be used to acquire either of the semaphores s and t, as shown below.

---

box *user*
  partial action :: c; $s.P$ $\rightarrow$ c := *false*; use s's resource; $s.V$
  partial action :: c; $t.P$ $\rightarrow$ c := *false*; use t's resource; $t.V$
  {other actions of the cat}
end {*user*}

The implementation of the weak semaphore does not impose any restriction on its callers; for instance, a process that does not hold the semaphore may release it maliciously by executing a V operation, thereby causing the semaphore to be acquired by a process while another process is still holding it. The previous implementation is appropriate when the semaphore is shared among processes that are trustworthy. The more elaborate implementation, shown below, permits only the semaphore-holder to invoke the V operation successfully. If a non-holder attempts V, there is no effect (another possibility would be to make the program fail in this case). In the following, "pid" is the type of process-id; each caller on P or V supplies its id as an argument. Value of *holder* is the id of the process that holds the semaphore; *holder* is *nil* if no process holds the semaphore.

---

cat *semaphore1*
  pid $holder = nil$ {initially the semaphore is not held}
  partial method $P(i$: pid$)$ ::
    $holder = nil \ \rightarrow \ holder := i$
  total method $V(i$: pid$)$ ::
    if $holder = i$ **then** $holder := nil$ **endif**
end {*semaphore1*}

In this solution a process cannot transfer a semaphore it holds to another process, the latter to release the semaphore subsequently. Next, we use a ticket: an accepted call on P returns a ticket (an arbitrary positive integer) and a call on V has effect only if it is attempted by the ticket holder. Let $PN.pnat(j)$ return a positive integer in j, and any positive integer is a possible output. See section 7.4 for an implementation of a box AN so that $AN.anat(k)$ returns a natural number in k and *any* natural number is a possible output; box PN may be similarly implemented (or, $PN.pnat(j)$ may call $AN.anat(k)$ and set j to $k+1$).

---

cat *semaphore2*
  nat $holder = 0$ {initially the semaphore is not held};

  partial method $P(i$: nat$)$ ::
    nat j;
    $holder = 0 \ \rightarrow \ PN.pnat(j); \ i, holder := j, j$

  total method $V(i$: nat$)$ ::
    if $holder = i$ **then** $holder := 0$ **endif**

end {*semaphore2*}

This solution still does not guarantee that the holder has exclusive access to the resource. A determined intruder can attempt to guess the holder's ticket value in a series of attempts. For use in an environment where malice rules, tickets should carry digital signatures [54, 157]. The current implementation may be effective against errors caused by poor programming or hardware malfunction.

None of the implementations shown in this section guarantees absence of individual starvation. Consider a cat that has a partial action of the form

$$c; s.P \;\rightarrow\; \ldots$$

where s is an instance of a weak semaphore and pre-condition c remains *true* as long as $s.P$ is not accepted. It cannot be guaranteed that the call on $s.P$ will ever be accepted. However, we can assert a simple form of progress: if each accepted P is subsequently followed by a call on V, eventually some process's call on P is accepted.

4.9.2 Strong semaphore

A strong semaphore guarantees absence of individual starvation. In Seuss terminology, if a cat contains a partial action of the form $c; s.P \;\rightarrow\; \ldots$, where s is a strong semaphore and pre-condition c remains *true* as long as $s.P$ is rejected, execution of this partial action will eventually be effective. A formal specification of the strong semaphore is given in section 12.5.

The following cat, shown in section 3.2.5, implements a strong semaphore. The call on P includes the process-id as an argument. Procedure P adds the caller-id to queue q if id is not in q, and it grants the semaphore to a caller iff the semaphore is available and the caller-id is at the head of the queue.

---

cat *StrongSemaphore*
  seq(id) $q = \langle\rangle$;
  boolean $avail = true$;

  partial method $P(i$: id) ::
      $avail \;\wedge\; i = q.head \;\rightarrow\; avail, q := false, q.tail$
    $\not\!/\; i \notin q \;\rightarrow\; q := q +\!\!+ i$ {i is appended to q}

  total method V:: $avail := true$
end {*StrongSemaphore*}

Sequence q may be replaced by a fair bag, as was done for the unordered channel *nch* (see section 4.1.3). Note that a call on P is rejected even when the queue is empty and the semaphore is available. It is straightforward to add an alternative to grant the semaphore in this case.

Below, we consider a variation using ticket t as a parameter instead of the process-id. The value of t is the position of the process in the queue, q. A process calls P with t as an argument; t is 0 in the first call by a process. Method P rejects the call, but it sets t to the position of this process in the queue. If P is called with a t-value equal to the head position of the queue and the semaphore is available, the call is accepted. A call on V makes the semaphore available iff the caller shows the appropriate ticket; otherwise, the call has no effect. Since the tickets have consecutive values as integers, we use two variables, f and r, to keep the range of ticket values, i.e., we have

invariant $f \leq r$
the outstanding tickets are the integers i, $f \leq i < r$

Note that $f = r$ implies that there is no outstanding ticket.

---

cat *StrongSemaphore1*
  nat $f, r = 1, 1$;
  boolean *avail* = *true*; {initially the semaphore is available}

  partial method $P(t\text{: nat})$ ::
    $\mathit{avail} \ \wedge \ f = t \ \wedge \ f < r \ \ \rightarrow \ \mathit{avail}, f := \mathit{false}, f + 1$
  $\not\downarrow \ t = 0 \ \ \rightarrow \ t, r := r, r + 1$

  total method $V(t\text{: nat})$::
    if $t = f - 1$ **then** $\mathit{avail} := \mathit{true}$ **endif**
end {*StrongSemaphore1*}

This solution can be made more secure by assigning random integers as ticket values, as was done for the weak semaphore. To guarantee absence of starvation, the tickets are appended to a queue when they are issued.

---

cat *StrongSemaphore2*
  seq(nat) $q = \langle\rangle$;
  nat *holder* = 0 {initially the semaphore is available};

  partial method $P(t\text{: nat})$ ::
    $\mathit{holder} = 0 \ \wedge \ t = q.\mathit{head} \ \rightarrow \ \mathit{holder}, q := t, q.\mathit{tail}$
  $\not\downarrow \ t = 0 \ \rightarrow$ change t so that $t > 0 \ \wedge \ t \notin q$; $q := q +\!\!+ t$

  total method $V(t\text{: nat})$::
    if $\mathit{holder} = t$ **then** $\mathit{avail} := \mathit{true}$ **endif**
end {*StrongSemaphore2*}

A process that requests a semaphore is a *persistent* caller if it calls P eventually following each rejected call. A caller that is not persistent is *transient*. The solutions for the strong semaphore work only if *all* callers are persistent. If there is even a single transient caller, it will block all other callers from acquiring the semaphore. Unfortunately, there exists no implementation for the strong semaphore in this case: there is no guarantee that every persistent caller will eventually acquire the semaphore (given that every holder of the semaphore eventually releases it) in the presence of transient callers [100]. A reasonable compromise is to add a new total method to the strong semaphore that a transient caller may call to remove its process-id from the queue of callers.

4.9.3 Snoopy semaphore

Typically, a semaphore that is associated with a resource is first acquired by a process by calling on P, then the resource is used, and finally the semaphore is released by calling V. We consider a variation of this traditional model of semaphore usage.

A resource is acquired by calling P and released by calling V, as before. However, the resource is not released unless there are outstanding requests by the other processes. This is an appropriate strategy if there is low contention for the resource, because a process may use the resource as long as it is not required by the others.

We describe a new kind of semaphore, called *SnoopySemaphore*, and show how it can be used to solve this problem. In section 4.10.3, we employ the snoopy semaphore to solve a multiple resource allocation problem in a starvation-free fashion.

We adopt the strategy that a process that has acquired and used a resource *snoops* from time to time to see if there is demand for it. If there is demand, it releases the semaphore; otherwise, it may continue to use the resource.

We add a new method, S (for snoop), to *semaphore*. Thus, *Snoopy Semaphore* has three methods: P, V, and S. Methods P and V have the same meaning as for traditional semaphores: a process attempts to acquire the semaphore by calling partial method P and releases it by calling V. The partial method S accepts a call only if the last call on P (by some other process) has been rejected.

A process typically calls S after using the resource at least once, and it releases the semaphore if S accepts. In the following implementation of the snoopy semaphore, boolean variable b is set to *false* whenever a call on P is accepted and *true* whenever a call on P is rejected. Thus, b is *false* when a process acquires the semaphore; if it subsequently detects that b is *true* (then the semaphore is in demand), it releases the semaphore. A weak snoopy semaphore is shown next.

```
cat SnoopySemaphore
  boolean b = false {b is true if the last call on P is rejected};
  boolean avail = true {initially the semaphore is available};

  partial method P ::
    avail      →  avail, b := false , false
  ↯ ¬avail     →  b := true

  total method V:: avail := true

  partial method S:: b  →  skip
end {SnoopySemaphore}
```

In this solution, there is no guarantee that a specific process will ever acquire the semaphore. The next solution is similar to *StrongSemaphore*. Since that solution already maintains a queue of process-ids (whose calls on P were rejected), we can implement S very simply.

```
cat StrongSnoopySemaphore
  seq(pid) q = ⟨⟩;
  boolean avail = true {initially the semaphore is available};

  partial method P(i: pid) ::
    avail ∧ i = q.head  →  avail, q := false , q.tail
  ↯ i ∉ q               →  q := q +| i

  total method V::   avail := true

  partial method S::   q ≠ ⟨⟩  →  skip
end {StrongSnoopySemaphore}
```

The solutions employing tickets can also be used with snoopy semaphores. Note that the two methods S and V may be combined into a single method if every process calls V only after a call on S is accepted. We have retained them as two separate methods to allow a process to release the semaphore unconditionally.

Exercise (Dining philosophers) The following solution to the dining philosophers problem is taken from [29]. The two incident forks on a process (i.e., philosopher) are called *left* and *right*. Each fork has a state, *clean* or *dirty*. Initially, all forks are dirty, a specific process M holds both incident

forks and every other process, except the right neighbor of M holds its left fork.

The solution consists of the following actions. A *hungry* process requests any incident fork it does not hold; on acquiring it, the process cleans it. A *hungry* process that holds both forks transits to *eating*, and then it dirties the forks. A process releases a fork only if it is not *eating*, the fork is dirty, and there is demand for the fork.

Solve the problem encoding each fork as a snoopy semaphore. □

Exercise (Committee coordination) The following problem, taken from [32, chapter 14], combines exclusion and synchronization. A university has a number of *committees*, each of which has one or more *members* (a member is a professor). A member is either *working* or *waiting*. A committee can hold a meeting only if *all* its members are *waiting* (when a meeting is started all its members switch their states to *working*), and no two committees may meet simultaneously if they have a common member. Every meeting ends eventually. Devise a solution so that if all members of a committee are *waiting*, then *some* member attends a meeting, possibly of another committee.

In asynchronous systems, a member is a process, a committee is a synchronization, and a meeting is an occurrence of a synchronization event; in rendezvous-based communication a committee has two members —sender and receiver— and the goal of a meeting is to transfer a message.

Hint: Create a box for each committee and professor. There is an action in each committee to start a meeting; its guard must guarantee that all its members are *waiting*. Employ the technique shown for mutual exclusion using negative alternatives (section 3.2.5), to code the guard succinctly. □

4.10 Multiple Resource Allocation

A typical problem in resource allocation has a set of *resources* and a set of *processes* where each process is in one of three states: *thinking*, *hungry*, and *eating*. A *thinking* process has no need for any resource. A *thinking* process may become *hungry* for exclusive use of a subset of the resources; the specific subset may differ with each *thinking*-to-*hungry* transition for a process. A *hungry* process remains *hungry* until it acquires all the resources it needs; then it transits to the *eating* state. Every *eating* process eventually transits to the *thinking* state; then it releases all resources it holds.

A solution for this problem specifies the steps to be taken by a *hungry* process to acquire all the resources it needs and the protocol for releasing the resources. A solution is *starvation-free* if each *hungry* process eventually transits to *eating*; a solution is *deadlock-free* if some *hungry* process eventually transits to *eating*. We associate a semaphore with each resource, and henceforth we do not distinguish between the semaphore and the resource

it represents. Different kinds of semaphores guarantee different properties of the solutions.

The problem stated above is quite general. If there is a single resource, the problem is equivalent to the mutual exclusion problem where the *critical section* corresponds to the *eating* state: two processes cannot be in the *eating* state simultaneously because the access to the resource would not be exclusive. The problem also subsumes the classical *dining philosophers* problem and its variations [29, 58]: there are an equal number of processes and resources, numbered 0 through N, and process i needs resources i and $i \oplus 1$ when it is *hungry*, where $\oplus$ is addition modulo $(N + 1)$.

We show the code for a generic process. The action for transition from *thinking* to *hungry* is not shown; it is part of an underlying program that sets boolean array *needs*, where *needs*$[i]$ is "the process needs resource i". Also, the transition from *eating* to *thinking* is not shown; every *eating* process eventually transits to *thinking* and *needs* and d remain unchanged by the transition. The resources are numbered 0 through N, and r is an array $[0..N]$ of semaphores, one for each resource. The state of a process —*thinking*, *hungry*, or *eating*— is in variable *state*. We write *thinking* as an abbreviation for *state* = *thinking*; similarly *hungry* and *eating*.

The solutions we propose are all based on well-known algorithms from operating systems: a *hungry* process acquires the resources it needs in increasing order of resource index. Each process has a local variable d such that a *hungry* process has acquired all resources it needs from 0 through $d - 1$. That is, we have the invariant at each process that

$$\begin{array}{l}\langle \forall i : \ 0 \le k < d : \\ \quad (\text{process is } \textit{hungry} \text{ and } \textit{needs}[k]) \ \Rightarrow \ \text{process holds semaphore } k \\ \rangle\end{array}$$

4.10.1 A deadlock-free solution

In the first solution, the semaphores are weak semaphores, i.e., of type *semaphore* from section 4.9.1. A process releases all semaphores it holds when it is in *thinking* state; the order of release is immaterial.

We argue that this solution is deadlock-free. Assume to the contrary that at some point in the computation a set of processes are *hungry* but all processes remain *hungry* forever. Choose a point such that there is no *thinking*-to-*hungry* transition beyond this point, because this transition can happen only a finite number of times before all processes become *hungry*. Since no process eats, no semaphore is released. Consider a *hungry* process j whose d value is d_j. Since it is permanently *hungry*, its calls on $r[d_j].P$ are permanently rejected. Therefore, semaphore d_j is held by a process k. From the invariant, k is *hungry*, $d_j < d_k$, and k is blocked for d_k. Proceeding in this fashion, we can show an infinite sequence of increasing d values, an impossibility.

---

box $r[0..N]$: *semaphore*

box $user_i$
  array$[0..N]$**(boolean)** $needs = false$;
  enum $(0..N+1)$ $d = 0$;
  enum $(thinking, hungry, eating)$ $state = thinking$;

  partial action *acquire* ::
      $hungry \ \wedge \ d \leq N \ \wedge \ \neg needs[d] \ \rightarrow \ d := d+1$
    | $hungry \ \wedge \ d \leq N \ \wedge \ needs[d]; \ r[d].P \ \rightarrow \ d := d+1$

  partial action *eat* ::
      $hungry \ \wedge \ d > N \ \rightarrow \ state := eating$; use resources

  partial action *release* ::
      $thinking \ \wedge \ d > N \ \rightarrow$
        while $d \neq 0$ **do**
          $d := d-1$; **if** $needs[d]$ **then** $r[d].V$ **endif**
        enddo
end $\{user_i\}$

In chapter 6 we show how to avoid proofs by contradiction, as given above for progress properties.

4.10.2 A starvation-free solution

The next solution is starvation-free. We employ

box $r[0..N]$: *StrongSemaphore*

and everything else remains the same, except that the process-id has to be passed as an argument to $r[d].P$. The proof is along the same lines as the one for the weak semaphore case. Suppose some process j remains permanently blocked for semaphore d_j. Then semaphore d_j is never released beyond some point in the computation; if it is infinitely often released, j will acquire it according to the properties of strong semaphore. Let the permanent holder of d_j be process k, and according to the invariant k is *hungry*, $d_j < d_k$, and k is blocked for d_k. We derive an impossibility as before. A formal proof of absence of starvation is given in section 12.6.

4.10.3 A deadlock-free solution using snoopy semaphores

The next solution employs an array of snoopy semaphores. A semaphore is not released until there is demand for it. We introduce another boolean

array *holds* for each process: $holds[i]$ is *true* iff this process holds semaphore i. Variable d is set to 0 along with the transition from *eating* to *thinking*.

---

box $r[0..N]$: *SnoopySemaphore*

box $user_i$
  array$[0..N]$**(boolean)** $needs = false$;
  array$[0..N]$**(boolean)** $holds = false$;
  enum $(0..N+1)$ $d = 0$;
  enum $(thinking, hungry, eating)$ $state = thinking$;

  partial action *acquire* ::
    $hungry \ \wedge \ d \leq N \ \wedge \ (\neg needs[d] \ \vee \ holds[d]) \ \rightarrow \ d := d+1$
  | $hungry \ \wedge \ d \leq N \ \wedge \ needs[d] \ \wedge \ \neg holds[d]; \ r[d].P \ \rightarrow$
      $holds[d] := true; \ d := d+1$

  partial action *eat* ::
    $hungry \ \wedge \ d > N \ \rightarrow \ state := eating$; use resources

  $\langle [\![j : 0 \leq j \leq N$:
    partial action $release.j$ ::
      $holds[j] \ \wedge \ (\neg needs[j] \ \vee \ j \geq d); \ r[j].S \ \rightarrow$
        $r[j].V; \ holds[j] := false$
  $\rangle$
end $\{user_i\}$

We sketch a proof of absence of deadlock. Consider a system state where a group of processes are permanently *hungry*, the rest are permanently *thinking*, and none of the processes can execute any action effectively. We derive a contradiction. Let process u be permanently *hungry*, waiting for resource d_u. Then the pre-condition of the first alternative of *acquire* in the box for u is *false*; since $hungry_u$ holds permanently, $d \leq N$ holds too; hence it can be asserted that $needs_u[d_u] \ \wedge \ \neg holds_u[d_u]$ is *true*. Therefore, the pre-condition of the second alternative of *acquire* is *true*, so $r[d_u].P$ is called. Since u remains permanently *hungry*, this call to $r[d_u].P$ is rejected. So some process v holds d_u. Process v attempts the release action for d_u eventually and the action is ineffective (otherwise, d_u would be released and some process would acquire the resource, breaking the deadlock). Since $r[d_u].P$ was rejected, the call on $r[d_u].S$ accepts. Therefore, the release action for d_u is ineffective because

$$holds_v[d_u] \ \wedge \ (\neg needs_v[d_u] \ \vee \ d_u \geq d_v)$$

does not hold. Given $holds_v[d_u]$, deduce that $needs_v[d_u] \wedge (d_u < d_v)$ holds; also $needs_v[d_u] \ \Rightarrow \ hungry_v$. Thus, we have shown that for a *hungry*

process u waiting for resource d_u there is a *hungry* process v waiting for d_v, where $d_u < d_v$. Applying this argument repeatedly we find a *hungry* process w for which $d_w > N$— a contradiction, since w can then transit to *eating*.

4.11 Concluding Remarks

We have considered a large number of examples in this chapter to illustrate that the few programming constructs of Seuss are adequate. The examples come from a variety of areas. Most of our solutions are considerably more succinct than the published solutions for these problems. We expect that the solutions for problems from other application domains will be equally succinct.

4.12 Bibliographic Notes

See Andrews [11] for a comprehensive survey of language constructs and standard examples in parallel and distributed programming. The problem of the disk head scheduler of section 4.1.5 and a solution for it in Ada appear in [94]; a solution in Linda appears in Gelernter [75]. The model of faulty channel in section 4.1.6 is from [159]. Lazy caching in section 4.3 is from Afek, Brown, and Merritt [6]. There is a huge literature on discrete-event simulation (section 4.4.1); a standard reference is Law and Kelton [121], and Misra [133] is an introduction to parallel discrete-event simulation. See [4] for a study of discrete-event simulation as an action system. The process network example in section 4.5 is inspired by a solution in [32, section 8.2.3]; the problem is attributed to Hamming in Dijkstra [60, chapter 17]. The solution for barrier synchronization is due to Rajeev Joshi.

The readers-writers problem first appeared in Courtois, Heymans, and Parnas [49]. The concept of the weak semaphore (section 4.9.1) was introduced in Dijkstra [58] to solve the mutual exclusion problem [57]. The snoopy semaphore in section 4.9.3 is the author's creation. The committee coordination problem (Exercise on page 85) is studied at depth in Bagrodia [16]. The deadlock-free resource allocation algorithm in section 4.10 is standard in operating systems literature; the variation of this algorithm using snoopy semaphore is new.

Methodology for designing parallel programs has received much less attention than their sequential counterparts; see Chandy and Taylor [36] for a collection of principles.

5
Safety Properties

5.1 Introduction

The most fundamental attribute of any program is "correctness". Correctness has many interpretations, ranging from the narrow technical ones —the kind advocated in this book— to esoteric science-fiction notions that require a machine to mimic human behavior. In this book, we develop the theory for two major classes of program properties: *safety* and *progress*. We study safety properties in this chapter, and progress in the next chapter, for the programming model of chapter 2. A new class of properties, called *maximality*, is introduced in chapter 7. Logics for program composition are developed in chapters 8 and 9. In chapter 12, we extend the logic for the full programming model of chapter 3. In each chapter, we develop the theory —which is typically quite small— and apply the theory to a variety of examples to show its effectiveness.

Lamport [113] gives the following informal meaning for safety: "Something will *not* happen". Roughly, a safety property constrains the permitted actions —and therefore the permitted state changes— of a program. For instance, requiring that an integer x be nondecreasing in a program prevents any action of the program from decreasing x. Clearly, an action that causes no state change —*skip*— implements any safety property. Of particular interest are special kinds of safety properties, such as *invariant* (that a predicate remains *true* at all times during a program's execution), *stable* (that a predicate remains *true* once it becomes *true*), and *constant* (that an expression never changes value).

The primary operator for expressing safety properties is **co** (for *co*nstrains). This operator facilitates reasoning by induction on the number of computation steps. Most safety properties that arise in practice are succinctly expressed using **co**. Additionally, **co** has simple manipulation rules that permit easy deduction of new properties from the given ones.

Overview of the chapter

Operator **co** for a program is defined in terms of the actions of the program. Several special cases of **co** —*invariant*, *stable*, and *constant*— arise frequently in practice; they are described in section 5.3. The derived rules for **co** —the main ones being conjunctivity and disjunctivity— are given in section 5.4. There we describe the substitution axiom and the elimination theorem, which are essential devices for constructions of succinct proofs.

The major part of this chapter, section 5.5, illustrates applications of the theory. Safety properties for a number of programs are stated and manipulated. The examples are chosen from diverse application areas to demonstrate the usefulness of the proposed operator. Section 5.6 contains a few theoretical results. Concluding remarks are in section 5.7.

5.2 The Meaning of **co**

Consider action systems introduced in chapter 2. The results of this chapter apply to *any* action system with an arbitrary —finite or infinite— number of actions. The only restriction we impose is that there be a *skip* action in the program; execution of *skip* does not change the program state.

We write p **co** q to denote that whenever p holds before execution of any action, q holds after the execution. Formally,

$$p \ \ \mathbf{co} \ \ q \ \equiv \ \langle \forall t :: \ \{p\} \ \ t \ \ \{q\} \rangle$$

where t is an action of the system (and the quantification is over all actions).[1]

Given p **co** q, it follows that $p \Rightarrow q$, because execution of *skip* in a state that satisfies p results in a state that is same as the previous state, and this state satisfies q only if $p \Rightarrow q$. All safety properties hold in a program in which *skip* is the only action.

If t is of the form $g \ \rightarrow \ s$, then $\{p\} \ \ t \ \ \{q\}$ is established by proving $\{p \wedge g\} \ \ s \ \ \{q\}$. Ineffective executions of actions can be ignored in establishing safety properties because such executions are equivalent to *skip*; see appendix A.4.1 for details.

Given that p **co** q is a property of the program, once p holds, predicate q continues to hold up to (and including) the point where p ceases to hold (if

[1] See appendix A.4.1 for an explanation of the notation $\{p\}\ t\ \{q\}$.

p ever ceases to hold). That is, once p holds, it continues to hold until $\neg p \wedge q$ holds. In particular, p **co** p means that p holds forever once it becomes *true*.

An equivalent formulation of **co** using Dijkstra's *wp*-calculus [61] is

$$p \ \textbf{co} \ q \ \equiv \ \langle \forall t :: \ p \ \Rightarrow \ wlp.t.q \rangle .$$

This formulation allows us to establish the derived rules for **co** in section 5.3 by exploiting the properties of *wlp*.

Note on the binding powers of operators Operator **co** has lower binding power than all arithmetic and predicate calculus operators. Thus,

$p \wedge q \ \ \textbf{co} \ \ r \wedge s$ is to be interpreted as
$(p \wedge q) \ \ \textbf{co} \ \ (r \wedge s)$. □

Mathematical modeling often consists of converting imprecise or ambiguous informal descriptions to formal descriptions. Such is the case with safety properties which are informally stated using "... may remain *true as long as* ...", "... can be changed *only if* ...", or "... it is *never the case* that ...". The corresponding formal descriptions using **co** are usually easy to construct. Experience helps; therefore, this chapter includes a large number of examples and exercises of varying difficulties. Below, we show some typical informal descriptions and their formal counterparts.

Examples

In the following examples, x and y are integer-valued variables.

1. Once x is zero it remains zero until it becomes positive. Other ways of stating this fact:

   x can become nonzero only by becoming positive
   x cannot be decreased if it is zero.

   We observe that for any action if $x = 0$ is a pre-condition, then either x remains zero or x becomes positive following the action, i.e.,

   $$x = 0 \ \ \textbf{co} \ \ x \geq 0 \ .$$

2. x remains positive as long as y is. This statement is ambiguous; each of the following formal descriptions represents a reasonable interpretation.

   $$\begin{array}{ll} x > 0 \ \wedge \ y > 0 & \textbf{co} \ \ y > 0 \ \Rightarrow \ x > 0 \\ x > 0 & \textbf{co} \ \ y > 0 \ \Rightarrow \ x > 0 \\ x > 0 \ \wedge \ y > 0 & \textbf{co} \ \ x > 0 \end{array}$$

The first property says that if both x and y are positive, they will remain so if y remains positive (we cannot tell anything about x if y turns negative). The second property says that if x is positive, it will remain so until y is nonpositive. The third property says that if both x and y are positive, the first one to turn nonpositive —if it ever does so— is y.

3. x never decreases. One way to formalize this is to start with the equivalent: if x has a certain value m, it continues to have that value until it exceeds m. This is identical to example (1), except that 0 is now replaced by m. That is, for any m

$$x = m \ \ \textbf{co} \ \ x \geq m \ .$$

Here, m is a free variable; the property is universally quantified over all integers m. Therefore, this property actually represents a set of properties, one property corresponding to each value of m. Another way to express that x never decreases is

$$x \geq m \ \ \textbf{co} \ \ x \geq m \ .$$

The formal correspondence between the two properties shown here is the subject of exercise (7a). The second property is in a particularly important form that will be studied in section 5.3.

4. x may only be incremented (i.e., increased by 1). This means that in any step, either x retains its old value, say m, or acquires the new value $m + 1$. Using free variable m,

$$x = m \ \ \textbf{co} \ \ x = m \ \vee \ x = m + 1 \ .$$

5. A line in a delay-insensitive circuit is held at its current value until it has been read. Let x be the variable that denotes the line value and let y be the variable into which x's value is read. Using a free variable m

$$x = m \ \wedge \ y \neq m \ \ \textbf{co} \ \ x = m \ \vee \ y = m \ .$$

How does this differ from the following?

$$x = m \ \ \textbf{co} \ \ x = m \ \vee \ y = m$$
$$x = m \ \wedge \ y \neq m \ \ \textbf{co} \ \ x = m$$

6. A philosopher may transit from the *thinking* state only to the *hungry* state. Using predicates t and h to represent that a (particular) philosopher is in the *thinking* or *hungry* state, respectively, the above says that whenever t is falsified, h is established. However,

$$t \;\; \textbf{co} \;\; h$$

is incorrect, because t does not imply h. In such cases, we put the predicate in the left-hand side (lhs) as a disjunct in the right-hand side (rhs) to form

$$t \;\; \textbf{co} \;\; t \vee h$$

This property says that whenever t is falsified ($t \vee h$) holds; therefore, $\neg t \wedge (t \vee h)$ holds in such a state, which implies that h holds whenever t is falsified.

7. A message is received only if it has been sent. Such a statement is best treated by considering its contrapositive: Any message that is unsent remains unreceived. (Hint: apply contrapositive if you see "only", "only if" or "provided that" in a description.) There are two possible interpretations —using *sent* and *received* to denote that a specific message has been sent and received, respectively—

$$\neg sent \;\wedge\; \neg received \;\; \textbf{co} \;\; \neg received$$
$$\neg sent \;\wedge\; \neg received \;\; \textbf{co} \;\; received \;\Rightarrow\; sent$$

The second property permits *received* and *sent* to be made *true* simultaneously, i.e., in one action —a rendezvous-style communication— whereas the first property prohibits simultaneous transmission and delivery.

8. Variables x and y are never changed simultaneously. This is an important property that holds in every asynchronous system where x and y are variables that are changed by different processes. Using free variables m and n,

$$x, y = m, n \;\; \textbf{co} \;\; x = m \;\vee\; y = n$$

That is, at least one of the variables is unchanged by action execution.

9. An integer can be added to a set S of integers provided that it exceeds all elements of S. Taking the contrapositive, any integer outside the set that does not exceed all elements stays outside the set. Using free variable m that ranges over integers,

$$m \notin S \;\wedge\; m \leq S.max \;\; \textbf{co} \;\; m \notin S$$

Here, $S.max$ is the largest element of S; it is $-\infty$ if S is empty. The stated property allows several integers to be added to S in a single action. How should it be changed to say that at most one integer can be added by an action?

10. Sequence q is changed only by appending an item to it or removing its front item. This states a fact about how queues are manipulated. Let free variable Q range over possible values of q, and u be a free variable of the type that can be appended to q. We use $+\!\!+$ to append an item at the front or back of a queue.

$$q = Q \;\; \textbf{co} \;\; q = Q \;\vee\; \langle \exists u :: \; q = Q +\!\!+ u \rangle \;\vee\; \langle \exists u :: \; Q = u +\!\!+ q \rangle$$

Note that simultaneous removal and appending operations are prohibited by this property.

Relationship of **co** *to other kinds of safety operators*

Standard safety properties are simply expressed using **co**. We show some below; others appear in examples and exercises in this chapter.

In [32] we defined p **unless** q, for arbitrary predicates p and q, to mean that once p holds, it continues to hold as long as q does not; if p and q hold simultaneously, nothing can be asserted about the next state. Formally, p **unless** q is defined as follows.

$$\langle \forall t :: \; \{p \wedge \neg q\} \;\; t \;\; \{p \vee q\} \rangle$$

Then we have

$$p \textbf{ unless } q \;\equiv\; p \wedge \neg q \;\; \textbf{co} \;\; p \vee q$$

Conversely, if $p \;\Rightarrow\; q$, then

$$p \textbf{ co } q \;\equiv\; p \textbf{ unless } \neg p \wedge q$$

To say that p can be falsified only if q holds as a pre-condition, we write

$\langle \forall t :: \; \{p \wedge \neg q\} \;\; t \;\; \{p\} \rangle$, i.e.,
$p \wedge \neg q \;\; \textbf{co} \;\; p$

To express that once p holds it continues to hold until $(\neg p \wedge q)$ holds, we write

$\langle \forall t :: \; \{p\} \;\; t \;\; \{p \;\vee\; (\neg p \wedge q)\} \rangle$, i.e.,
$p \;\; \textbf{co} \;\; p \vee q$

Note We had adopted **unless** in [32] as the primary operator for expressing safety properties. Later experience (see, particularly, Staskauskas [165]) suggested that simplicity of formal manipulations is at least as important as the expressive power of an operator. Theoretically, **unless** and **co** are equally expressive, while the latter has more pleasing derived rules that allow simpler manipulations. □

5.3 Special Cases of **co**

5.3.1 Stable, invariant, constant

Several special cases of **co** appear frequently in practice, so we have special names for these cases. For predicate p and expression e,

$$\begin{array}{lcl} \textbf{stable}\ p & \equiv & p\ \ \textbf{co}\ \ p \\ \textbf{invariant}\ p & \equiv & \textbf{initially}\ p \text{ and } \textbf{stable}\ p \\ \textbf{constant}\ e & \equiv & \langle \forall k :: \ \textbf{stable}\ e = k \rangle, \\ & & \text{where } k \text{ is a free variable of the same type as } e. \end{array}$$

From the above, **stable** p means that once p is *true*, it remains *true* forever because p **co** p is

$$\langle \forall s :: \ \{p\}\ \ s\ \ \{p\} \rangle$$

which means that no action falsifies p. Predicates *false* and *true* are stable in any program. Stable predicates are ubiquitous, as in: "The system is deadlocked", or "The number of messages sent along a channel exceeds 10". We will see many instances of stable predicates in section 5.5.

A predicate is *invariant* if it holds initially and it is stable. Therefore, an invariant is always *true* during a program execution. Predicate *true* is an invariant except in the pathological case where *false* holds initially. The notion of invariant is a fundamental concept in program design. Note that we associate an invariant with a program, not with any point in the program text.

There is a subtle difference between a predicate being invariant and a predicate being always true. The difference is analogous to the distinction between "provability" and "truth" in logic. The following example is instructive.

---

program *distinction*
  integer $x, y = 0, 0$;

  $x, y := \ 0, x$
end {*distinction*}

Now, $x = 0 \ \wedge \ y = 0$ is an invariant because it holds initially and

$$\{x = 0 \ \wedge \ y = 0\}\ \ x, y := \ 0, x\ \ \{x = 0 \ \wedge \ y = 0\}$$

Since $(x = 0 \ \wedge \ y = 0) \ \Rightarrow \ (y = 0)$, we assert that $y = 0$ is always *true* during program execution. However, $y = 0$ cannot be shown to be invariant, because $y = 0$ cannot be shown to be stable according to our definition:

$$\{y = 0\}\ \ x, y := \ 0, x\ \ \{y = 0\}$$

does not hold. In section 5.4.3, we show that a predicate that is always true is also an invariant, using an extended notion of invariant that employs the substitution axiom.

An expression e is *constant* if its value never changes during a program execution. Our definition says that once e has value k, it will continue to have that value. The reader should verify that the familiar constants —*true*, 3, 'HELLO'— are constants according to our definition (see exercise 1g). Note that any expression built out of constants is a constant.

As is the case for invariants, there are expressions whose values never change but that cannot be proved constant according to our definition. For example, y cannot be shown to be constant in program *distinction*, given above, though its value is always zero. We use the substitution axiom, described in section 5.4.3, to prove that such an expression is constant.

5.3.2 Fixed point

For a given program, the fixed point predicate FP holds on "termination"; that is, for a state in which FP holds, further execution of the program will not change state, and for a state in which FP does not hold, some execution of the program causes it to change state.

A rule for computing the FP of a set of actions is given in section 2.3.2. Let action i be of the form $g_i \rightarrow C_i$ and predicate b_i hold in exactly those states where the execution of C_i has no effect. Then

$$FP \equiv \langle \forall i :: g_i \Rightarrow b_i \rangle$$

Observe that FP holds in any state where all g_is are *false*.

The rule for computing b_i is elaborated next. Assume that the action bodies contain only assignments and conditionals; there is no effective procedure to compute the FP of a loop. For an assignment statement $x := e$, the FP is $x = e$. Thus, the FP for $x := y+1$ is $x = y+1$ and for $x := x+1$ it is *false*. Similarly, for a multiple assignment statement, say, $x, y := e, f$, the FP is $x, y = e, f$.

The FP of a conditional statement **if** b **then** R **else** S **endif** , is

$$(b \Rightarrow F_R) \wedge (\neg b \Rightarrow F_S)$$

where F_R, F_S are the FPs of R and S, respectively. As an example, we compute the FP of **if** b **then** $x := x+1$ **else** $x := -x$ **endif** .

$$\begin{array}{ll} & (b \Rightarrow x = x+1) \wedge (\neg b \Rightarrow x = -x) \\ \equiv & \{\text{arithmetic}\} \\ & (b \Rightarrow \mathit{false}) \wedge (\neg b \Rightarrow x = 0) \\ \equiv & \{\text{predicate calculus}\} \\ & \neg b \wedge (\neg b \Rightarrow x = 0) \\ \equiv & \{\text{predicate calculus}\} \\ & \neg b \wedge x = 0 \end{array}$$

Sequential composition poses a problem in computing *FP*. The *FP* of $R;S$ is not $F_R \wedge F_S$. Though execution of $R;S$ starting in a state that satisfies $F_R \wedge F_S$ has no effect, there may be other states (that do not satisfy $F_R \wedge F_S$) in which the execution has no effect either. For instance, the *FP* of

$$x := x+1;\ x := x-1$$

is *true* whereas the conjunctions of the *FP*s of the two assignment statements yields *false*. We outline a procedure for computing the *FP* of $R;S$. The strategy is to transform $R;S$ to an equivalent program that has no sequential composition.

Suppose the first component of a sequential composition is a conditional. Then, the code is of the form

$$\textbf{if}\ b\ \textbf{then}\ R\ \textbf{else}\ S\ \textbf{endif}\ ;\ T \qquad \text{which is equivalent to}$$
$$\textbf{if}\ b\ \textbf{then}\ R;T\ \textbf{else}\ S;T\ \textbf{endif}$$

A similar transformation can be applied when the second component is a conditional (if both components are conditionals, then either of the two possible transformations may be applied). Repeated applications transform the program to a form where sequential composition is applied to assignment statements only; we leave it to the reader to prove this result, using induction on the program structure.

Now we show how to eliminate sequential composition applied to assignment statements. Note that

$$x := e;\ x := f \qquad \text{is equivalent to}$$
$$x := f[x := e]$$

where $f[x := e]$ is the expression obtained from f by replacing every free occurrence of x by e. For distinct variables,[2] say x and y,

$$x := e;\ y := f \qquad \text{is equivalent to}$$
$$x, y\ :=\ e, f[x := e]$$

In general, $x := e;\ Y := F$, where the second statement assigns to a list of variables, is equivalent to (1) if x is in Y, then $Y := F[x := e]$, and (2) if x is not in Y, then $x, Y\ :=\ e, F[x := e]$.

For a set of actions of the form

$$\langle [\!] \ i ::\ g_i\ \rightarrow\ C_i \rangle$$

FP is

$$\langle \forall i ::\ g_i \Rightarrow b_i \rangle$$

where b_i is the *FP* of C_i. Thus, for

---

[2] It is not always easy to decide if two variables are distinct, for instance when the variables are the elements $A[i]$, $A[j]$ of an array A.

$\langle [\!] \ i : \ 0 \leq i < N : \ m := max(m, A[i]) \rangle$

FP is calculated as follows.

$$
\begin{array}{ll}
 & FP \\
\equiv & \{\text{the definition of } FP\} \\
 & \langle \forall i : \ 0 \leq i < N : \ m = max(m, A[i]) \rangle \\
\equiv & \{\text{arithmetic}\} \\
 & \langle \forall i : \ 0 \leq i < N : \ m \geq A[i] \rangle \\
\equiv & \{\text{arithmetic}\} \\
 & m \geq \langle max \ i : \ 0 \leq i < N : \ A[i] \rangle
\end{array}
$$

The notion of program termination can be couched in terms of FP: a program is guaranteed to terminate iff it eventually reaches a state that satisfies FP; this is a progress property that is discussed in chapter 6. The well-known phrase "The program is deadlock-free" means that $\neg FP$ is always *true*; then, it is always possible to change the program state.

We give the following formal characterization of FP in section 5.6.3: it is the weakest predicate p such that $p \ \wedge \ b$ is stable for any b. We show that there is a unique weakest predicate of this form. The following result is a direct consequence of this characterization.

- (Stability at fixed point) For any predicate b, **stable** $(FP \wedge b)$.

5.4 Derived Rules

5.4.1 Basic rules

All these rules follow directly from facts about the predicate transformer *wlp*; see appendix A.4.2. Here, p, q, p', q', and r are arbitrary predicates.

- false **co** p
- p **co** true
- (conjunction, disjunction)

$$\frac{p \ \textbf{co} \ q \ , \ p' \ \textbf{co} \ q'}{\begin{array}{l} p \wedge p' \ \textbf{co} \ q \wedge q' \\ p \vee p' \ \textbf{co} \ q \vee q' \end{array}}$$

The conjunction and disjunction rules follow from the junctivity and monotonicity properties of *wlp* [61] and of logical implication. These rules generalize in the obvious manner to any set —finite or infinite— of **co**-properties, because *wlp* and logical implication are universally conjunctive and universally disjunctive. As corollaries of conjunction and disjunction —taking the conjunction of r **co** true and p **co** q, and the disjunction of false **co** r and p **co** q, respectively— we obtain

- (lhs strengthening) $\dfrac{p \ \mathbf{co} \ q}{p \wedge r \ \mathbf{co} \ q}$
- (rhs weakening) $\dfrac{p \ \mathbf{co} \ q}{p \ \mathbf{co} \ q \vee r}$

Also, **co** is transitive:

- (transitivity) $\dfrac{p \ \mathbf{co} \ q \ , \ q \ \mathbf{co} \ r}{p \ \mathbf{co} \ r}$

This is because $q \Rightarrow r$ from q **co** r, so the rhs of p **co** q can be weakened to p **co** r. However, transitivity seems to be of little value because any application of transitivity can be replaced by lhs strengthening —strengthen q to p in q **co** r— or rhs weakening —weaken q to r in p **co** q.

Operator **co** is a form of temporal implication. It shares many of the properties of logical implication, such as the ones shown above. However, it is not reflexive (i.e., p **co** p does not always hold) nor are we allowed to deduce a contrapositive ($\neg q$ **co** $\neg p$ cannot be deduced from p **co** q).

Since the lhs of a **co**-property can be strengthened and its rhs can be weakened, we write a proof of p **co** s, say, in the following format; see appendix A.3 for an explanation of this proof format.

$$
\begin{array}{ll}
 & p \\
\Rightarrow & \{\text{why } p \Rightarrow q\} \\
 & q \\
\mathbf{co} & \{\text{why } q \ \mathbf{co} \ r\} \\
 & r \\
\Rightarrow & \{\text{why } r \Rightarrow s\} \\
 & s
\end{array}
$$

5.4.2 Rules for the special cases

The following rules follow from the conjunction and disjunction rules given above.

- (stable conjunction, stable disjunction)
$$\frac{p \ \mathbf{co} \ q \ , \ \mathbf{stable} \ r}{\begin{array}{l} p \wedge r \ \mathbf{co} \ q \wedge r \\ p \vee r \ \mathbf{co} \ q \vee r \end{array}}$$
A special case of the above is
- $$\frac{\mathbf{stable} \ p \ , \ \mathbf{stable} \ q}{\begin{array}{l} \mathbf{stable} \ p \wedge q \\ \mathbf{stable} \ p \vee q \end{array}}$$
- $$\frac{\mathbf{invariant} \ p \ , \ \mathbf{invariant} \ q}{\mathbf{invariant} \ p \wedge q}$$
- (constant formation) Any expression built out of constants and free variables is a constant.

5.4.3 Substitution axiom

An invariant may be replaced by true *and vice versa in any property of a program.*

The substitution axiom allows us to deduce properties that we cannot deduce directly from the definition. For example, given p **co** q and **invariant** J, we conclude that

$$p \wedge J \ \textbf{co} \ q \ , \ p \wedge J \ \textbf{co} \ q \wedge J \ , \ p \vee \neg J \ \textbf{co} \ q \wedge J \ .$$

In particular, given that **invariant** p and $p \Rightarrow q$, we show **invariant** q, as follows.

invariant p	, given
invariant $p \wedge q$	, $p \ \equiv \ p \wedge q$, since $p \Rightarrow q$
invariant q	, replace p by *true* using the substitution axiom

For program *distinction* of section 5.3.1, we showed that $y = 0$ is always *true* though we could not show it to be invariant. However, we could show that $x = 0 \ \wedge \ y = 0$ is invariant. Using the argument given above, since $(x = 0 \ \wedge \ y = 0) \ \Rightarrow \ (y = 0)$, we now claim that $y = 0$ is invariant. Thus, the substitution axiom allows us to remove the distinction between always *true* and invariant.

Another consequence of the substitution axiom is that a theorem and an invariant have the same status; an invariant can be treated as a theorem, and a theorem, of course, is an invariant. Therefore, we often write simply "J", rather than "**invariant** J".

Note For compositions of programs, a generalization of the substitution axiom is given in section 8.4. □

Rationale for the substitution axiom

In the definition of p **co** q, we interpreted $\{p\}\ t\ \{q\}$ to mean that if action t is started in any state that satisfies p, q holds in the resulting state on completion of t. What is "any state"? We restrict the state space to the *reachable* states: a *reachable state* is a state (i.e., an assignment of values to variables) that may arise during a computation; i.e., either it satisfies the initial condition or it is a state that may result by applying an action to a reachable state. It follows that every reachable state satisfies every invariant. In fact, the set of reachable states is the set of states that satisfy the strongest invariant (we show the existence of the strongest invariant in section 5.6.2).

For action t in a program, we take $\{p\}\ t\ \{q\}$ to mean that if t is started in any reachable state that satisfies p, the resulting (reachable) state satisfies q. We prove $\{p\}\ t\ \{q\}$ by showing, for some invariant J,

$$p \wedge J \ \Rightarrow \ wlp.t.q \ .$$

This is because the set of states that satisfy $p \wedge J$ *includes* all reachable states that satisfy p (since a reachable state satisfies any invariant). In particular, we can use the types of variables, or even *true*, as the invariant J. Even though the notion of invariant is defined using **co**, and the definition of **co** seems to require the notion of invariant, there is no circularity if we start with a known invariant, such as *true*. We can then deduce other invariants that can be applied in deducing further **co**-properties and invariants.

5.4.4 Elimination theorem

Free variables are essential to our theory. Free variables can be introduced in the lhs by strengthening and in the rhs by weakening, e.g., from p **co** q we deduce for program variable x and free variable m

$$p \ \wedge \ x = m \ \ \textbf{co} \ \ q$$
$$p \ \ \textbf{co} \ \ q \ \vee \ x \neq m$$

Free variables can be eliminated by taking conjunctions or disjunctions. We give a useful theorem below for eliminations of free variables by employing disjunction.

Let p be a predicate, x a program variable, and m a free variable (of the same type as x). Let $p[x := \ m]$ be the predicate obtained by replacing all free occurrences of x by m in p. If p names[3] no program variable other than x, then $p[x := \ m]$ has no program variable, so it is a constant. In particular, $p[x := \ m]$ is then stable. Observe that

$$p \ \equiv \ \langle \exists m :: \ p[x := \ m] \ \wedge \ x = m \rangle$$

Elimination Theorem

$$\frac{\begin{array}{c} x = m \ \ \textbf{co} \ \ q \ , \text{ where } m \text{ is free} \\ p \text{ names neither } m \text{ nor any program variable other than } x \end{array}}{p \ \ \textbf{co} \ \ \langle \exists m :: \ p[x := \ m] \ \wedge \ q \rangle}$$

Proof:

$x = m \ \ \textbf{co} \ \ q$, premise
$p[x := \ m] \ \wedge \ x = m \ \ \textbf{co} \ \ p[x := \ m] \ \wedge \ q$
, stable conjunction with $p[x := \ m]$
$\langle \exists m :: \ p[x := \ m] \ \wedge \ x = m \rangle \ \ \textbf{co} \ \ \langle \exists m :: \ p[x := \ m] \ \wedge \ q \rangle$
, disjunction over all m
$p \ \equiv \ \langle \exists m :: \ p[x := \ m] \ \wedge \ x = m \rangle$
, see discussion above
$p \ \ \textbf{co} \ \ \langle \exists m :: \ p[x := \ m] \ \wedge \ q \rangle$, from above two □

[3] "p names no program variable other than x" means that no program variable other than x occurs free in p.

Note The consequent of the elimination theorem can be written as

$$p \ \textbf{co} \ \langle \exists m : \ p[x := \ m] : \ q \rangle. \qquad \square$$

The elimination theorem can be applied where x is a list of variables and m is a list of free variables. In the following example, we apply the theorem with $x = (u, v)$.

Example Given

$$u, v = m, n \ \textbf{co} \ \ u, v = m, n \ \vee \ (m > n \ \wedge \ u, v = m - 1, n)$$

we show that **stable** $u \geq v$. Using $p \ \equiv \ u \geq v$ in the elimination theorem we have

$$\begin{array}{ll} & u \geq v \\ \textbf{co} & \{\text{elimination theorem}\} \\ & \langle \exists m, n :: \ (m \geq n) \\ & \wedge \ (\langle u, v = m, n \rangle \ \vee \ \langle m > n \ \wedge \ u, v = m - 1, n \rangle) \rangle \\ \Rightarrow & \{\text{weaken rhs}\} \\ & \langle \exists m, n :: \ u \geq v \ \vee \ u \geq v \rangle \\ \Rightarrow & \{\text{simplify}\} \\ & u \geq v \end{array} \qquad \square$$

5.4.5 Distinction between properties and predicates

Given that p and q are predicates, p **co** q is a *property*. Though it is tempting to view a property also as a predicate, we do not do so. Therefore, it is syntactically incorrect to write, for instance,

$$p \ \vee \ (q \ \ \textbf{co} \ \ r)$$

for predicates p, q, and r. However, we use the logical symbols $\wedge$, $\Rightarrow$, $\equiv$ in combining properties with the following interpretation.

For properties α and β, $\alpha \ \wedge \ \beta$ holds in a program provided that both α and β are properties of the program. Property $\alpha \ \Rightarrow \ \beta$ holds in a program P if by taking α as an additional premise β can be inferred as a property of P. Equivalently, if Π is the set of *all* properties of P, then β can be deduced from $\Pi \ \cup \ \{\alpha\}$. And $\alpha \ \equiv \ \beta$ holds whenever $\alpha \ \Rightarrow \ \beta$ and $\beta \ \Rightarrow \ \alpha$.

As a small example, consider a program over integer variables x and y whose actions are shown below. (There is no fairness in this program execution.)

$$\begin{array}{ll} & x, y := \ x + 1, y + 1 \\ [\!] & x, y := \ x - 1, y - 1 \end{array}$$

It is possible to show for this program that x ascending $\Rightarrow$ y ascending, or, more formally,

$$\langle \forall\ m ::\ \textbf{stable}\ x \geq m \rangle\ \Rightarrow\ \langle \forall\ n ::\ \textbf{stable}\ y \geq n \rangle$$

Similarly,

$$\langle \forall\ m ::\ \textbf{stable}\ x \leq m \rangle\ \Rightarrow\ \langle \forall\ n ::\ \textbf{stable}\ y \leq n \rangle$$

We sketch a proof of the latter property. For arbitrary integers n and r

stable $x - y = r$	, from the program text
stable $x \leq n + r$	, from $\langle \forall\ m ::\ \textbf{stable}\ x \leq m \rangle$
stable $(x - y = r\ \wedge\ y \leq n)$	, conjunction of above two, simplify
stable $y \leq n$	, disjunction of above over all r $\square$

5.5 Applications

We apply our theory to several small problems. In each case, we convert an informal description to a set of **co**-properties, apply some of the manipulation rules given in section 5.4, and interpret the derived results.

5.5.1 Non-operational descriptions of algorithms

It is often preferable to describe an algorithm not by program text but by its properties; the properties constitute the specification of the algorithm. There are several advantages to this approach: (1) we can express a family of algorithms by one set of properties, because many implementation details can be ignored while writing the properties; (2) it is usually easier to prove facts about an algorithm from its properties than from its code; and (3) it is often easier to understand an algorithm from its properties than from its code. We choose a very simple problem from sequential programming —computing the maximum of a set of numbers— to illustrate these aspects. Section 5.5.2 contains another small exercise of this nature and in section 5.5.3 a small concurrent programming example is treated using this approach.

The typical algorithm to compute the maximum value v of a nonempty finite set S of integers is to (1) initially assign a small value, $-\infty$, say, to v and (2) then scan the elements of S in some order, updating v whenever the scanned element has a larger value. Instead of expressing this algorithm in the notation of a programming language, we describe it by its properties, by focusing on the allowable changes to v. Using a free variable m that ranges over integers and $-\infty$,

$$\textbf{initially}\ v = -\infty$$
$$v = m\ \ \textbf{co}\ \ v = m\ \vee\ (v \in S\ \wedge\ v > m)$$

The initial condition is as described above. The **co**-property says that v is changed only to a higher value that is also in S. This description ignores

the order in which the elements of S are scanned, leaving open a number of possibilities for implementation (one of which we discuss below). Now we can deduce several properties.

1. v is nondecreasing; i.e., for any n

   stable $v \geq n$

2. v never exceeds the maximum of S, i.e.,

   invariant $v \leq M$, where $M = \langle max\ x :\ x \in S :\ x \rangle$

3. Once v is in S, it remains in S, i.e., **stable** $v \in S$.

The proofs of all these properties appeal to the elimination theorem; we show one below.

• Proof of **invariant** $v \leq M$:
initially $v = -\infty$; hence, **initially** $v \leq M$. We show **stable** $v \leq M$.

$v \leq M$
co {elimination theorem: $v = m$ **co** $v = m \ \vee\ (v \in S \ \wedge\ v > m)$}
$\langle \exists\ m ::\ (m \leq M \ \wedge\ v = m) \ \vee\ (m \leq M \ \wedge\ v \in S \ \wedge\ v > m) \rangle$
$\Rightarrow$ {weaken each disjunct}
$\langle \exists m ::\ v \leq M \ \vee\ v \in S \rangle$
$\Rightarrow$ {simplify}
$v \leq M \ \vee\ v \in S$
$\Rightarrow$ {$v \in S \ \Rightarrow\ v \leq M$, from the definition of M}
$v \leq M$ □

We cannot yet prove that v will eventually equal M; we have to wait until we develop the theory of progress in chapter 6.

A refinement of this algorithm is the following. Set S is represented by an array A, and the elements of A are scanned in the order $A[0], A[1] \ldots$. We again describe this algorithm by its properties, i.e., by the way its variables are manipulated. Let $A[0..s-1]$ be the segment of the array that has already been scanned. Then

initially $v, s = -\infty, 0$
$v, s = m, k$ **co** $v, s = m, k \ \vee\ v, s = max(m, A[k]), k+1$

where m ranges over integers and $-\infty$, and k ranges over the indices of A. We prove that v is the maximum over the scanned segment, i.e.,

invariant $v = \langle max\ i :\ 0 \leq i < s :\ A[i] \rangle$

Initially, this property holds since

$-\infty = \langle max\ i :\ 0 \leq i < 0 :\ A[i] \rangle$

Next we prove

• Proof of **stable** $v = \langle max\ i :\ 0 \leq i < s :\ A[i] \rangle$:

$$
\begin{array}{ll}
 & v = \langle max\ i :\ 0 \leq i < s :\ A[i] \rangle \\
\textbf{co} & \{\text{elimination theorem}\} \\
 & \langle \exists m, k :: \\
 & \quad (m = \langle max\ i :\ 0 \leq i < k :\ A[i] \rangle\ \wedge\ v, s = m, k) \vee \\
 & \quad (m = \langle max\ i :\ 0 \leq i < k :\ A[i] \rangle\ \wedge\ v, s = max(m, A[k]), k+1) \\
 & \rangle \\
\Rightarrow & \{\text{arithmetic}\} \\
 & v = \langle max\ i :\ 0 \leq i < s :\ A[i] \rangle
\end{array}
$$

□

5.5.2 Common meeting time

This problem is discussed in Chandy and Misra [32, section 1.4]; a more general version was discussed in section 1.2. The purpose of this example, much like the one in section 5.5.1, is to explore a family of design alternatives by considering the safety properties common to all members of the family.

It is required to find the earliest meeting time acceptable to every member in a group. Time is non-negative and real-valued. To simplify notation, assume that there are only two persons, F and G, in the group. Associated with F and G are functions, f and g, respectively, where each function maps non-negative reals to non-negative reals (i.e., times to times). For any real t, $f(t)$ is the earliest time at or after t when F can meet; $g(t)$ is similarly defined. Time t is acceptable to F iff $f(t) = t$. Time t is a *common meeting time* iff it is acceptable to both F and G, i.e., $f(t) = t\ \wedge\ g(t) = t$. The goal is to design an algorithm that computes the earliest (i.e., smallest non-negative) common meeting time, provided that one exists.

Several algorithms and their implementations on various architectures are described in [32]. Here, we define the essential safety properties that are common to *all* these algorithms.

First, we have to make certain assumptions about f and g so that the earliest meeting time can be computed effectively. Our verbal description suggests that $t \leq f(t)$, but we won't require this property for the following derivation; it is needed for the implementation (see the discussion at the end of this section). We postulate that f and g be monotonic, i.e., for all non-negative real m and n:

$$
\begin{array}{ll}
m \leq n\ \Rightarrow\ f(m) \leq f(n) & \text{(CMT1)} \\
m \leq n\ \Rightarrow\ g(m) \leq g(n)\ . &
\end{array}
$$

We adopt the following strategy in computing the earliest meeting time: define a variable t, non-negative and real, whose value never exceeds the earliest common meeting time, and that is increased eventually if it is not a

common meeting time. The progress property is discussed in section 6.5.2. Here, we consider the safety aspect of the problem, given by the first requirement on t. A strategy for implementing this requirement is to initially set t to 0, and modify t by: if t's value is m before an action, it does not exceed both $f(m)$ and $g(m)$ after the action. It is not obvious that this strategy prevents t from exceeding the earliest common meeting time; we prove this fact below.

The formal description of the strategy is as follows.

$$\textbf{initially } t = 0 \tag{CMT2}$$
$$t = m \ \textbf{co} \ t \leq max(f(m), g(m)) \tag{CMT3}$$

Let $com(n)$ denote that n is a common meeting time, i.e.,

$$com(n) \ \equiv \ \langle f(n) = n \ \wedge \ g(n) = n \rangle$$

We prove from (CMT1–CMT3) that t exceeds no common meeting time; i.e., for any n,

$$com(n) \ \Rightarrow \ t \leq n \tag{CMT4}$$

The initial condition for the invariance of (CMT4) follows from (CMT2). The remaining proof obligation is, rewriting (CMT4),

$$\textbf{stable } (\neg com(n) \ \vee \ t \leq n) \ . \tag{CMT5}$$

We prove (CMT5) next. Its proof requires a subproof that is given following the main proof.

• Proof of **stable** $(\neg com(n) \ \vee \ t \leq n)$:

$$\begin{array}{ll}
 & \neg com(n) \ \vee \ t \leq n \\
\textbf{co} & \{\text{elimination theorem on (CMT3),} \\
 & \quad \text{where } p \text{ is } \neg com(n) \ \vee \ t \leq n\} \\
 & \langle \exists \ m :: \ (\neg \ com(n) \ \vee \ m \leq n) \ \wedge \ t \leq max(f(m), g(m)) \rangle \\
\equiv & \{\text{rewrite the first term as } \neg \ com(n) \ \vee \ (com(n) \wedge m \leq n)\} \\
 & \langle \exists \ m :: \ (\neg \ com(n) \ \vee \ (com(n) \wedge m \leq n)) \\
 & \wedge \ t \leq max(f(m), g(m)) \rangle \\
\equiv & \{\text{rewrite in disjunctive form}\} \\
 & \langle \exists \ m :: \ (\neg \ com(n) \ \wedge \ t \leq max(f(m), g(m))) \\
 & \vee \ (com(n) \ \wedge \ m \leq n \ \wedge \ t \leq max(f(m), g(m))) \rangle \\
\Rightarrow & \{\text{the first disjunct implies } \neg com(n), \\
 & \quad \text{the second disjunct implies (see below) } t \leq n\} \\
 & \langle \exists \ m :: \ \neg \ com(n) \ \vee \ t \leq n \rangle \\
\Rightarrow & \{\text{predicate calculus}\} \\
 & \neg \ com(n) \ \vee \ t \leq n
\end{array}$$

□

This establishes (CMT5). Now we prove the result claimed in the above proof.

• Proof of $\langle com(n) \ \wedge \ m \leq n \ \wedge \ t \leq max(f(m), g(m)) \rangle \ \Rightarrow \ \langle t \leq n \rangle$:

$$
\begin{array}{ll}
 & t \\
\leq & \{\text{from the antecedent, } t \leq max(f(m), g(m))\} \\
 & max(f(m), g(m)) \\
\leq & \{m \leq n \text{ from the antecedent,} \\
 & \quad (m \leq n) \ \Rightarrow \ \langle f(m) \leq f(n) \ \wedge \ g(m) \leq g(n) \rangle \text{ from (CMT1),} \\
 & \quad max \text{ is monotonic in its arguments}\} \\
 & max(f(n), g(n)) \\
= & \{\text{from } com(n) \text{ in the antecedent, } f(n) = n \text{ and } g(n) = n\} \\
 & max(n, n) \\
= & \{\text{arithmetic}\} \\
 & n
\end{array}
$$

□

The strategy given by (CMT3) is quite general. It subsumes the following strategies.

$$
\begin{array}{l}
t = m \ \ \textbf{co} \ \ t = m \\
t = m \ \ \textbf{co} \ \ t = m \ \vee \ t = max(f(m), g(m)) \\
t = m \ \ \textbf{co} \ \ t = m \ \vee \ t = f(m) \ \vee \ t = g(m)
\end{array}
$$

In each of these properties, the rhs is stronger than that of (CMT3); hence, (CMT3) —and, consequently, (CMT5)— can be derived from each of them. A weak safety property like (CMT3) is preferable for initial design explorations because it constrains the allowable actions only minimally. Each of the following programs, P0–P2, (in which the initial condition, $t = 0$, is not shown) implements (CMT3). Also, (P3) implements (CMT3) provided that f and g map naturals to naturals.

$$
\begin{array}{lll}
\text{P0} & :: & t := \ t \qquad \{\text{does nothing useful}\} \\
\text{P1} & :: & t := \ f(t) \ [\!] \ t := \ g(t) \\
\text{P2} & :: & t := max(f(t), g(t)) \\
\text{P3} & :: & t < max(f(t), g(t)) \ \rightarrow \ t := \ t + 1
\end{array}
$$

A useful strengthening of (CMT3) is to require that t be nondecreasing; i.e.,

$$t = m \ \ \textbf{co} \ \ m \leq t \leq max(f(m), g(m)). \qquad \text{(CMT3}')$$

This property is easily implemented (by programs P1 and P2, for instance) if we know that functions f and g are ascending:

$$n \leq f(n) \ \wedge \ n \leq g(n). \qquad \text{(CMT0)}$$

The reader can prove (by applying elimination theorem to CMT3′) that

$$\textbf{stable} \ \langle t = n \ \wedge \ com(n) \rangle$$

i.e., t does not change once its value equals a common meeting time. Interestingly, neither (CMT0) nor (CMT3′) is required in deriving (CMT4). Another fact about f (and g) suggested by the verbal description is that $f(f(n)) = f(n)$, which has not been used in our derivations.

5.5.3 A small concurrent program: token ring

The method advocated in sections 5.5.1 and 5.5.2 —describing an algorithm by its properties— is most profitably applied to concurrent algorithms. Here, we consider a message transmission problem over a ring network.

A set of processes are connected in a ring where the message transmissions take place over the edges of the ring; thus, a process may communicate only with its left or right neighbor. At most one process may transmit at any time. A technique to meet this requirement is to have a single token circulate in the ring and allow only the token holder to transmit. The following is an abstraction of this solution.

A process is in one of three states: waiting to transmit (also called *hungry*), transmitting (also called *eating*), or neither of the above (also called *thinking*). The terms *hungry*, *eating*, *thinking* are taken from the dining philosophers problem, which is a standard abstraction in resource allocation. An *eating* process can transit only to *thinking* (see TR1, below); a *thinking* process can transit only to *hungry* (see TR2); a *hungry* process can transit only to *eating* (TR3); a *hungry* process remains *hungry* as long as it does not hold the token (TR4); and the process holding the token relinquishes it only if it becomes non-*eating* (TR5). We would like to prove that there is at most one *eating* process —i.e., at most one transmitting process— in the system. An intuitive proof is obvious: there is at most one token in the system, and an *eating* process holds the token. Unfortunately, this proof leaves out a number of details. For instance, the rule for relinquishing the token, as stated above, is vague; it is not clear if a *hungry* process can relinquish a token. Further, in the initial state, the *eating* process, if any, has to hold the token (TR0). Such minor details, which are often ignored in informal descriptions, cause major problems.

Notation We write t_i to denote that process i is *thinking*; similarly h_i and e_i stand for *hungry* and *eating*. These predicates are mutually exclusive, and $h_i \vee t_i \vee e_i$ holds. The position of the token is in variable p, i.e., $p = i$ (as in TR5) denotes that process i holds the token. In the following, i ranges over all processes. □

$$\textbf{initially } e_i \;\Rightarrow\; p = i \tag{TR0}$$
$$e_i \;\; \textbf{co} \;\; e_i \vee t_i \tag{TR1}$$
$$t_i \;\; \textbf{co} \;\; t_i \vee h_i \tag{TR2}$$
$$h_i \;\; \textbf{co} \;\; h_i \vee e_i \tag{TR3}$$
$$h_i \;\wedge\; p \neq i \;\; \textbf{co} \;\; h_i \tag{TR4}$$
$$p = i \;\; \textbf{co} \;\; p = i \;\vee\; \neg e_i \tag{TR5}$$

Properties (TR0–TR5) specify only the safety aspects of the system. In particular, the protocol for transmitting the token (which is sent to the neighbor in a specific direction on the ring) is completely ignored in this specification. The complete protocol is specified in section 6.5.3. Here,

we show that the partial specification is sufficient for establishing mutual exclusion, i.e., that there is at most one *eating* process.

The assertion that there is at most one *eating* process can be expressed by the invariant

$$\langle \forall i, j :: \ e_i \ \wedge \ e_j \ \ \Rightarrow \ \ i = j \rangle$$

This result is shown by proving that an *eating* process holds the token; i.e., for any i, **invariant** $e_i \ \Rightarrow \ p = i$. Then

$e_i \ \wedge \ e_j$	, given
$p = i \ \wedge \ p = j$	, from $e_i \ \Rightarrow \ p = i$ and $e_j \ \Rightarrow \ p = j$
$i = j$	, predicate calculus

Next, we prove **invariant** $e_i \ \Rightarrow \ p = i$, for any i. The result follows for the initial state from (TR0). The remaining proof obligation is as follows:

• Proof of **stable** $(e_i \ \Rightarrow \ p = i)$, for any i:

$t_i \ \vee \ h_i \ \vee \ p = i$ **co** $t_i \ \vee \ h_i \ \vee \ p = i \ \vee \ \neg e_i$	
	, disjunction of (TR2, TR4, TR5)
stable $e_i \ \Rightarrow \ p = i$	, replace $t_i \vee h_i$ by $\neg e_i$ and rewrite □

Observe that (TR1, TR3) are unnecessary for the proof of mutual exclusion. This proof is concise, partly because it does not mimic the usual common-sense reasoning.

5.5.4 From program texts to properties

It is sometimes useful to specify how a variable, or a group of variables, is changed in a program. Specifying changes for all variables amounts to describing all safety properties of a program. We show how such a property can be derived from the program text.

Consider an integer variable x that is changed only by the following two actions in a program.

$$x := x + 1 \ \ [\!] \ \ x := x - 1$$

We then assert, where m is a free variable,

$$x = m \ \ \textbf{co} \ \ (x = m) \ \vee \ (x = m + 1) \ \vee \ (x = m - 1)$$

This observation can be generalized as follows. For action s and predicate q, we write *sp.s.q* to denote the *strongest post-condition* resulting from execution of s in a state that satisfies q. The strongest post-condition can be computed by the formula given below, when the action is given by an assignment statement and the pre-condition specifies the exact values of all variables. Let X be the set of all variables accessed (read or written) in a program and M be a list of free variables.

$$sp.(X := E).(X = M) \;\equiv\; (X = E[X := M])$$

where $E[X := M]$ is obtained from E by replacing all free occurrences of X by M. If P is a conditional statement, **if** b **then** R **else** S **endif**, then

$$sp.P.(X = M) \;\equiv\; \langle b[x := M] \wedge sp.R.(X = M)\rangle \;\vee\; \langle \neg b[x := M] \wedge sp.S.(X = M)\rangle$$

As shown in section 5.3.2, a conditional statement can be transformed in such a way that R and S are assignment statements.

Let A_i denote action i of the program. Then the program has the following safety property.

$$X = M \;\; \textbf{co} \;\; \langle x = M\rangle \;\vee\; \langle \exists i :: \; sp.A_i.(X = M)\rangle$$

As an example, consider a program that has two integer variables x and y and the following actions.

$$\begin{array}{ll} & x := x + 1 \\ [\!] & \textbf{if} \;\; x > y \;\; \textbf{then} \;\; y := y + 1 \;\; \textbf{endif} \end{array}$$

The first assignment is equivalent to $x, y := x + 1, y$ and the second to $x, y := x, y + 1$. The derived property is

$$\begin{array}{lll} x, y = m, n & \textbf{co} & (x, y = m, n) \\ & & \vee\; (x, y = m + 1, n) \\ & & \vee\; (m > n \;\wedge\; x, y = m, n + 1) \\ & & \vee\; (m \leq n \;\wedge\; x, y = m, n) \end{array}$$

A property that is derived in this manner encodes *all* safety properties of a program. If only the safety properties are of interest, the program text can be discarded in favor of the single derived property. Further, the derived property is in a form to which the elimination theorem can be applied, to deduce how any group of variables is modified in the program. For the example above, it follows from the derived property that

$$\begin{array}{lll} x, y = m, n \;\; \textbf{co} \;\; x = m \;\vee\; x = m + 1 & & \text{, weaken rhs} \\ x = m \;\; \textbf{co} \;\; x = m \;\vee\; x = m + 1 & & \text{, disjunction over all } n \end{array}$$

That is, x can be incremented only by the actions of the program. Similarly, we may deduce the following property, by weakening the rhs of the derived property, to describe the possible changes in y.

$$x, y = m, n \;\; \textbf{co} \;\; (y = n) \;\vee\; (m > n \;\wedge\; y = n + 1)$$

To prove a safety property for this action system —say, **stable** $x \geq y$— we can start either with the program text (and show that each action preserves the property) or with the safety property of the system derived as above and then deduce the result. Whenever we have a mixture of program

fragments and properties, the latter strategy is preferable since we can then work using a single notation.

Next, we consider an example from Habermann [82]. The central theorem of that paper follows quite simply by converting the program text to a safety property and applying the elimination theorem. (The theorem can be derived equally easily by showing that a certain predicate is invariant.) The following is a rewriting of a program from [82].

---

box *Habermann*
  nat $np, nw = 0, 0$;
  nat $c = C$ {C is some specific initial value};

  wait :: **if** $nw < c$ **then** $np := np + 1$ **endif** ; $nw := nw + 1$
[] *signal* :: **if** $nw > c$ **then** $np := np + 1$ **endif** ; $c := c + 1$

end {*Habermann*}

Here, *wait* encodes a P operation on a semaphore and *signal* is a V. Variable np is the number of successful calls on *wait*, and nw is the total number of calls on *wait* (where each unsuccessful caller is queued). Value of c is C plus the total number of calls to *signal*. The main result proved in [82] is that $np = min(nw, c)$.

Rewrite *wait* as

if $nw < c$
  then $np, nw, c := np + 1, nw + 1, c$
  else $np, nw, c := np, nw + 1, c$
endif

With the pre-condition $np, nw, c = m, n, r$, the post-condition of *wait* is

$$(n < r \ \wedge \ np, nw, c = m + 1, n + 1, r) \ \vee$$
$$(n \geq r \ \wedge \ np, nw, c = m, n + 1, r)$$

Similarly, the corresponding post-condition for *signal* is

$$(n > r \ \wedge \ np, nw, c = m + 1, n, r + 1) \ \vee$$
$$(n \leq r \ \wedge \ np, nw, c = m, n, r + 1)$$

The derived safety property of the program is

$$\begin{array}{lll} np, nw, c = m, n, r & \textbf{co} & (np, nw, c = m, n, r) \\ & & \vee \ (n < r \ \wedge \ np, nw, c = m + 1, n + 1, r) \\ & & \vee \ (n \geq r \ \wedge \ np, nw, c = m, n + 1, r) \\ & & \vee \ (n > r \ \wedge \ np, nw, c = m + 1, n, r + 1) \\ & & \vee \ (n \leq r \ \wedge \ np, nw, c = m, n, r + 1) \end{array}$$

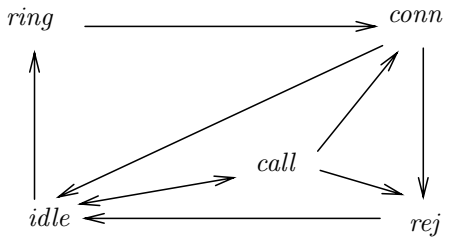


Figure 5.1: A state transition diagram for a telephone

From the initial condition, $np = min(nw, c)$ holds initially, regardless of the value of C. Applying the elimination theorem, it can be shown that $np = min(nw, c)$ is stable. Therefore $np = min(nw, c)$ is invariant.

5.5.5 Finite state systems

Finite state descriptions are often used for specifications of communication protocols and discrete control systems. Here, we show how the state transitions can be described in our theory. As an example of a finite state system, we consider an extremely simplified version of a telephone system; see Staskauskas [165] for a more realistic version.

We focus attention on a single telephone. We postulate it to have the following states:

idle: the handset is on-hook and not ringing

ring: the handset is on-hook and ringing

call: the handset is off-hook and a number is being dialed

conn: the telephone is engaged in a call

rej: the telephone is receiving a busy signal or the other party has disconnected

Our model is so simple that we cannot even distinguish in state *conn* if this telephone initiated or received a call.

A possible state transition diagram is shown in Fig. 5.1. Most of the transitions are self-explanatory; for instance, the transition from *conn* to *idle* is taken when the user hangs up, and the transition from *call* to *rej* is effected by the telephone switch giving a busy signal to the caller.

The safety properties of a finite state system can be obtained automatically as follows. There is one safety property for each state. For a state x that has possible transitions to states y and z, the corresponding property is

$$\text{state} = x \;\; \textbf{co} \;\; \text{state} = x \;\vee\; \text{state} = y \;\vee\; \text{state} = z$$

which expresses the fact that from the current state, x, an action can effect a state change only to y or z. The safety properties obtained from the diagram in Fig. 5.1 are as follows (in these properties we write, for instance, *idle* to denote that the state is *idle*).

$$\begin{array}{lll} \mathit{idle} & \textbf{co} & \mathit{idle} \vee \mathit{ring} \vee \mathit{call} \\ \mathit{call} & \textbf{co} & \mathit{call} \vee \mathit{idle} \vee \mathit{conn} \vee \mathit{rej} \\ \mathit{conn} & \textbf{co} & \mathit{conn} \vee \mathit{idle} \vee \mathit{rej} \\ \mathit{rej} & \textbf{co} & \mathit{rej} \vee \mathit{idle} \\ \mathit{ring} & \textbf{co} & \mathit{ring} \vee \mathit{conn} \end{array}$$

The transitions in Fig. 5.1 are of two types —those made by the user and those made by the telephone switch. For instance, the transition from *idle* to *ring* is made by the telephone switch (the user can never make a telephone ring) whereas all transitions to *idle* are made by the user. Fig. 5.2 depicts the transition diagrams for the user and the switch separately. Clearly, properties can be written down for each of these figures, as shown above.

In these diagrams, we have not shown the conditions under which a transition takes place. For instance, the switch causes a transition from *call* to *conn* only if a connection has been made to the dialed number, and the transition from *conn* to *idle* by the user is made as a result of the user's hanging up. Sometimes the condition cannot be expressed as a finite state property. To see this, let *num* be a variable in which the number dialed by the user is stored (in this extremely trivial description we ignore the fact that digits are stored one by one into *num*; instead, we assume that the entire number is stored into *num* in a single atomic action). Variable *num* can also take on a special value ϕ, which denotes that it contains no number.

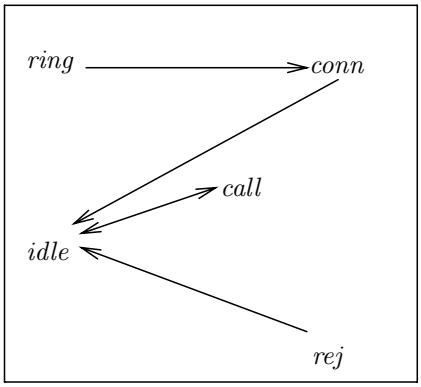
A pre-condition for the transition of user u from *call* to *conn* is that

its *num* differs from ϕ
its called number v is in *ring* state

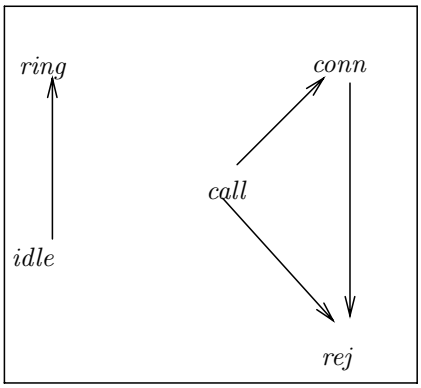
A post-condition of this action is that u and v are both in *conn* states or that u hangs up. This fact can be expressed by (where we prefix states by the user identity to avoid confusion): for all users u and v, where $u \neq \phi$ and $v \neq \phi$,

$$\begin{array}{l} u.\mathit{call} \ \wedge \ u.\mathit{num} = v \ \wedge \ v.\mathit{ring} \\ \qquad\qquad \textbf{co} \\ (u.\mathit{call} \ \wedge \ u.\mathit{num} = v \ \wedge \ v.\mathit{ring}) \\ \vee \ u.\mathit{idle} \ \vee (u.\mathit{conn} \ \wedge \ v.\mathit{conn}). \end{array}$$

However, this is a very coarse property; we cannot even state that a user is connected to exactly one party in state *conn*. By suitably introducing other variables (for instance, $u.\mathit{party}$ to denote the party to which u is connected in state *conn*; then $u.\mathit{conn} \wedge u.\mathit{num} \neq \phi \ \Rightarrow \ u.\mathit{party} = u.\mathit{num}$) we can obtain a more refined specification.



Transitions made by the user



Transitions made by the switch

Figure 5.2: Partitioning the transitions of Fig. 5.1 between the user and the switch

It becomes clear as we add more and more details to a specification that the finite state diagram provides a gross description of the flow of control. It is sometimes appealing to work with a diagram because a visual check may suffice to deduce a property. However, many of these systems have aspects that cannot be captured by any reasonable finite number of states.

5.5.6 Auxiliary variables

In stating and verifying properties of programs, it is sometimes necessary to introduce *auxiliary* variables, variables whose values at any point in the computation depend on only the *history* of the program variable values. For example, for variable u define an auxiliary variable v whose value is the number of times u's value has changed during the course of a computation. Now, v can be introduced into the program by defining its initial value to be zero, and incrementing it whenever u is changed. The problem with this approach is that the relationship between u and v as stated above is lost; it has to be gleaned from the program text. Since we have a logical operator to relate values of various variables over the course of a computation, we can define v directly, as follows.

initially $v = 0$
$u, v = m, n$ **co** $u, v = m, n \ \vee \ (u \neq m \ \wedge \ v = n + 1)$

This property asserts that (1) every change in u is accompanied by an incrementation of the value of v, and (2) as long as u does not change, v does not change either. It is now a simple matter to prove various facts about v, e.g., that v is nondecreasing.

A useful auxiliary variable is the sequence of distinct values written into a given variable; for a variable x let $\widehat{x}$ be this sequence. Then, for m and n of the appropriate type

initially $\widehat{x} = \langle x \rangle$ $\quad \{\langle x \rangle$ is the sequence consisting of the value of $x\}$
$x, \widehat{x} = m, n$ **co** $x, \widehat{x} = m, n \ \vee \ (x \neq m \ \wedge \ \widehat{x} = n +\!\!+ \ x)$

Consider a semaphore s, weak or strong, from section 4.9. Let auxiliary variables np and nv be the numbers of successful P and V operations. Initially, np and nv are both 0; define them by

$$
\begin{array}{l}
s, np, nv = m, n, r \\
\qquad \textbf{co} \\
(s, np, nv = m, n, r) \\
\vee \ (s, np, nv = m - 1, n + 1, r) \ \vee \ (s, np, nv = m + 1, n, r + 1)
\end{array}
$$

By applying the elimination theorem, it follows that $s - nv + np$ is constant. Since np and nv are zero initially, the initial value of $s - nv + np$ is the initial value of s, and this value is retained by $s - nv + np$ all through the computation.

5.5.7 Deadlock

A deadlock arises in a ring of processes when each process waits for its left neighbor. It is intuitively obvious that this system state persists forever. However, a formal proof or even a rigorous argument based on this verbal description is messy. Most such proofs are by contradiction: if this system state does not persist forever, there is a process that stops waiting; let x be the first process to stop waiting; x can stop waiting only if the process for which it has been waiting is not waiting; therefore, there is a process that stopped waiting earlier than x, contradicting our choice of x as the first process to stop waiting. Such informal arguments, though intuitively appealing, are neither precise about the assumptions they make —for instance, given that x is waiting for y, can x and y stop waiting simultaneously— nor concise.

"Process x waits for y" means that whenever both x and y are waiting, both continue to wait until y stops waiting. This informal description is quite ambiguous; it admits at least two interpretations, as shown below. Denote by w_x that process x is waiting (and similarly, w_y for y); then, x waits for y means

$$w_x \wedge w_y \;\; \textbf{co} \;\; w_x \qquad \text{(strong-wait)} \tag{D1}$$
$$w_x \wedge w_y \;\; \textbf{co} \;\; w_y \Rightarrow w_x \qquad \text{(weak-wait)} \tag{D2}$$

In strong-wait, both processes cannot stop waiting simultaneously; in weak-wait they can. By weakening the rhs of (D1) we get (D2). Therefore, strong-wait is indeed stronger than weak-wait.

First, we show that two processes x and y that are waiting strongly for each other are deadlocked. That is, **stable** $w_x \wedge w_y$. The proof avoids explicit arguments about time and arguments based on contradiction. The essence of the proof is induction on the number of computation steps, which is captured in the **co**-properties.

$$w_x \wedge w_y \;\; \textbf{co} \;\; w_x \qquad \text{, } x \text{ strong-waits for } y$$
$$w_y \wedge w_x \;\; \textbf{co} \;\; w_y \qquad \text{, } y \text{ strong-waits for } x$$
$$w_x \wedge w_y \;\; \textbf{co} \;\; w_x \wedge w_y \qquad \text{, conjunction of above two}$$

Next, we prove a stronger result for any finite ring of processes: if one process in a ring strong-waits and the remaining processes weak-wait, there is deadlock.

Let the processes be indexed 0 through N, $0 < N$, and w_i denote that process i is waiting. Suppose process 0 strong-waits for process N, and process $(i+1)$ weak-waits for process i, $0 \le i < N$.

$$w_0 \wedge w_N \;\; \textbf{co} \;\; w_0 \qquad \text{, use (D1) for strong-wait} \tag{D3}$$
$$\langle \forall i : \; 0 \le i < N : \; w_{i+1} \wedge w_i \;\; \textbf{co} \;\; w_i \Rightarrow w_{i+1} \rangle$$
$$\text{, use (D2) for weak-wait} \tag{D4}$$

This completes the mathematical modeling of the problem.

• Proof of **stable** $\langle \forall i :\ 0 \le i \le N :\ w_i \rangle$:

$\langle \forall i :\ 0 \le i < N :\ w_{i+1} \wedge w_i \rangle$ **co** $\langle \forall i :\ 0 \le i < N :\ w_i \Rightarrow w_{i+1} \rangle$
, take conjunction of (D4), over all i, $0 \le i < N$
$\langle \forall i :\ 0 \le i \le N :\ w_i \rangle$ **co** $w_0 \ \wedge\ \langle \forall i :\ 0 \le i < N :\ w_i \Rightarrow w_{i+1} \rangle$
, take conjunction of the above and (D3); simplify the lhs
stable $\langle \forall i :\ 0 \le i \le N :\ w_i \rangle$
, use the principle of mathematical induction in rhs □

Contrast this argument with a typical informal argument of the following kind. Consider the state of the system where all processes are waiting. If any process $(i+1)$, $i \ge 0$, stops waiting subsequently, process i must have stopped waiting then or before $(i+1)$. Apply the same argument to i, $i-1$, etc., to conclude that process 0 stops waiting at or before any other process stops waiting. However, process N has to stop waiting strictly prior to process 0. Apply the same kind of argument from process N down to $(i+1)$, to conclude that $(i+1)$ stops waiting before the time we postulated for it to stop waiting.

Our experience suggests that a temporal argument, particularly if it employs contradiction, can be replaced by a succinct formal proof using **co**-properties.

Waiting for any one resource

A more involved example of mutual waiting arises when a process can wait for any one of a set of processes. This is the typical situation where processes hold resources and a waiting process can proceed after receiving *any* of the resources for which it is waiting. Fig. 5.3 shows an example in which four processes, x, y, z and u, are deadlocked. The arrows from x (to y and z) denote that x is waiting for one of y and z. The waiting condition for x — using w_x, w_y, and w_z to denote that x, y, and z are waiting, respectively— is

$$w_x \wedge w_y \wedge w_z \ \ \textbf{co} \ \ w_x$$

Unlike the previous case, the existence of a cycle in the graph does not guarantee deadlock. A nonempty set S of nodes in a directed graph forms a *knot* if every outgoing edge from a node in S is to a node in S. Fig. 5.3 has the knots $\{x, y, z, u\}$ and $\{z, u\}$. We show that when all processes in a knot S are waiting in this sense, they are deadlocked. Let i be indexed over processes in S. As before, w_i denotes that i is waiting. Let process i have outgoing edges to nodes in S_i, $S_i \subseteq S$.

$w_i \ \wedge\ \langle \forall j :\ j \in S_i :\ w_j \rangle$ **co** w_i , waiting condition for process i
$\langle \forall i :\ i \in S :\ w_i \ \wedge\ \langle \forall j :\ j \in S_i :\ w_j \rangle \ \rangle$ **co** $\langle \forall i :\ i \in S :\ w_i \rangle$
, conjunction over all i in S
stable $\langle \forall i :\ i \in S :\ w_i \rangle$, simplify lhs using $S_i \subseteq S$

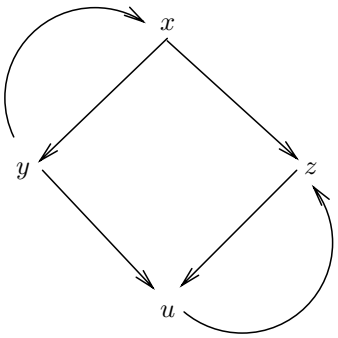


Figure 5.3: A knot of waiting processes

Observe that the type of waiting is captured by the **co**-property for each i, and the structure of the knot is exploited in simplifying the conjunction. An informal proof, we suspect, would be much longer.

5.5.8 Axiomatization of a communication network

In this example, we develop axioms to describe a message communicating network. To keep matters simple, suppose that we have two processes A and B that communicate via two one-way channels. The number of messages sent by a process is at least as large as the number received by the other, and both quantities are non-negative. Further, the number of messages sent and received along each channel is nondecreasing. A channel is *empty* if the number of messages sent equals the number of messages received along that channel. The state of a process is either *idle* or *active*. An idle process remains idle until it receives a message. Only active processes can send messages.

Our main interest is in formally specifying the properties of the system. We also show that when both processes are idle and both channels are empty, the system is terminated, in the sense that no further message is sent or received, nor will any of the processes become active.

In describing a system, we first have to decide on the variables whose values determine the state of the system. The choice is not always clear. In this case, for instance, we do not explicitly introduce a channel state. Instead, we introduce sA and sB to denote the number of messages sent by A and B and rA and rB for the number of messages received by A and B, respectively. The channel from A to B is *empty* if $sA = rB$. That the number of messages sent by a process is at least as large as the number received by the other and both quantities are non-negative is given by

$$\begin{array}{ll} \textbf{invariant } sA \geq rB \geq 0 & \\ \textbf{invariant } sB \geq rA \geq 0 & \text{(CN1)} \end{array}$$

The number of messages sent and received are nondecreasing. Using free variables m and n (of type natural)

$$\begin{array}{ll} \textbf{stable } sA \geq m, & \textbf{stable } sB \geq m \\ \textbf{stable } rA \geq n, & \textbf{stable } rB \geq n \end{array} \tag{CN2}$$

For the state of a process, boolean variable qA denotes that A is idle; similarly, qB. The following property says that each process remains idle as long as it receives no message.

$$\begin{array}{l} qA \ \wedge \ rA = m \ \ \textbf{co} \ \ rA = m \ \Rightarrow \ qA \\ qB \ \wedge \ rB = m \ \ \textbf{co} \ \ rB = m \ \Rightarrow \ qB \end{array} \tag{CN3}$$

Note that it is not specified if an idle process becomes active upon receiving a message; the specification allows the process to be in either state. That only active processes may send messages is best understood by "an idle process does not send messages", i.e.,

$$\begin{array}{l} qA \ \wedge \ sA = n \ \ \textbf{co} \ \ sA = n \\ qB \ \wedge \ sB = n \ \ \textbf{co} \ \ sB = n \end{array} \tag{CN4}$$

A digression There is a subtle point in the formulation of CN4. We prohibit an idle process from receiving a message, becoming active, and then sending a message, all in one step. To allow this possibility, we could write (CN4) analogously to (CN3):

$$\begin{array}{l} qA \ \wedge \ sA = n \ \ \textbf{co} \ \ sA = n \ \Rightarrow \ qA \\ qB \ \wedge \ sB = n \ \ \textbf{co} \ \ sB = n \ \Rightarrow \ qB \end{array}$$

Observe that with this specification, we can no longer prove termination: the specification does not say that an idle process has to receive a message *strictly* prior to becoming active and sending a message. Therefore, in a state where both processes are idle and both channels are empty, a next state is possible where both processes are idle, both channels are empty, *and* one extra message has been sent and received along every channel. □

This completes the axiomatization. The axioms capture only some aspects of the system behavior. In particular, we have no guarantee that all messages are eventually delivered (a progress property) or that messages are delivered in the same order in which they are sent (a safety property). It should be clear how to encode the fact that a channel is first-in–first-out. Exercise 15 asks for axioms analogous to (CN1–CN4) for arbitrary networks.

The state in which both processes are *idle* and both channels are *empty* is given by

$$qA \ \wedge \ qB \ \wedge \ sA, sB = rB, rA$$

We can show this predicate to be stable. But that is not enough; it leaves open the possibility, for instance, that the values of sA and rB could change

while preserving this predicate. Therefore, we prove that once this predicate holds, none of the variable values will change. Since $sA, sB = rB, rA$ from the predicate, it is sufficient to show that rA and rB do not change value; i.e., for free variables m and n we show:

- **stable** $qA \ \wedge \ qB \ \wedge \ rA, rB = sB, sA \ \wedge \ rA, rB = m, n$: (CN5)

  $qA \ \wedge \ rA = m$ **co** $rA = m \ \Rightarrow \ qA$
  , repeating (CN3)
  $qA \ \wedge \ sA = n$ **co** $sA = n$
  , repeating (CN4)
  $qA \ \wedge \ rA, sA = m, n$ **co** $(rA = m \ \Rightarrow \ qA) \ \wedge \ sA = n$
  , conjunction of above two
  $qB \ \wedge \ rB, sB = n, m$ **co** $(rB = n \ \Rightarrow \ qB) \ \wedge \ sB = m$
  , as above; for B use n, m in place of m, n

  $(qA \ \wedge \ qB) \ \wedge \ rA, sA = m, n \ \wedge \ rB, sB = n, m$
  co
  $(rA = m \ \Rightarrow \ qA) \ \wedge \ (rB = n \ \Rightarrow \ qB) \ \wedge \ sA, sB = n, m$
  , conjunction of above two

  stable $rA \geq m \ \wedge \ rB \geq n$
  , from CN2

  $(qA \ \wedge \ qB) \ \wedge \ rA, rB = sB, sA \ \wedge \ rA, rB = m, n$
  co
  $(rA, rB = m, n \ \Rightarrow \ qA \wedge qB) \wedge sA, sB = n, m \wedge rA \geq m \wedge rB \geq n$
  , conjoin above two, simplify

  $(qA \ \wedge \ qB) \ \wedge \ rA, rB = sB, sA \ \wedge \ rA, rB = m, n$
  co
  $(qA \ \wedge \ qB) \ \wedge \ rA, rB = sB, sA \ \wedge \ rA, rB = m, n$
  , from (CN1), $sA \geq rB$, $sB \geq rA$; simplify rhs

  stable $(qA \ \wedge \ qB) \ \wedge \ rA, rB = sB, sA \ \wedge \ rA, rB = m, n$
  , from above □

5.5.9 Coordinated attack

The following vicious version of a synchronization problem has received considerable attention in the literature. We show how it can be dealt with, relatively simply, using our theory. The following description of the problem is from Gray [77].

> Two divisions of an army are camped on two hilltops overlooking a common valley. In the valley awaits the enemy. It is clear that if both divisions attack the enemy simultaneously they will win the battle, whereas if only one division attacks it will be defeated. The divisions do not initially have plans for launching an attack on the enemy, and the commanding general of the first division wishes to coordinate a simultaneous attack (at some time the next day). Neither general will decide to attack unless he is sure that the other will attack with him. The generals can only communicate by means of a messenger. Normally, it takes the messenger one hour to get from one encampment to the other. However, it is possible that he will get lost in the dark or, worse yet, be captured by the enemy. Fortunately, on this particular night, everything goes smoothly. How long will it take them to coordinate an attack?

The original arguments to show that there is no protocol to achieve coordinated attack were long and cumbersome. Halpern and Moses [83] gave the first convincing proof of this fact based on notions of *knowledge* and *common knowledge*. We give a very brief outline of that theory as it pertains to this problem.

For process x and predicate p, predicate $x\ k\ p$ stands for "x knows p"; value of this predicate is a function of the state of x. Hence, $x\ k\ p$ does not change as long as the state of x does not change. For predicate p, predicate cp denotes that p is *common knowledge* (for some group of processes). We have, for any process x in the group,

$$cp \equiv x\ k\ (cp)$$

That is, p is common knowledge iff all processes know it to be common knowledge.

The coordinated attack problem requires us to establish common knowledge of the attack time. Halpern and Moses [83] showed that if no atomic action affects the state of more than one process, as is customarily the case in asynchronous message-communicating systems, then common knowledge cannot be gained. Later, Chandy and Misra [30] showed that under the same assumptions, common knowledge can be neither gained nor lost; that is, a previously unplanned attack cannot be coordinated, nor can a planned coordinated attack be called off. This result holds under the much weaker assumption that no action affects the states of *all* processes (though any proper subset of processes can be affected). We prove this more general result below.

Since an atomic action does not affect the states —hence the knowledge predicates— of *all* processes simultaneously, in a group with more than one process, knowledge (or ignorance) of at least one process is unchanged by

any action. We write this assertion —where x_i is process i, i is quantified over all processes in the group, and q_i is any predicate—

$$\langle \forall i :: x_i \; k \; q_i \rangle \quad \textbf{co} \quad \langle \exists i :: x_i \; k \; q_i \rangle \tag{CA1}$$
$$\langle \forall i :: \neg(x_i \; k \; q_i) \rangle \; \textbf{co} \; \langle \exists i :: \neg(x_i \; k \; q_i) \rangle \tag{CA2}$$

Theorem (common knowledge) In a group with more than one process where no atomic action affects the states of *all* processes, common knowledge can be neither gained nor lost; that is, for any predicate p,

constant cp

Proof: In (CA1, CA2), given above, instantiate every q_i by cp. Then, $(x_i \; k \; q_i) = (x_i \; k \; (cp))$. From the definition of cp, $(x_i \; k \; (cp)) = cp$. Hence,

$\langle \forall i :: cp \rangle$ **co** $\langle \exists i :: cp \rangle$
, from CA1
cp **co** cp , from above (i ranges over a nonempty set)
$\neg cp$ **co** $\neg cp$, similarly, from CA2
constant cp , from above two, using the definition of constant □

5.5.10 Dynamic graphs

This example illustrates how we express and deduce facts about dynamic data structures. The data structure here is a finite directed graph that can be changed by the following operation: all edges incident on a node may be directed toward that node in one (atomic) step.

A node that has no outgoing edge is called a *bottom* node. We are required to show that no path is ever created to a non-bottom node; i.e., if there is no path initially from node u to node v, $u \neq v$, there is no path from u to v at any point in the computation unless v is a bottom node at that point.

Let us first give a typical proof that draws on the well-known theorems and terminology of graph theory. Suppose that there is no path from u to v before a step. Call all the edges of the graph "old" at this time. Following the step that redirects all incident edges toward a node w, call the newly redirected edges to w "new". Suppose that there is a path from u to v following the step and v is non-bottom. Not all edges on this path are "old" because then there would have been a path before the step. Since some new edge is on this path, node w is on this path. We show that node w is not on the path, thus leading to a contradiction. Node w is different from v because v is assumed to be a non-bottom node, and w is a bottom node after the step. Node w is not an intermediate node on the path nor is $w = u$ because w has no outgoing edge.

Our formal proof avoids the temporal argument and proof by contradiction. However, the proof has to make explicit the notion of a path. In the following proof, u, v and w range over the nodes of the graph. Let

$u\ R^k\ v,\ k \geq 1$: there is a path of length k from u to v
$u\ R\ v$: there is a path from u to v
$u.\ \bot$: u is a bottom node

We define these predicates as follows:

$$u\ R^1\ v \quad \equiv \quad \text{there is an edge from } u \text{ to } v \tag{GR0}$$
$$u\ R^{k+1}\ v \equiv \quad \langle \exists w ::\ u\ R^1\ w\ \wedge\ w\ R^k\ v \rangle, \text{ for } k \geq 1 \tag{GR1}$$
$$u\ R\ v \quad \equiv \quad \langle \exists j :\ j \geq 1 :\ u\ R^j\ v \rangle \tag{GR2}$$
$$u.\ \bot \quad \equiv \quad \langle \forall v :\ \neg u\ R\ v \rangle \tag{GR3}$$

The operation that changes the graph is described by the following **co**-property. It states that no edge from u to v is created as long as v remains non-bottom. For all nodes u and v:

$$\neg u\ R^1\ v\ \ \mathbf{co}\ \ \neg u\ R^1\ v\ \vee\ v.\ \bot \tag{GR4}$$

We prove that no path from u to v is created as long as v remains a non-bottom node. Note the resemblance between the statement of the theorem and (GR4).

Theorem $\neg u\ R\ v\ \ \mathbf{co}\ \ \neg u\ R\ v\ \vee\ v.\ \bot$
Proof: We show that for all k, $k \geq 1$, and all u and v,

$$\neg u\ R^k\ v\ \ \mathbf{co}\ \ \neg u\ R^k\ v\ \vee\ v.\ \bot \tag{GR5}$$

Taking the conjunction of (GR5) over all k, $k \geq 1$, concludes the proof of the theorem. The proof of (GR5) is by induction on k.

Case $k = 1$**:** The result follows from (GR4).

Case $k+1$**:** We show $\neg u\ R^{k+1}\ v\ \ \mathbf{co}\ \ \neg u\ R^{k+1}\ v\ \vee\ v.\ \bot$, assuming that $\neg u\ R^k\ v\ \ \mathbf{co}\ \ \neg u\ R^k\ v\ \vee\ v.\ \bot$ holds for all u and v.

$\neg u\ R^1\ w\ \ \mathbf{co}\ \ \neg u\ R^1\ w\ \vee\ w.\ \bot$
, from (GR4), using w for v
$\neg w\ R^k\ v\ \ \mathbf{co}\ \ \neg w\ R^k\ v\ \vee\ v.\ \bot$
, use induction hypothesis (GR5) with w in place of u
$\neg u\ R^1\ w\ \vee\ \neg w\ R^k\ v\ \ \mathbf{co}\ \ \neg u\ R^1\ w\ \vee\ \neg w\ R^k\ v\ \vee\ w.\ \bot\ \vee v.\ \bot$
, disjunction of the above two
$\neg u\ R^1\ w\ \vee\ \neg w\ R^k\ v\ \ \mathbf{co}\ \ \neg u\ R^1\ w\ \vee\ \neg w\ R^k\ v\ \vee\ \neg w\ R\ v\ \vee\ v.\ \bot$
, use (GR3) to replace $w.\ \bot$ by $\neg w\ R\ v$ in the rhs
$\neg u\ R^1\ w\ \vee\ \neg w\ R^k\ v\ \ \mathbf{co}\ \ \neg u\ R^1\ w\ \vee\ \neg w\ R^k\ v\ \vee\ \neg w\ R^k\ v\ \vee\ v.\ \bot$
, use (GR2) to replace $\neg w\ R\ v$ by $\neg w\ R^k\ v$ in the rhs
$\langle \forall w ::\ \neg u\ R^1\ w\ \vee\ \neg w\ R^k\ v \rangle\ \ \mathbf{co}$
$\langle \forall w ::\ \neg u\ R^1\ w\ \vee\ \neg w\ R^k\ v \rangle\ \vee\ v.\ \bot$
, take conjunction of the above property over all w
$\neg u\ R^{k+1}\ v\ \ \mathbf{co}\ \ \neg u\ R^{k+1}\ v\ \vee\ v.\ \bot$
, simplify both sides using (GR1) □

We note, as a corollary, that once node u is outside any cycle —then $\neg u\ R\ u$ holds— u remains outside all cycles. In particular, if the graph is initially acyclic, it stays acyclic.

Corollary **stable** $\neg u\ R\ u$.
Proof:

$\neg u\ R\ v$ **co** $\neg u\ R\ v\ \vee\ v.\bot$	, theorem
$\neg u\ R\ u$ **co** $\neg u\ R\ u\ \vee\ u.\bot$	, replace v by u in the above
$u.\bot \Rightarrow\ \neg u\ R\ u$	, from (GR3)
$\neg u\ R\ u$ **co** $\neg u\ R\ u\ \vee\ \neg u\ R\ u$	, from above two
stable $\neg u\ R\ u$	, predicate calculus □

5.5.11 A treatment of real time

This section contains a brief description of real-time properties of action systems and how they may be expressed using **co**. In section 5.5.12 we consider a mutual exclusion algorithm that exploits real time; we construct its proof using the concepts introduced in this section.

We adopt the following idea from Abadi and Lamport [3]: the program has a variable *now* that represents the current time. This variable is real valued (it need not be positive). Its initial value is immaterial. Programmers have no control over how this variable is modified. All we may assume is that *now* is nondecreasing (see RT1), and *now* increases eventually beyond any specific real number (see RT2). RT2 is a progress property; progress properties are discussed in chapter 6. For completeness we include this property here though we do not use it any further. For any real r,

stable $now \geq r$	(RT1)
$true\ \mapsto\ now > r$	(RT2)

Note that the value of *now* may be different before and after an action execution, to model that the execution consumes some real time.

A real-time program makes use of one or more clocks that may proceed at different rates. Our treatment models a single clock; additional clocks may be introduced by having variables analogous to *now*, one for each clock. For instance, the fact that drift rate between a pair of clocks is bounded can be expressed as an invariant relating the corresponding variables, now_1 and now_2.

A program may utilize real time to delay certain actions, or a scheduler may schedule certain actions with specified frequency. Such requirements may be specified as safety properties using *now* and certain auxiliary variables that we describe next.

Punch of predicates

For predicate p let $\overline{p}$ be the last point in time p became *true*; call $\overline{p}$ the *punch* variable of p, for the time p punched the clock.[4] The following properties state how $\overline{p}$ is altered; here p is any predicate that does not include *now* or any punch variable.

$$\textbf{initially } \overline{p} \leq now \tag{RT3}$$
$$\textbf{initially } p \;\Rightarrow (\overline{\neg p} \leq \overline{p}) \tag{RT4}$$
$$p \;\wedge\; \overline{p} = r \;\; \textbf{co} \;\; \overline{p} = r \tag{RT5}$$
$$\neg p \;\wedge\; \overline{p} = r \;\; \textbf{co} \;\; (\neg p \;\wedge\; \overline{p} = r) \;\vee\; (p \;\wedge\; \overline{p} = now) \tag{RT6}$$

Properties (RT3, RT4) encode the initial conditions. (RT5) says that $\overline{p}$ is unaltered by the execution of an action if p holds before the execution, and (RT6) says that if p becomes *true* following an action execution —i.e., $\neg p$ holds before and p after the action execution— then $\overline{p}$ is assigned the value of *now* following the execution.

Note Rajeev Joshi has observed that (RT6) may be weakened to

$$\neg p \;\wedge\; \overline{p} = r \;\; \textbf{co} \;\; (\neg p \;\wedge\; \overline{p} = r) \;\vee\; (p \;\wedge\; r \leq \overline{p} \leq now)$$

that is, $\overline{p}$ is assigned a value between its old value and the current value of *now* if p becomes *true*. This definition allows more flexibility without affecting any of the results of this section. □

Variables $\overline{p}$ and $\overline{\neg p}$ may have the same value if, for instance, *now* does not change during an interval in which p is set to *false* and then to *true*. To avoid such possibilities we may require a strong monotonicity property: execution of any action that changes some variable value consumes some real time. Property (RT7), below, states this fact: if any predicate p is falsified by the execution of an action —i.e., some variable value has changed— then *now* strictly increases. We do not use strong monotonicity in our examples.

$$p \;\wedge\; now = r \;\; \textbf{co} \;\; p \;\vee\; now > r \tag{RT7}$$

The punch variables can be used to state real-time analogs of safety and progress properties. The requirement that once p becomes *true* it continues to remain *true* for at least δ time units —analogous to safety— is expressed by

$$\textbf{invariant } (now \leq \overline{p} + \delta \;\Rightarrow\; p) \tag{RT8}$$

That is, if no more than δ time units have elapsed since p last became *true*, then p is still *true*. Note that this condition is consistent with the initial condition, (RT4), in the sense that initial values can be assigned to all

[4] The notation chosen for punch becomes problematic when dealing with long formulae; in this book the notation does not cause a problem because predicates to which punch is applied are short in length.

punch variables that satisfy both (RT4) and (RT8). The requirement that once p becomes *true* it should be falsified within ϵ time units —analogous to progress— is expressed by

$$\textbf{invariant}\ (now > \overline{p} + \epsilon\ \Rightarrow\ \neg p) \qquad \text{(RT9)}$$

That is, if more than ϵ units have elapsed since p last became *true*, then p is *false*.

To state that execution of action α takes at most ϵ units to complete, introduce an auxiliary boolean variable q and augment α with the assignment $q := \neg q$. Then the following property expresses the timing constraint:

$$q = Q\ \wedge\ now = r\ \ \textbf{co}\ \ q = Q\ \vee\ now \leq r + \epsilon$$

where Q is a free variable. That is, any action that alters q —the only such action is α— increases now by no more than ϵ. A similar property can express a lower bound on the execution time of α.

Derived properties

We derive a few properties from (RT1–RT6), some of which are used in the proof of the real-time mutual exclusion algorithm of section 5.5.12. Henceforth, p is any predicate that does not mention now or any punch variable, and r is any real number.

• (RT10) **invariant** $\overline{p} \leq now$:
Proof: This proposition holds initially, from (RT3). We show next that $\overline{p} \leq now$ is stable.

$$\begin{array}{ll}
\overline{p} = r\ \ \textbf{co}\ \ (\overline{p} = r)\ \vee\ (p\ \wedge\ \overline{p} = now) & \text{, disjunction (RT5, RT6)} \\
\textbf{stable}\ now \geq r & \text{, (RT1)} \\
\overline{p} = r\ \wedge\ now \geq r\ \ \textbf{co}\ \ (\overline{p} = r\ \wedge\ now \geq r)\ \vee\ (\overline{p} = now) & \\
\qquad & \text{, conjunction, weaken rhs} \\
\overline{p} = r\ \wedge\ \overline{p} \leq now\ \ \textbf{co}\ \ \overline{p} \leq now & \text{, rewrite lhs, weaken rhs} \\
\textbf{stable}\ \overline{p} \leq now & \text{, disjunction over all } r \qquad \square
\end{array}$$

• (RT11) **stable** $\overline{p} \geq r$: Proof left to the reader. $\square$

• (RT12) **invariant** $p\ \Rightarrow (\overline{\neg p} \leq \overline{p})$:
Proof: From (RT4), $p\ \Rightarrow (\overline{\neg p} \leq \overline{p})$ holds initially. We show it is stable.

$$\begin{array}{ll}
p\ \wedge\ \overline{\neg p} = s\ \ \textbf{co}\ \ (p\ \wedge\ \overline{\neg p} = s)\ \vee\ (\neg p\ \wedge\ \overline{\neg p} = now) & \\
\qquad & \text{, (RT6) with } \neg p \text{ for } p \text{ and } s \text{ for } r \\
p\ \wedge\ \overline{\neg p} = s\ \ \textbf{co}\ \ \overline{\neg p} = s\ \vee\ \neg p & \text{, weaken rhs} \\
p\ \wedge\ \overline{p} = r\ \ \textbf{co}\ \ \overline{p} = r & \text{, (RT5)} \\
p\ \wedge\ \overline{p}, \overline{\neg p} = r, s\ \ \textbf{co}\ \ (\overline{p}, \overline{\neg p} = r, s)\ \vee\ \neg p & \\
\qquad & \text{, conjoin and weaken rhs} \\
p\ \wedge\ \overline{\neg p} \leq \overline{p}\ \ \textbf{co}\ \ (\overline{\neg p} \leq \overline{p})\ \vee\ \neg p & \text{, disjunction over } s \leq r \qquad (1)
\end{array}$$

$\neg p \ \wedge \ \overline{p} = r \ \ \mathbf{co} \ \ \neg p \ \vee \ \overline{p} = now$	, weaken rhs of (RT6)
$\neg p \ \ \mathbf{co} \ \ \neg p \ \vee \ \overline{p} = now$	, disjunction over r
$\neg p \ \ \mathbf{co} \ \ \neg p \ \vee \ \overline{\neg p} \leq \overline{p}$	, (RT10): $\overline{p} = now \ \Rightarrow \ \overline{\neg p} \leq \overline{p}$
$\neg p \ \vee \ \overline{\neg p} \leq \overline{p} \ \ \mathbf{co} \ \ \neg p \ \vee \ \overline{\neg p} \leq \overline{p}$	, disjunction of (1) and above
$\mathbf{stable} \ p \ \Rightarrow (\overline{\neg p} \leq \overline{p})$	, rewrite □

We leave it for the reader to prove the stronger result

$\mathbf{stable} \ p \ \Rightarrow (\overline{\neg p} < \overline{p})$

from the strong monotonicity property, (RT7).

• (RT13) $(\mathbf{invariant} \ p) \ \Rightarrow \ (\mathbf{constant} \ \overline{p}) \wedge (\mathbf{constant} \ \overline{\neg p})$:
Proof:

$p \ \wedge \ \overline{p} = r \ \ \mathbf{co} \ \ \overline{p} = r$	, (RT5)
$\overline{p} = r \ \ \mathbf{co} \ \ \overline{p} = r$	, substitution axiom: $\mathbf{invariant} \ p$
$\mathbf{constant} \ \overline{p}$	, definition of constant

Similarly,

$p \ \wedge \ \overline{\neg p} = r \ \ \mathbf{co} \ \ (p \ \wedge \ \overline{\neg p} = r) \ \vee \ (\neg p \ \wedge \ \overline{\neg p} = now)$	, (RT6) with $\neg p$ for p
$\overline{\neg p} = r \ \ \mathbf{co} \ \ \overline{\neg p} = r$	, substitution axiom: $\mathbf{invariant} \ p$
$\mathbf{constant} \ \overline{\neg p}$	, definition of constant □

• (RT14) $\langle \mathbf{stable} \ p \rangle \ \Rightarrow \ \langle \mathbf{stable} \ (p \ \wedge \ \overline{p} = r) \rangle$:
Proof:

$p \ \wedge \ \overline{p} = r \ \ \mathbf{co} \ \ \overline{p} = r$	, (RT5)
$\mathbf{stable} \ p$	, hypothesis
$p \ \wedge \ \overline{p} = r \ \ \mathbf{co} \ \ p \ \wedge \ \overline{p} = r$	, conjunction □

5.5.12 A real-time mutual exclusion algorithm

Fischer [70] has proposed a mutual exclusion algorithm that ingeniously exploits real time. We prove the correctness of this algorithm using only the fact that time is nondecreasing (RT1 of section 5.5.11). Other important facts about time —that eventually time increases beyond any bound (RT2 of section 5.5.11)— are unnecessary for this proof. The proof establishes a suitable invariant that implies mutual exclusion.

Informal description of the algorithm

There are N processes, numbered 1 through N, and a global variable x that assumes an integer value between 0 and N. Fig. 5.4 shows the state transitions of process i, $1 \leq i \leq N$. A process transits from e to a to wait for

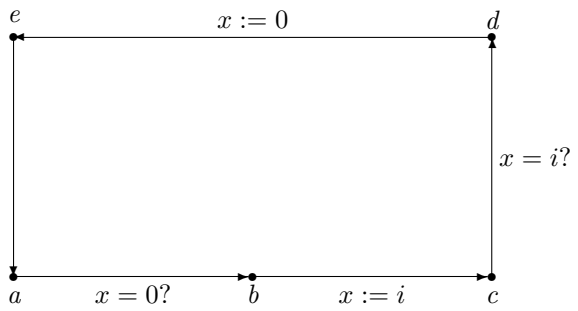


Figure 5.4: The state transitions of process i, $1 \leq i \leq N$

entry to its critical section. The edges of the other transitions are labeled with either an assignment, $x := i$ or $x := 0$, or a test, $x = 0?$ or $x = i?$. An assignment on an edge denotes that the state transition is accompanied by the assignment of the corresponding value to x. A test on an edge denotes that the transition takes place only if the test succeeds. Process state is d when it is in the critical section. Assume that all tests and assignments are atomic. Initially, all processes are in states e and $x = 0$.

It is easy to construct a scenario where two processes are in their critical sections simultaneously, given only the state transitions shown in Fig. 5.4. Timing constraints, given below, guarantee that this possibility is avoided.

(T1) Transition from b to c is completed within a unit of time.

(T2) Transition from c to d takes more than one unit of time.

Condition (T2) may be implemented by process i waiting for more than a unit of time before testing $x = i?$. In particular, the transition from c to d may never be completed.

Remark There is no fairness requirement (see section 6.2) on action executions. In particular, since there is no requirement that a process transit out of its d state, it may stay forever within its critical section, preventing other processes from entering their critical sections. □

Formal description of the algorithm

Let s_i denote the state of process i; s_i takes values from $\{a, b, c, d, e\}$. The initial state of the system is given by

initially $\langle \forall i :: \ s_i = e \rangle \ \wedge \ x = 0$.

The actions of process i are given below.

Process i

$$
\begin{array}{lll}
\quad \alpha_i :: \ s_i = e & \rightarrow & s_i := \ a \\
[\!] \ \beta_i :: \ s_i = a \ \wedge \ x = 0 & \rightarrow & s_i := \ b \\
[\!] \ \gamma_i :: \ s_i = b & \rightarrow & s_i, x := \ c, i \\
[\!] \ \delta_i :: \ s_i = c \ \wedge \ x = i & \rightarrow & s_i := \ d \\
[\!] \ \epsilon_i :: \ s_i = d & \rightarrow & s_i, x := \ e, 0
\end{array}
$$

end {Process i}

We use the following abbreviations: a_i for $s_i = a$, b_i for $s_i = b \ldots$, e_i for $s_i = e$. Observe that these predicates are disjoint, i.e., $a_i \ \wedge \ b_i \ \equiv \ \mathit{false}$, etc. We use three punch variables, $\overline{b_i}$, $\overline{c_i}$, and $\overline{d_i}$, in our proof.

Timing constraints

We can now state (T1,T2) formally. For all i, $1 \leq i \leq N$,

$$c_i \ \Rightarrow \ \overline{c_i} \leq 1 + \overline{b_i} \qquad \text{(T1)}$$

$$d_i \ \Rightarrow \ 1 + \overline{c_i} < \overline{d_i} \qquad \text{(T2)}$$

Proof of mutual exclusion

We establish that predicate D, below, is an invariant.

$$D :: \ \langle \forall i :: \ d_i \ \Rightarrow \ x = i \rangle$$

Mutual exclusion is then immediate:

$$(d_i \ \wedge \ d_j) \ \Rightarrow \ (x = i \ \wedge \ x = j)$$
$$(x = i \ \wedge \ x = j) \ \Rightarrow \ (i = j)$$

Predicate D holds initially. In the rest of this subsection we establish (see FM2) that D is stable in the given program augmented with (T1, T2). Henceforth, i and j range over 1 through N, and let

$$B = \langle max \ i :: \ \overline{b_i} \rangle$$

The structure of the proof is as follows. Property (FM2) establishes the main result, that D is stable. This proof relies on (FM1) —which is derived solely from the program text— and (FM3) —which is derived solely from the timing constraints (T1, T2). The crux of the proof of (FM2) is that if $d_j \wedge D$ holds in a state for some j, all actions except γ_i, $i \neq j$, preserve D. (FM3) shows that γ_i cannot be executed effectively in the given state.

• (FM1) **invariant** $(x = j \ \Rightarrow \ B \leq \overline{c_j})$:
Proof: This proposition holds initially since $x \neq j$, for all j, initially. Next, we show that $(x = j \ \Rightarrow \ B \leq \overline{c_j})$ is stable.

$\overline{b_i} \leq \overline{c_j}$ **co** $\overline{b_i} \leq \overline{c_j} \ \vee \ x = 0$, program text; see below
$\langle \forall i :: \ \overline{b_i} \leq \overline{c_j} \rangle$ **co** $\langle \forall i :: \ \overline{b_i} \leq \overline{c_j} \rangle \ \vee \ x = 0$
, conjunction over all i
$B \leq \overline{c_j}$ **co** $B \leq \overline{c_j} \ \vee \ x = 0$, rewrite
$x \neq j$ **co** $x \neq j \ \vee \ \overline{c_j} = now$, program text; see below
$x \neq j \ \vee \ B \leq \overline{c_j}$ **co** $x \neq j \ \vee \ \overline{c_j} = now \ \vee \ B \leq \overline{c_j} \ \vee \ x = 0$
, disjunction of above two
$\overline{c_j} = now \ \Rightarrow \ B \leq \overline{c_j}$ and $x = 0 \ \Rightarrow \ x \neq j$
, (RT10) from page 128; arithmetic
stable $(x = j \ \Rightarrow \ B \leq \overline{c_j})$, from above two, simplify rhs

The following property, which is used in the proof of (FM1),

$$\overline{b_i} \leq \overline{c_j} \ \textbf{co} \ \overline{b_i} \leq \overline{c_j} \ \vee \ x = 0$$

can be derived from the program text; we need consider only those actions that may change $\overline{b_i}$ or $\overline{c_j}$. Variable $\overline{b_i}$ is changed only by the effective execution of β_i, which has $x = 0$ as a post-condition. Whenever $\overline{c_j}$ is changed it is set to *now*, thus preserving $\overline{b_i} \leq \overline{c_j}$.

The following property, which is used in the proof of (FM1),

$$x \neq j \ \textbf{co} \ x \neq j \ \vee \ \overline{c_j} = now$$

follows from the program text, because the only action that can falsify $x \neq j$ is γ_j, which sets c_j to *true*, so $\overline{c_j}$ to *now*. □

• (FM2) **stable** D:
Proof:

$d_j \ \wedge \ B \leq \overline{c_j} \ \wedge \ D$ **co**
$\quad D \ \vee \ (d_j \ \wedge \ B \leq \overline{c_j} \ \wedge \ \langle \exists i : \ i \neq j : \ c_i \ \wedge \ \overline{c_i} = now \rangle)$
, program text; see below
$d_j \ \wedge \ B \leq \overline{c_j} \ \wedge \ D$ **co** D , apply corollary of FM3 on rhs
$d_j \ \wedge \ D$ **co** D , $(d_j \wedge D) \Rightarrow (x = j) \Rightarrow \{\text{FM1}\} \ (B \leq \overline{c_j})$
$\langle \exists j :: \ d_j \rangle \ \wedge \ D$ **co** D , disjunction over all j (1)
$\neg d_j$ **co** $(d_j \ \Rightarrow \ x = j)$, program text; see below
$\langle \forall j :: \ \neg d_j \rangle$ **co** $\langle \forall j :: \ d_j \ \Rightarrow \ x = j \rangle$, conjunction over all j
$\neg \langle \exists j :: \ d_j \rangle$ **co** D , rewrite lhs, rhs
$\neg \langle \exists j :: \ d_j \rangle \ \wedge \ D$ **co** D , strengthen lhs
stable D , disjunction of above and (1)

The following property, which is used in the proof of (FM2),

$$d_j \ \wedge \ B \leq \overline{c_j} \ \wedge \ D \ \textbf{co} \ D \ \vee \ (d_j \wedge \ B \leq \overline{c_j} \ \wedge \ \langle \exists i : \ i \neq j : \ c_i \wedge \overline{c_i} = now \rangle)$$

follows from the program text. Consider the actions of the program in two groups: actions of process j and the remaining actions. In the first group, the only action that may be executed effectively given that d_j holds is ϵ_j, and its execution preserves D. Next, consider the actions of process i, $i \neq j$.

Action α_i preserves D. Neither β_i nor δ_i is executed effectively in a state where $d_j \wedge D$ holds, because $(d_j \wedge D) \Rightarrow (x = j)$ and the pre-conditions for effective executions of β_i and δ_i are $x = 0$ and $x = i$, respectively. Action ϵ_i is not executed effectively either because

$$\begin{array}{ll} & (d_j \wedge D) \\ \Rightarrow & \{\text{from } D\} \\ & x = j \\ \Rightarrow & \{\text{arithmetic}\} \\ & \langle \forall k :\ k \neq j :\ x \neq k \rangle \\ \Rightarrow & \{\text{from } D\} \\ & \langle \forall k :\ k \neq j :\ \neg d_k \rangle \end{array}$$

Hence, the pre-condition d_i for the effective execution of ϵ_i does not hold. Therefore, the only action that may be executed effectively is γ_i, for some i, $i \neq j$, which preserves $d_j \wedge\ B \leq \overline{c_j}$ (because it does not change either d_j, any b_i or c_j) and establishes $c_i\ \wedge\ \overline{c_i} = now$ as a post-condition.

The following property, which is used in the proof of (FM2),

$$\neg d_j\ \ \textbf{co}\ \ (d_j\ \Rightarrow\ x = j)$$

follows from the program text: the only action that falsifies $\neg d_j$, i.e., establishes d_j, is δ_j, and it preserves its pre-condition $x = j$. □

• (FM3) $(d_j\ \wedge\ B \leq \overline{c_j}\ \wedge\ c_i\ \wedge\ \overline{c_i} = now)\ \equiv\ \mathit{false}$, for all i and j:
Proof: We derive a contradiction, $\overline{d_j} < \overline{d_j}$, from the predicate in the lhs and (T1, T2).

$$\begin{array}{ll} & \overline{d_j} \\ \leq & \{\text{invariant (RT10) from page 128}\} \\ & now \\ = & \{\overline{c_i} = now, \text{ from the lhs predicate}\} \\ & \overline{c_i} \\ \leq & \{c_i \text{ from the lhs predicate and T1: } c_i\ \Rightarrow\ \overline{c_i} \leq 1 + \overline{b_i}\} \\ & 1 + \overline{b_i} \\ \leq & \{B = \langle max\ i ::\ \overline{b_i} \rangle\} \\ & 1 + B \\ \leq & \{B \leq \overline{c_j}, \text{ from the lhs predicate}\} \\ & 1 + \overline{c_j} \\ < & \{d_j \text{ from the lhs predicate and T2: } d_j\ \Rightarrow\ 1 + \overline{c_j} < \overline{d_j}\} \\ & \overline{d_j} \end{array}$$

□

By taking disjunction of (FM3) over all i, $i \neq j$,

Corollary $(d_j\ \wedge\ B \leq \overline{c_j}\ \wedge\ \langle \exists i :\ i \neq j :\ c_i\ \wedge\ \overline{c_i} = now \rangle)\ \equiv\ \mathit{false}$

5.6 Theoretical Results

In this section, we define the notions of *weakest pre-condition*, *strongest post-condition*, and *strongest invariant*. We show that predicate *FP* can be defined as the weakest solution of an equation.

5.6.1 Strongest rhs; weakest lhs

In a given program, for any predicate p there is a strongest q such that p **co** q; similarly, for any q there is a weakest p such that p **co** q. We prove only the first result; the proof of the second one is similar.

Theorem (strongest rhs) For any p there exists a q such that

$$p \ \textbf{co} \ q \quad \text{and}$$
$$\langle \forall r : \ p \ \textbf{co} \ r : \ q \ \Rightarrow \ r \rangle$$

Proof: Predicate q is the conjunction of the rhs of all **co**-properties in which p appears in the lhs:

$$q \ \equiv \ \langle \wedge \, b : \ p \ \textbf{co} \ b : \ b \rangle$$

Since **co** is universally conjunctive, we have p **co** q. Any r for which p **co** r holds, we get $q \Rightarrow r$ from the definition. □

5.6.2 Strongest invariant

Every program has a (unique) strongest invariant. This is proved by defining the strongest invariant to be the conjunction of all invariants. We give a longer alternative proof that provides a "constructive" procedure for obtaining the strongest invariant. The essence of the procedure is to start with the initial condition —call it p_0— and obtain a sequence of p_is where p_i **co** p_{i+1} and p_{i+1} is the strongest rhs for p_i. The disjunction of all the p_is is the strongest invariant.

Lemma Let $p_0, \ldots, \ p_i, \ldots$ be an infinite sequence where, for all i, $i \geq 0$,

$$p_i \ \textbf{co} \ p_{i+1}$$

Then, **stable** $\langle \exists i :: \ p_i \rangle$.
Proof: Taking the disjunction over all the **co**-properties, p_i **co** p_{i+1}, we get

$$\langle \exists i : \ i \geq 0 : \ p_i \rangle \ \textbf{co} \ \langle \exists i : \ i \geq 0 : \ p_{i+1} \rangle$$

Weakening the rhs by p_0 as a disjunct yields the result. □

Theorem (strongest invariant) Let p_0 be the initial condition. For all i, $i \geq 0$, let p_i **co** p_{i+1}, where p_{i+1} is the strongest rhs for p_i. Then, $\langle \exists i :: \ p_i \rangle$ is the strongest invariant.

Proof: Let $P = \langle \exists i :: p_i \rangle$. From the previous lemma, P is stable. Further, since $p_0 \Rightarrow P$, initially P holds. Therefore, P is an invariant.

To show that P is the strongest invariant, we show that for any invariant J, $P \Rightarrow J$, i.e. $\langle \exists i :: p_i \rangle \Rightarrow J$, or equivalently $\langle \forall i :: p_i \Rightarrow J \rangle$.

• Proof of $\langle \forall i :: p_i \Rightarrow J \rangle$: The proof is by induction on i.

$p_0 \Rightarrow J$: Since p_0 is the initial condition, $p_0 \Rightarrow J$.

Assume $p_i \Rightarrow J$ and show $p_{i+1} \Rightarrow J$:

p_i **co** p_{i+1}	, given
$p_i \wedge J$ **co** $p_{i+1} \wedge J$	, stable conjunction with J
p_i **co** $p_{i+1} \wedge J$	, $p_i \wedge J \equiv p_i$ since $p_i \Rightarrow J$
$p_{i+1} \Rightarrow p_{i+1} \wedge J$	, p_{i+1} is the strongest rhs in any **co**-property whose lhs is p_i
$p_{i+1} \Rightarrow J$	, predicate calculus

□

The strongest invariant includes the initial condition, p_0, as a disjunct. Therefore, if p_0 is not identically *false*, then neither is the strongest invariant. Conversely, if p_0 is *false*, so is p_1 —because *false* is stable— so the strongest invariant is *false*. We have often used the term "reachable states" informally in this chapter. Formally, a state is *reachable* iff it satisfies the strongest invariant.

5.6.3 Fixed point

Predicate *FP* characterizes the set of states that do not change as a result of program execution. We define *FP* using **co**. Observe that any subset of states that satisfy *FP* is stable, i.e., for any predicate b

$$\textbf{stable}\ (FP \wedge b)$$

However, this does not identify a unique *FP*. In fact, $\mathit{false} \wedge b$ is stable for any b. We define *FP* to be the *weakest* predicate p such that $p \wedge b$ is stable for all b. It can be shown that *FP* has the following closed form (see exercise 17).

$$FP \equiv \langle \exists p : (\forall b :: \textbf{stable}\ p \wedge b) : p \rangle$$

As is the case for the strongest invariant, it is difficult to compute *FP* using the above formulation; we have shown in section 5.3.2 how to obtain *FP* by syntactic manipulations of the program text.

A program is "deadlock-free" iff *FP* is always *false*, i.e., $\neg FP$ is always *true*. Using the substitution axiom, $\neg FP$ is then invariant.

5.7 Concluding Remarks

This section contains an assessment of the strengths and weaknesses of **co**. Earlier, we had gained considerable experience in writing and manipulating **unless** properties [32, 165], a predecessor of **co**. There is ample reason to believe that **co** would be at least as powerful for expressing the safety properties. The manipulation rules for **co** are simple and effective. The examples in section 5.5 —particularly common meeting time (section 5.5.2), deadlock (section 5.5.7), and coordinated attack (section 5.5.9)— were handled by concise proofs.

The proposed theory has several limitations. The first concerns expressibility. Some safety properties cannot be expressed using **co** or can be expressed only with some difficulty. A property that cannot be expressed is as follows: for each value n between 0 and 9 there exists a finite execution of the program so that x has value n at the end of the execution. This kind of property is succinctly expressed using branching time temporal logic. Our theory, based on linear temporal logic, is inadequate in such cases.[5] This limitation, though, is deliberate; we have tried to avoid elaborate theories, and branching time logics would require such theories.

Temporal logic is an elegant extension of classical logic that includes the temporal operators $\Box$ and $\Diamond$ (read *always* and *eventually*). The primary operator for expressing the safety properties is $\Box$; for example, the temporal formula $p \;\Rightarrow\; \Box p$ denotes that once p is *true* it is always *true*; i.e., p is stable. Temporal logic has a well-developed theory, and it has been applied to a variety of problems in computer science; for a treatment of the theory see Manna and Pnueli [127, 128]. Some properties are easily expressible in linear temporal logic but are difficult to express using **co**. Consider the property: once x exceeds 5, it remains positive. In temporal logic, this is simply

$$x > 5 \;\Rightarrow\; \Box(x > 0)$$

Such a property cannot be written directly using **co** because whether $x > 0$ holds in some state depends not just on the preceding state but if x has exceeded 5 in the past. Introducing an auxiliary boolean variable b that is *true* iff x has ever exceeded 5, we can specify the desired behavior by

$$\textbf{initially } b \;\equiv\; x > 5$$
$$x > 5 \;\Rightarrow\; b$$
$$\textbf{stable } (b \;\wedge\; x > 0)$$

[5] However, see chapter 7; a program that generates each value between 0 and 9 in some execution is a maximal program for the specification that some value between 0 and 9 is generated.

Though the specification in temporal logic is succinct, for a given program the proof of $x > 5 \;\Rightarrow\; \Box(x > 0)$ would require the introduction of a predicate analogous to b.

Event predicates, see Hehner [85] and Hehner and Hoare [86], and TLA, see Lamport [118], have proved useful for describing safety properties. An example of an event predicate is $x' \geq x$, which says that the value of x after any action —this value is denoted by x'— is at least the value of x before the action —the value before the action is denoted by x. This formalism is attractive because the inference rules are simply those of predicate calculus. Lamport combines event predicates, temporal operators and quantification (over variables and actions) to obtain a powerful logic, called TLA. He advocates using the same logic for representing both a system and its properties, thus simplifying the proofs of implementations.

A disadvantage of our theory (compared to event predicates and TLA) is that we often have to introduce free variables in stating the properties. For instance, "x is nondecreasing" is written as

$$\textbf{stable}\ x \geq m$$

with a free variable m, whereas

$$x' \geq x$$

is an event predicate that expresses the same fact. We have found free variables to be particularly useful in the treatment of progress properties and in constructing proofs by induction on the values of free variables.

Our choice of logical operators was influenced by the desire to employ induction as the basis of our proofs. Hence, we rejected the primary temporal-logic operators $\Box$ (always) and $\Diamond$ (eventually), in favor of **co**, **ensures**, and $\mapsto$ (*leads-to*) of chapter 6. A stable property is justified by induction on the number of program steps where each step preserves the predicate, **ensures** provides a mechanism to capture (a weak form of) fairness, and *leads-to* is proved by induction on the number of "**ensures** steps".

5.8 Bibliographic Notes

The notions of pre- and post-conditions are from Floyd [71] and Hoare [89]. The *wp*-calculus, a treatment of predicate transformers, and their applications in program semantics are in Dijkstra and Scholten [61]; see appendix A.4.2 for a brief treatment. See Apt and Olderog [12] or Francez [73] for applications of these ideas in program verifications. Portions of this chapter appeared earlier in Misra [139]; they are reprinted here with permission from Elsevier Science.

Lamport [113] was the first to coin the terms *safety* and *liveness* in the sense used here (we use *progress* instead of *liveness*), and he gave the first formal definition of safety [115]; the formal definition of progress appears

in Alpern and Schneider [7]. There are a number of papers on the substitution axiom, in particular Sanders [158], Knapp [108], and Misra [134, note 14]; the rationale given in page 102 is from Knapp. A clear example of the distinction between invariant and always *true* is in van Gasteren and Tel [168].

State diagrams have been used very effectively in Harel and Politi [84] for descriptions and designs of reactive systems. The treatment of auxiliary variables in section 5.5.6 follows Misra [134, note 15]. See Fagin et al. [68] for a detailed knowledge-based treatment of the coordinated attack problem. The treatment of dynamic graphs is based on Chandy and Misra [32, chapter 12] and Misra [134, note 2].

See Lamport [117] and Sanders [158], for the notion of the strongest invariant. For completeness of UNITY logic see Jutla, Knapp, and Rao [103] and Cohen [42].

The axioms for ring network in section 5.5.8 are from Misra [135, section 6.1]. A more general case, based on Chandy and Misra [33], is treated in exercise 15. The idea of encoding real time in variable *now*, as described in section 5.5.11, is from Abadi and Lamport [3]. The formalization using **co** and the notion of punch variables are due to Carruth [27]. The real-time mutual exclusion algorithm of section 5.5.12 is due to Fischer [70], who never published it; the algorithm is described in Lamport [116, section 2] and a simplified version of it appears in Abadi and Lamport [3, section 3.4]. The proof of Fischer's algorithm given here closely follows Carruth and Misra [28]; Shankar [162] has constructed a mechanical proof of this algorithm using the theorem prover PVS [147]. Schneider, Bloom, and Marzullo [160] contains ideas similar to punch variables and also one of the earliest proofs of this algorithm. Emerson et al. [66] describes the use of RTCTL (Real-Time Computation Tree Logic) for model checking hard real-time systems.

Rutger Dijkstra [63] has developed an algebraic theory, called computation calculus, with which he has axiomatized the UNITY logic; the work is a promising new direction for combining state-based and action-based reasoning. Hoare and He [93] have proposed an ambitious approach to unify a large number of theories for specifications and designs of programs, ranging from purely declarative to sequential and distributed programs.

Probabilistic safety properties have been studied in Morgan, McIver, and Seidel [143] and Rao [155]; the former proposes predicate transformers with arithmetic operators, whereas the latter introduces logical operators in the style of work presented in this chapter.

Bisimulation —see Milner [132]— represents an entirely different approach to correctness that is particularly effective in verifying that two action systems are equivalent.

There have been several successful efforts at mechanizing the UNITY logic; notable are Andersen, Petersen, and Pettersson [9, 10], Heyd and Crégut [88], and Paulson [150]. Goldschlag [76] has implemented a ver-

sion of this logic on top of the Boyer-Moore theorem prover [21, 22]. Kaltenbach [104] has developed a model checker for finite state UNITY programs that computes the strongest invariant automatically.

5.9 Exercises

1. (Formal manipulation) Prove the following.

   (a) $\dfrac{p \wedge q \;\; \textbf{co} \;\; p, \;\; \textbf{stable} \; \neg q}{p \;\; \textbf{co} \;\; p \vee \neg q}$

   (b) $\dfrac{p \;\; \textbf{co} \;\; q, \;\; \neg p \;\; \textbf{co} \;\; \neg q}{\textbf{constant} \; p}$

   (c) (cancellation) $\dfrac{p \;\; \textbf{co} \;\; q \vee r, \;\; q \;\; \textbf{co} \;\; b}{p \;\; \textbf{co} \;\; b \vee r}$

   (d) $\dfrac{p \wedge \neg FP \;\; \textbf{co} \;\; p}{\textbf{stable} \; p}$ (FP is the fixed point of the program)

   (e) $\dfrac{\textbf{invariant} \; I, \;\; I \;\; \textbf{co} \;\; p}{\textbf{stable} \; p}$

   Hint: Use the substitution axiom.

   (f) **constant** 3

   (g) $\dfrac{\textbf{constant} \; x + y, \;\; \textbf{constant} \; y}{\textbf{constant} \; x}$

2. (Manipulation of sets) Prove the following where g is any set, G and x are free variables and p and q are predicates that do not contain free occurrences of x.

   (a) $\dfrac{p \;\wedge\; x \in g \;\; \textbf{co} \;\; x \in g \;\vee\; q}{p \;\wedge\; g \supseteq G \;\; \textbf{co} \;\; g \supseteq G \;\vee\; q}$

   (b) $\dfrac{p \;\wedge\; x \notin g \;\; \textbf{co} \;\; x \notin g \;\vee\; q}{p \;\wedge\; g \subseteq G \;\; \textbf{co} \;\; g \subseteq \; G \;\vee\; q}$

   Hint: Note that $g \subseteq G \;\equiv\; \langle \forall x :: \; x \in g \;\Rightarrow\; x \in G \rangle$.

3. (Proving by parts) Let $p(x, y)$ and $q(x, y)$ be predicates that name only program variables x and y. To prove

   $$p(x, y) \;\; \textbf{co} \;\; q(x, y)$$

   the following strategy is suggested: for free variables m and n, prove

   $$p(x, n) \;\; \textbf{co} \;\; q(x, n)$$
   $$p(m, y) \;\; \textbf{co} \;\; q(m, y)$$

   Show that this is not a valid strategy. Next, show that the strategy is valid if x and y are never changed simultaneously.

4. (Substitution axiom) Show that **constant** y holds in program *distinction* of section 5.3.1.

5. (Substitution axiom) Given that **invariant** $x = y$ holds in a program, show that x can be replaced by y in any **co**-property.

6. (Deriving properties from program text) For the following program, *Alternate*, show that

   (a) x remains unchanged as long as it differs from y, i.e., for any m,

   $$x = m \ \wedge \ x \neq y \ \ \textbf{co} \ \ x = m$$

   (b) **invariant** $0 \leq x - y \leq 1$

   (c) The program is deadlock-free.

---

program *Alternate*
  nat $x = 1$;
  nat $y = 0$;

  $x = y \ \rightarrow \ x := x + 1$
$[\!]$ $x \neq y \ \rightarrow \ y := y + 1$
end {*Alternate*}

---

7. (Elimination theorem) Prove, for integer program variables x and y, and free variables m and n:

   (a) $(x = m \ \ \textbf{co} \ \ x \geq m) \ \equiv \ (\textbf{stable} \ x \geq n)$

   (b) $$\frac{x, y = m, n \ \ \textbf{co} \ \ x, y = m, n \ \vee \ (m > n \ \wedge \ x, y = m - 1, n)}{\textbf{stable} \ x \geq y}$$

   (c) $$\frac{x, y = m, n \ \ \textbf{co} \ \ x, y = m, n \ \vee \ x, y = m + 1, n - 1}{\textbf{constant} \ x + y}$$

   (d) If x is nondecreasing and y is nonincreasing, then $x \geq y$ is stable.

   (e) If a set is nongrowing, its size is nonincreasing.

   (f) $$\frac{x = m \ \ \textbf{co} \ \ p \qquad p \text{ names neither } m \text{ nor any variable other than } x}{\textbf{stable} \ p}$$

8. (Elimination theorem) Let f be a function that does not name m or any program variable other than x. Show, for any m and n, and a binary relation $\sim$ that

(a)
$$\frac{\begin{array}{c} x = m \;\; \textbf{co} \;\; f.x \;\sim\; f.m \\ \sim \text{ transitive} \end{array}}{\textbf{stable}\ f.x \;\sim\; n}$$

(b)
$$\frac{x = m \;\; \textbf{co} \;\; f.x = f.m}{\textbf{constant}\ f.x}$$

(c) Function f preserves relation $\sim$ iff $x \;\sim\; y \;\Rightarrow\; f.x \;\sim\; f.y$. Show:

$$\frac{\begin{array}{c} x = m \;\; \textbf{co} \;\; x \;\sim\; m \\ f \text{ preserves } \sim \\ f \text{ names neither } m \text{ nor program variables other than } x \end{array}}{f.x = n \;\; \textbf{co} \;\; f.x \;\sim\; n}$$

9. (A generalization of the elimination theorem) Consider a set of **co**-properties whose lhs, collectively, we call L and its rhs R. We write L **co** R to denote these properties. It is given that L' **co** R' can be deduced from L **co** R. Suppose that each property in the lhs L is strengthened by conjoining predicate p and in R weakened by having disjunction with q; call the new set of properties $L \wedge p$ **co** $R \vee q$. Show that $L' \wedge p$ **co** $R' \vee q$ can be deduced from $L \wedge p$ **co** $R \vee q$.

   Apply this result to prove the following generalization of the elimination theorem.

$$\frac{\begin{array}{c} r \wedge x = m \;\; \textbf{co} \;\; q \vee r' \text{ , where } m \text{ is free} \\ r \text{ and } r' \text{ do not name } m \\ p \text{ names neither } m \text{ nor program variables other than } x \end{array}}{r \wedge p \;\; \textbf{co} \;\; \langle \exists m :: \; p[x := \; m] \;\wedge\; q \rangle \vee r'}$$

10. (From prose to formula) Convert the following verbal descriptions into formal properties and then establish the conclusions wherever suggested. Note that, like most verbal descriptions, each of the following admits several possible interpretations.

    (a) Predicates p and q change synchronously.

    Hint: There is a neat way to write this.

    (b) For an integer variable x, let sx be an auxiliary variable that is increased by the new value of x whenever x is changed; sx is not changed otherwise. Show that if x is always non-negative, $sx \geq x$ is stable.

    (c) Let x and y be integer variables. Let c be an auxiliary variable that counts the number of assignments to x or y that causes x to exceed y. Show that if $x \leq y$ is invariant, $c = r$ is stable for any r.

    (d) Formalize: Once integer x exceeds 5, it never decreases.

(e) A vertex (in a directed graph) remains *black* as long as it has only black neighbors. Show that the state where all vertices are black persists.

11. (More prose to formula) An impermeable vessel consists of chambers numbered 0 through N. The particles inside the vessel move according to the following law: a particle may stay in its current chamber or move to a higher-numbered chamber. Particles cannot leave the vessel or enter it. Show that

    (a) The set of particles in the chambers at or above j, $0 \leq j \leq N$, is nonshrinking.

    (b) The set of particles in the chambers at or below j, $0 \leq j \leq N$, is nongrowing.

12. (Another safety operator) For a variable or a group of variables x write

    $$x : \ p \ \rightarrow \ q$$

    to denote that (in a given program) x can change only if p is a pre-condition and every change in x guarantees q as a post-condition. Express this property using **co**. Show that

    (a) $\dfrac{x : \ p \ \rightarrow \ q}{x : \ p \vee p' \ \rightarrow \ q \vee q'}$

    (b) $\dfrac{x : \ p \ \rightarrow \ q \ , \ x : \ p' \ \rightarrow \ q'}{x : \ p \wedge p' \ \rightarrow \ q \wedge q'}$

13. (Fixed point) Show programs whose *FP*s are as follows. Variable x has type integer.

    (a) *true*

    (b) $x \neq 0$

    (c) $x \leq 0$

    (d) $\langle \exists n :: \ x^n = 64 \rangle$. Here n is a positive integer.

14. (Finite state descriptions) There are two communicating processes, P and Q; any message sent by P is acknowledged by Q. In this exercise, we consider the transfer of a single message and the corresponding acknowledgment. The state of the system is given by two boolean variables, p and q. Initially, both variables are *false*. Variable p is set to *true* by P when it sends a message and set to *false* when it receives an acknowledgment. Variable q is set to *true* when Q receives the message; it remains *true* thereafter.

    Draw a state transition diagram. Construct the **co**-properties. Find an equivalent set of properties where a single literal ($p, q, \neg p$, or $\neg q$) appears in the lhs.

15. (Axiomatizing communication networks) Consider a ring network consisting of at least two processes. Develop a formalization of its properties along the same lines as in section 5.5.8. Then provide a similar formalization for an arbitrary network.

    Suggested notation: For a ring network, let i' denote the process to which process i sends messages. For channel c in an arbitrary network, let $s.c$ and $r.c$ denote the number of messages sent and received along c. For sets of processes x and y use xy to denote the set of channels from a process in x to a process in y. Use the following abbreviations:

    $$(s = r).xy \;\equiv\; \langle \forall c :\; c \in xy :\; s.c = r.c \rangle$$
    $$(s \geq r).xy \;\equiv\; \langle \forall c :\; c \in xy :\; s.c \geq r.c \rangle$$

    Similarly, $(s = L).xy$ or $(r = L).xy$ for free L are defined. In both cases, prove results analogous to (CN5). The problem for the general network has been treated in [33].

16. Show that for any predicate p there exists (in a given program) a strongest stable predicate weaker than p. Denote this predicate by $ss.p$ and show that

    (a) ss is monotonic: $(p \;\Rightarrow\; q) \;\Rightarrow\; (ss.p \;\Rightarrow\; ss.q)$
    (b) **stable** $p \;\equiv\; (ss.p \equiv p)$
    (c) ss is idempotent: $ss.(ss.p) \;\equiv\; ss.p$
    (d) ss is universally disjunctive: $ss.\langle \exists\; i ::\; p_i \rangle \;\equiv\; \langle \exists\; i ::\; ss.p_i \rangle$

    For the weakest stable predicate stronger than p, similar facts may be deduced.

17. (Closed form for fixed point) In this exercise, x, b, p, and FP are predicates. Define FP by

    $$FP \;\equiv\; \langle \exists p :\; \langle \forall b ::\; \textbf{stable}\ (p \wedge b) \rangle :\; p \rangle$$

    Show that FP is the weakest solution (in x) to

    $$\langle \forall b ::\; \textbf{stable}\ (x \wedge b) \rangle$$

5.10 Solutions to Exercises

1. (a)

$p \wedge q$ **co** p	, given
stable $\neg q$	, given
$p \vee \neg q$ **co** $p \vee \neg q$	, disjunction of above two
p **co** $p \vee \neg q$	, strengthen lhs of above

(b)
$p \;\Rightarrow\; q$, from p **co** q
$\neg p \;\Rightarrow\; \neg q$, from $\neg p$ **co** $\neg q$
$p \;\equiv\; q$, from above two
p **co** p , from p **co** q and $p \;\equiv\; q$
$\neg p$ **co** $\neg p$, from $\neg p$ **co** $\neg q$ and $p \;\equiv\; q$
constant p , above two; definition of constant

(c)
$q \;\Rightarrow\; b$, from q **co** b
p **co** $q \vee r$, given
p **co** $b \vee r$, weaken rhs using $q \;\Rightarrow\; b$

(d)
$p \wedge \neg FP$ **co** p , given
stable $p \wedge FP$, stability at fixed point
p **co** p , disjunction of above two
stable p , rewrite the above

(e)
I **co** p , given
$I \wedge p$ **co** p , strengthen lhs
p **co** p , substitution axiom: replace I by *true* in the lhs

(f) We have to show for all m, **stable** $3 = m$.
Now,
$\langle \forall m :\; m \neq 3 :\; \textbf{stable}\ 3 = m \rangle$
, $3 = m$ is *false* for $m \neq 3$; *false* is stable
$\langle \forall m :\; m = 3 :\; \textbf{stable}\ 3 = m \rangle$
, $3 = m$ is *true* for $m = 3$; *true* is stable
$\langle \forall m ::\; \textbf{stable}\ 3 = m \rangle$
, disjunction of the above two

(g)
constant y , given
constant $x + y$, given
constant $(x + y) - y$, constant formation rule applied to the above two
constant x , from the above

2. (a)
$p \;\wedge\; x \in g$ **co** $x \in g \;\vee\; q$
, given
$\langle \forall x :\; x \in G :\; p \;\wedge\; x \in g \rangle$ **co** $\langle \forall x :\; x \in G :\; x \in g \;\vee\; q \rangle$
, conjunction over all x, $x \in G$
$p \;\wedge\; g \supseteq G$ **co** $g \supseteq G \;\vee\; q$
, simplify (p and q do not name x)

(b)
$$\begin{array}{ll} & p \ \wedge \ x \notin g \ \ \textbf{co} \ \ x \notin g \ \vee \ q \\ & \qquad\qquad \text{, given} \\ & \langle \forall x : \ x \notin G : \ p \ \wedge \ x \notin g \rangle \ \ \textbf{co} \ \ \langle \forall x : \ x \notin G : \ x \notin g \ \vee \ q \rangle \\ & \qquad\qquad \text{, conjunction over all } x,\ x \notin G \\ & p \ \wedge \ g \subseteq G \ \ \textbf{co} \ \ g \subseteq G \ \vee \ q \\ & \qquad\qquad \text{, simplify} \end{array}$$

3. Consider a program that consists of integer variables x and y and a single action

$$x, \ y := \ x + 1, \ y - 1$$

Let $p(x, y)$ and $q(x, y)$ be $x = y$ and $x = y \ \vee \ x = y + 1$, respectively. For any m and n, the following properties hold in the program:

$$\begin{array}{lll} p(x,n) \ \textbf{co} \ q(x,n), & \text{i.e.,} & x = n \ \ \textbf{co} \ \ x = n \ \vee \ x = n + 1 \\ p(m,y) \ \textbf{co} \ q(m,y), & \text{i.e.,} & m = y \ \ \textbf{co} \ \ m = y \ \vee \ m = y + 1 \end{array}$$

However, $p(x, y)$ **co** $q(x, y)$ does not hold in this program. Now, we prove that the strategy is valid assuming additionally that x and y are not changed simultaneously. That is,

$$\frac{\begin{array}{c} p(x,n) \ \ \textbf{co} \ \ q(x,n) \\ p(m,y) \ \ \textbf{co} \ \ q(m,y) \\ x, y = m, n \ \ \textbf{co} \ \ x = m \ \vee \ y = m \end{array}}{p(x,y) \ \ \textbf{co} \ \ q(x,y)}$$

The proof is as follows.

$$\begin{array}{ll} p(x,n) \ \wedge \ p(m,y) \ \wedge \ x, y = m, n \ \ \textbf{co} & \\ \quad q(x,n) \ \wedge \ q(m,y) \ \wedge \ (x = m \ \vee \ y = n) & \\ & \text{, conjunction of the premises} \\ p(x,y) \ \wedge \ x, y = m, n \ \ \textbf{co} \ \ q(x,y) & \text{, rewrite lhs and weaken rhs} \\ p(x,y) \ \ \textbf{co} \ \ q(x,y) & \text{, disjunction over all } m \text{ and } n \end{array}$$

4. The goal is to prove that for all m, **stable** $y = m$.

$$\begin{array}{ll} \langle \forall m : \ m \neq 0 : \textbf{stable} \ x = 0 \ \wedge \ y = 0 \ \wedge \ y = m \rangle & \\ & \text{, } \textit{false} \text{ is stable} \\ \textbf{stable} \ x = 0 \ \wedge \ y = 0 \ \wedge \ y = 0 & \text{, } \textbf{invariant} \ x = 0 \ \wedge \ y = 0 \\ \langle \forall m :: \ \textbf{stable} \ x = 0 \ \wedge \ y = 0 \ \wedge \ y = m \rangle & \\ & \text{, from the above two} \\ \langle \forall m :: \ \textbf{stable} \ y = m \rangle & \text{, use substitution axiom:} \\ & \quad \textbf{invariant} \ x = 0 \ \wedge \ y = 0 \end{array}$$

5. Let $p(x)$ **co** $q(x)$ denote a **co**-property in which $p(x)$ and $q(x)$ possibly mention x. Given that $x = y$ is invariant, we show $p(y)$ **co** $q(y)$.

$p(x)$ **co** $q(x)$, given
$p(x) \wedge x = y$ **co** $q(x) \wedge x = y$
, stable conjunction with $x = y$
$p(y) \wedge x = y$ **co** $q(y) \wedge x = y$
, $\langle p(x) \wedge x = y \rangle \equiv \langle p(y) \wedge x = y \rangle$;
similarly, $\langle q(x) \wedge x = y \rangle \equiv \langle q(y) \wedge x = y \rangle$
$p(y)$ **co** $q(y)$, $x = y$ is *true* using substitution axiom

6. (a) We show

$$\{x = m \wedge x \neq y\} \; x = y \rightarrow x := x + 1 \; \{x = m\}$$
$$\{x = m \wedge x \neq y\} \; x \neq y \rightarrow y := y + 1 \; \{x = m\}$$

Apply the axiom of assignment (see appendix A.4.1), and show from the program text:

$$x = m \wedge x \neq y \wedge x = y \Rightarrow x + 1 = m$$
$$x = m \wedge x \neq y \wedge x \neq y \Rightarrow x = m$$

(b) Prove this from the program text in a manner similar to exercise (6a).

(c)
$$\begin{aligned} FP &\equiv (x = y \Rightarrow x = x + 1) \wedge (x \neq y \Rightarrow y = y + 1) \\ &\equiv x \neq y \wedge x = y \\ &\equiv \mathit{false} \end{aligned}$$

Hence, the program is deadlock-free.

7. (a)
$x = m$ **co** $x \geq m$, given
$x \geq n$ **co** $\langle \exists m :: m \geq n \wedge x \geq m \rangle$
, elimination theorem
$x \geq n$ **co** $x \geq n$, simplify

Conversely,

$x \geq n$ **co** $x \geq n$, given
$x = m$ **co** $x \geq m$, rename n by m; strengthen lhs

(b)
$x \geq y$ **co** $\langle \exists m, n :: m \geq n \wedge$
$(x, y = m, n \vee (m > n \wedge x, y = m - 1, n)) \rangle$
, elimination theorem on premise
$x \geq y$ **co** $\langle \exists m, n :: x \geq y \vee x \geq y \rangle$
, simplify and weaken rhs
stable $x \geq y$, simplify rhs

(c)
$x + y = k$ **co** $\langle \exists m, n :: m + n = k \wedge$
$(x, y = m, n \vee x, y = m + 1, n - 1) \rangle$
, elimination theorem on premise
$x + y = k$ **co** $\langle \exists m, n :: x + y = k \rangle$
, weaken rhs
$x + y = k$ **co** $x + y = k$
, simplify rhs
constant $x + y$, definition of constant

(d)
$x = m$ **co** $x \geq m$, x is nondecreasing
$y = n$ **co** $y \leq n$, y is nonincreasing
$x, y = m, n$ **co** $x \geq m \ \wedge \ y \leq n$, conjunction of above two
$x \geq y$ **co** $\langle \exists m, n :: \ m \geq n \ \wedge \ x \geq m \ \wedge \ y \leq n \rangle$
, elimination theorem
$x \geq y$ **co** $x \geq y$, simplify rhs

(e) For free variable S of the same type as s we are given

stable $s \subseteq S$
$s = S$ **co** $s \subseteq S$, strengthen lhs

Let m be a free integer variable and $|s|$ denote the size of s.

$|s| \leq m$ **co** $\langle \exists S :: \ |S| \leq m \ \wedge \ s \subseteq S \rangle$
, elimination theorem
stable $|s| \leq m$, simplify

(f)
$x = m$ **co** p , given
p **co** $\langle \exists m :: \ p[x := \ m] \ \wedge \ p \rangle$, elimination theorem
p **co** $\langle \exists m :: \ p \rangle$, weaken rhs
p **co** p , p does not name m

8. (a)
$f.x \ \sim \ n$
co {elimination theorem}
$\langle \exists m :: \ f.m \ \sim \ n \ \wedge \ f.x \ \sim \ f.m \rangle$
$\Rightarrow$ {$\sim$ is transitive}
$\langle \exists m :: \ f.x \ \sim \ n \rangle$
$\Rightarrow$ {simplify}
$f.x \ \sim \ n$

(b) Use the result of exercise (8a) with "=" in the place of "$\sim$"; note that "=" is transitive. Therefore, we conclude, for any n,

stable $f.x = n$ or
constant $f.x$.

(c)
$f.x = n$
co {elimination theorem}
$\langle \exists m :: \ f.m = n \ \wedge \ x \ \sim \ m \rangle$
$\Rightarrow$ {f preserves $\sim$}
$\langle \exists m :: \ f.m = n \ \wedge \ f.x \ \sim \ f.m \rangle$
$\Rightarrow$ {weaken}
$\langle \exists m :: \ f.x \ \sim \ n \rangle$
$\equiv$ {simplify}
$f.x \ \sim \ n$

9. Consider the steps in the original proof of L' **co** R' from L **co** R. Each proof step asserts a property of the form u **co** v in one of the following ways: (1) u **co** v is a premise (a part of L **co** R), (2) u **co** v is either

false **co** v or u **co** *true*, or (3) u **co** v is deduced from two previous properties by either the conjunction or disjunction rule. Show that in each case lhs strengthening of the antecedent and consequent by the same predicate p is valid; similarly for rhs weakening by q.

This result can be used to generalize the elimination theorem. From $x = m$ **co** q, where m is free, and the appropriate restrictions on p, use the elimination theorem to deduce

$$p \ \textbf{co} \ \langle \exists m :: \ p[x := \ m] \ \wedge \ q \rangle.$$

Then strengthen all lhs with r and weaken rhs with r'.

10. (a) **constant** $(p \ \equiv \ q)$

(b) The following property describes the change in sx. For all integers m and n

$$x, sx = m, n \ \ \textbf{co} \ \ x, sx = m, n \ \vee \ (x \neq m \ \wedge \ sx = n + x)$$

Weaken the rhs of the above to get

$$x, sx = m, n \ \ \textbf{co} \ \ x, sx = m, n \ \vee \ sx = n + x$$

Now we show

• **stable** $sx \geq x$:

$$\begin{array}{ll} & sx \geq x \\ \equiv & \{\text{substitution axiom with } \textbf{invariant } x \geq 0\} \\ & sx \geq x \ \wedge \ x \geq 0 \\ \textbf{co} & \{\text{elimination theorem on the last } \textbf{co}\text{-property}\} \\ & \langle \exists m, n :: \\ & \ n \geq m \ \wedge \ m \geq 0 \ \wedge \ (x, sx = m, n \ \vee \ sx = n + x) \rangle \\ \Rightarrow & \{\text{weaken}\} \\ & sx \geq x \end{array}$$

(c) Let m, n, and r be free integer variables. Define integer variable c as follows.

$$\begin{array}{l} x, y, c = m, n, r \ \ \textbf{co} \ \ x, y, c = m, n, r \ \vee \\ \langle \neg(x, y = m, n) \ \wedge \ \langle (x > y \ \wedge \ c = r + 1) \ \vee \ (x \leq y \ \wedge \ c = r) \rangle \rangle \end{array}$$

Assuming that $x \leq y$ is invariant, we show that $c = r$ is stable, for any r.

• Proof of **stable** $c = r$:

$$\begin{array}{lll} x, y, c = m, n, r \ \ \textbf{co} \ \ x, y, c = m, n, r \ \vee \ x > y \ \vee \ c = r & & \text{, weaken rhs of the premise} \\ x, y, c = m, n, r \ \ \textbf{co} \ \ x > y \ \vee \ c = r & & \text{, simplify rhs} \\ x, y, c = m, n, r \ \ \textbf{co} \ \ c = r & & \text{, use } \textbf{invariant } x \leq y \text{ in rhs} \\ c = r \ \ \textbf{co} \ \ c = r & & \text{, disjunction over all } m \text{ and } n \end{array}$$

(d) Using a free variable m,

$$x > 5 \ \wedge \ x = m \ \ \textbf{co} \ \ x \geq m$$

A better formulation is

$$\textbf{stable} \ x > 5 \ \wedge \ x \geq m$$

(e) For vertex i, let S_i be the set of its neighbors. Let w_i denote that vertex i is black. Then the formalization is the same as in the case of mutual waiting in a knot (section 5.5.7) and the same proof applies.

11. Let c_i denote the set of particles in chamber i, $0 \leq i \leq N$. Let a_j, b_j be the sets of particles in the chambers at or above j and at or below j, respectively. That is, for any j, $0 \leq j \leq N$,

$$a_j = \langle \cup \ i : \ j \leq i \leq N : \ c_i \rangle$$
$$b_j = \langle \cup \ i : \ 0 \leq i \leq j : \ c_i \rangle$$

The law of particle movement within the vessel is given by, for any particle x and i, $0 \leq i \leq N$,

$$x \in c_i \ \ \textbf{co} \ \ x \in a_i \text{ and}$$
$$x \notin b_i \ \ \textbf{co} \ \ x \notin c_i \ .$$

We have to show, for all x and j, $0 \leq j \leq N$,

$$\textbf{stable} \ x \in a_j \tag{1}$$
$$\textbf{stable} \ x \notin b_j \ . \tag{2}$$

- Proof of (1), **stable** $x \in a_j$:

$$\begin{array}{ll}
x \in c_i \ \ \textbf{co} \ \ x \in a_i & \text{, the law of particle movement} \\
\langle \exists i : \ j \leq i \leq N : \ x \in c_i \rangle \ \ \textbf{co} \ \ \langle \exists i : \ j \leq i \leq N : \ x \in a_i \rangle & \\
& \text{, disjunction over all } i,\ j \leq i \leq N \\
x \in \langle \cup \ i : \ j \leq i \leq N : \ c_i \rangle \ \ \textbf{co} \ \ x \in \langle \cup \ i : \ j \leq i \leq N : \ a_i \rangle & \\
& \text{, rewrite} \\
x \in a_j \ \ \textbf{co} \ \ x \in a_j & \text{, use the definition of } a_j
\end{array}$$

- Proof of (2), **stable** $x \notin b_j$:

$$\begin{array}{ll}
x \notin b_i \ \ \textbf{co} \ \ x \notin c_i & \text{, the law of particle movement} \\
\langle \forall i : \ 0 \leq i \leq j : \ x \notin b_i \rangle \ \ \textbf{co} \ \ \langle \forall i : \ 0 \leq i \leq j : \ x \notin c_i \rangle & \\
& \text{, conjunction over all } i,\ 0 \leq i \leq j \\
x \notin \langle \cup \ i : \ 0 \leq i \leq j : \ b_i \rangle \ \ \textbf{co} \ \ x \notin \langle \cup \ i : \ 0 \leq i \leq j : \ c_i \rangle & \\
& \text{, rewrite} \\
x \notin b_j \ \ \textbf{co} \ \ x \notin b_j & \text{, use the definition of } b_j
\end{array}$$

12. First, express that p is a pre-condition for any change in x, using free variable m.

$$x = m \ \wedge \ \neg p \ \ \textbf{co} \ \ x = m$$

Next express that q is established as a post-condition whenever x is changed.

$$x = m \ \ \textbf{co} \ \ x = m \ \vee \ q$$

(a) Given $x: \ p \ \rightarrow \ q$ we have

$$x = m \ \wedge \ \neg p \ \ \textbf{co} \ \ x = m$$
$$x = m \ \ \textbf{co} \ \ x = m \ \vee \ q$$

Strengthen the lhs of the first property by $\neg p'$ and weaken the rhs of the second property by q' to obtain

$$x = m \ \wedge \ \neg p \ \wedge \ \neg p' \ \ \textbf{co} \ \ x = m$$
$$x = m \ \ \textbf{co} \ \ x = m \ \vee \ q \ \vee \ q', \text{ i.e.,}$$
$$x: \ p \vee p' \ \rightarrow \ q \vee q'$$

(b) We have

$$x = m \ \wedge \ \neg p \ \ \textbf{co} \ \ x = m \qquad (1)$$
$$x = m \ \ \textbf{co} \ \ x = m \ \vee \ q \qquad (2)$$
$$x = m \ \wedge \ \neg p' \ \ \textbf{co} \ \ x = m \qquad (3)$$
$$x = m \ \ \textbf{co} \ \ x = m \ \vee \ q' \qquad (4)$$

$$x = m \ \wedge \ (\neg p \vee \neg p') \ \ \textbf{co} \ \ x = m \quad \text{, disjunction of (1,3)}$$
$$x = m \ \ \textbf{co} \ \ x = m \ \vee \ (q \wedge q') \quad \text{, conjunction of (2,4)}$$
$$x: \ p \wedge p' \ \rightarrow \ q \wedge q' \quad \text{, definition}$$

13. Only the actions of the programs are shown.

(a) $x := \ x$ {this is also known as the *skip* action}

(b) $x = 0 \ \rightarrow \ x := \ x + 1$

(c) $x > 0 \ \rightarrow \ x := \ x + 1$

(d) $x \notin \ \{2, 4, 8, 64, -2, -8\} \ \rightarrow \ x := \ x + 1$.
This is because x is a root of 64 iff $x = 2, 4, 8, 64, -2$, or -8.

14. The state transition diagram is shown in Fig. 5.5. The **co**-properties are obtained directly from the diagram.

$$\neg p \wedge \neg q \ \ \textbf{co} \ \ \neg q \quad \{(\neg p \wedge \neg q) \ \vee \ (p \wedge \neg q) \ \equiv \ \neg q\}$$
$$p \wedge \neg q \ \ \textbf{co} \ \ p$$
$$p \wedge q \ \ \textbf{co} \ \ q$$
$$\textbf{stable} \ \neg p \wedge q$$

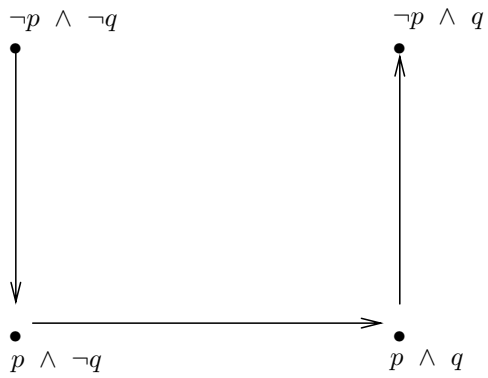


Figure 5.5: State transition diagram for exercise 15

By taking suitable disjunctions, we obtain

$$\begin{array}{rl} p & \textbf{co} \ \ p \vee q \\ \neg p & \textbf{co} \ \ \neg p \vee \neg q \\ q & \textbf{co} \ \ q \\ \neg q & \textbf{co} \ \ p \vee \neg q \end{array}$$

To see that the second group of properties is equivalent to the first, take suitable conjunctions of the properties in the second group to obtain the first group.

15. For a ring network, we use the following notation. For any process i,

i' = the identity of the process to which i sends messages
$q.i = i$ is idle
$s.i$ = the number of messages sent by i
$r.i$ = the number of messages received by i

Analogous to (CN1–CN4), we have for all i, and free variables m, n

$$s.i \geq r.i' \geq 0 \qquad \text{(CN}'1\text{)}$$
$$\textbf{stable} \ s.i \geq m, \ \textbf{stable} \ r.i \geq n \qquad \text{(CN}'2\text{)}$$
$$q.i \ \wedge \ r.i = m \ \ \textbf{co} \ \ r.i = m \ \Rightarrow \ q.i \qquad \text{(CN}'3\text{)}$$
$$q.i \ \wedge \ s.i = n \ \ \textbf{co} \ \ s.i = n \qquad \text{(CN}'4\text{)}$$

The property analogous to (CN5) can be written using a set of free variables, $m.i$, one for each process i:

$$\textbf{stable} \ \langle \forall i :: \ q.i \ \wedge \ r.i' = s.i \rangle \ \wedge \ \langle \forall i :: \ r.i = m.i \rangle \qquad \text{(CN}'5\text{)}$$

• Proof of (CN′5): The proof is similar to the proof of (CN5). Conjoin (CN′3, CN′4)

$q.i \ \wedge \ r.i, s.i = m, n \ \ \textbf{co} \ \ (r.i = m \ \Rightarrow \ q.i) \ \wedge \ s.i = n$

Replace m by $m.i$ and n by $m.i'$

$q.i \ \wedge \ r.i, s.i = m.i, m.i' \ \ \textbf{co} \ \ (r.i = m.i \ \Rightarrow \ q.i) \ \wedge \ s.i = m.i'$

Take conjunction over all i

$\langle \forall i :: \ q.i \ \wedge \ r.i, s.i = m.i, m.i' \rangle$ **co**
$\quad \langle \forall i :: \ (r.i = m.i \ \Rightarrow \ q.i) \ \wedge \ s.i = m.i' \rangle$

Since $r.i \geq m.i$ is stable —from (CN′2)— its conjunction over all i is stable. Conjoin $\langle \forall i :: \ r.i \geq m.i \rangle$ to both sides

$\langle \forall i :: \ q.i \ \wedge \ r.i, s.i = m.i, m.i' \rangle$ **co**
$\quad \langle \forall i :: \ (r.i = m.i \ \Rightarrow \ q.i) \ \wedge \ s.i = m.i' \rangle \ \wedge \ \langle \forall i :: \ r.i \geq m.i \rangle$

In the rhs using $s.i = m.i', r.i' \geq m.i'$ and $s.i \geq r.i'$ (from CN′1) we get $r.i' = m.i'$. That is, $\langle \forall i :: \ r.i = m.i \rangle$. Simplify the rhs. □

A similar treatment, with a more general notation, is used for general networks. Let z be the set of all nodes, iz the set of outgoing channels from i, zi the set of incoming channels to i, and zz the set of all channels. Let m and n be free variables, L and N be set of free variables, c a channel, and i a process. We are given

$s.c \geq r.c \geq 0$ (CN″1)
stable $s.c \geq m$, **stable** $r.c \geq n$ (CN″2)
$q.i \ \wedge \ (r = L).zi \ \ \textbf{co} \ \ (r = L).zi \ \Rightarrow \ q.i$ (CN″3)
$q.i \ \wedge \ (s = N).iz \ \ \textbf{co} \ \ (s = N).iz$ (CN″4)

Given $q.z \ \equiv \ \langle \forall i :: \ q.i \rangle$, we have to show

stable $q.z \ \wedge \ (s = r).zz \ \wedge \ (r = L).zz$ (CN″5)

The proof is similar to the one for ring network. Note that $(s = r).xy$ and $(s \geq r).xy$ are both *true* if x or y is empty. Also,

$\quad \langle (s = r).(x \cup x')(y \cup y') \rangle$
$\equiv \ \langle (s = r).xy \ \wedge \ (s = r).xy' \ \wedge \ (s = r).x'y \ \wedge \ (s = r).x'y' \rangle$

Similarly, expand $(s \geq r).(x \cup x')(y \cup y')$.

16. We give a construction for ss similar to that for the strongest invariant (section 5.6.2). Consider the infinite sequence of properties, where $q_0 \equiv p$ and q_{i+1} is the strongest rhs for q_i.

$q_i \ \ \textbf{co} \ \ q_{i+1}$, for $i \geq 0$.

Let $ss.p \equiv \langle \exists\ i ::\ q_i \rangle$. Then, $p\ \Rightarrow\ ss.p$, because $q_0\ \Rightarrow\ \langle \exists\ i ::\ q_i \rangle$. Also, $ss.p$ is stable, according to the lemma in section 5.6.2.

Finally, the proof that $ss.p$ is the strongest stable predicate weaker than p is along the same lines as for the proof of the strongest invariant. (In fact, $ss.p$ is the strongest invariant when p is the initial condition.)

(a) $(p\ \Rightarrow\ q)\ \Rightarrow\ (ss.p\ \Rightarrow\ ss.q)$

(a1) $p\ \Rightarrow\ ss.p\ \wedge\ ss.q$:

$p\ \Rightarrow\ q$	, antecedent
$q\ \Rightarrow\ ss.q$	, $ss.q$ is weaker than q
$p\ \Rightarrow\ ss.q$	, above two
$p\ \Rightarrow\ ss.p$	, $ss.p$ is weaker than p
$p\ \Rightarrow\ ss.p\ \wedge\ ss.q$	, above two

(a2) **stable** $ss.p\ \wedge\ ss.q$:

stable $ss.p$	, property of ss
stable $ss.q$	, similarly
stable $ss.p\ \wedge\ ss.q$	, conjunction of stable predicates

From $(a1)$ and $(a2)$, $ss.p \wedge ss.q$ is a stable predicate, and p implies this predicate. Since $ss.p$ is the strongest stable predicate that p implies

$ss.p\ \Rightarrow\ (ss.p\ \wedge\ ss.q)$, i.e.,
$ss.p\ \Rightarrow\ ss.q$

(b) **stable** $p\ \equiv\ (ss.p \equiv p)$:

First, we show **stable** $p \Rightarrow\ (ss.p \equiv p)$.

$p \Rightarrow p$	, predicate calculus
stable p	, premise
$ss.p\ \Rightarrow\ p$	, above two and definition of $ss.p$
$p\ \Rightarrow\ ss.p$	, $ss.p$ is weaker than p
$p\ \equiv\ ss.p$	, above two

Next, we show $(ss.p \equiv p)\ \Rightarrow\ (\textbf{stable}\ p)$.

$ss.p \equiv p$	, assume
stable $ss.p$	, definition of $ss.p$
stable p	, from above two

(c) $ss.(ss.p) \equiv ss.p$

stable $ss.p$	, fact about $ss.p$
$ss.(ss.p) \equiv ss.p$	, substitute $ss.p$ for p in (16b)

(d) $ss.\langle \exists\ i ::\ p_i \rangle\ \equiv\ \langle \exists\ i ::\ ss.p_i \rangle$:

First, we prove $ss.\langle \exists\ i ::\ p_i \rangle\ \Rightarrow\ \langle \exists\ i ::\ ss.p_i \rangle$.

$\langle \forall\ i ::\ p_i\ \Rightarrow\ ss.p_i \rangle$, definition of ss
$\langle \exists\ i ::\ p_i \rangle\ \Rightarrow\ \langle \exists\ i ::\ ss.p_i \rangle$
, predicate calculus (1)
$\langle \forall i ::\ \textbf{stable}\ ss.p_i \rangle$, definition of ss
stable $\langle \exists\ i ::\ ss.p_i \rangle$, disjunction on above
$ss.\langle \exists\ i ::\ p_i \rangle\ \Rightarrow\ \langle \exists\ i ::\ ss.p_i \rangle$
, from (1), above, definition of ss

Next, we show $\langle \exists\ i ::\ ss.p_i \rangle\ \Rightarrow\ ss.\langle \exists\ i ::\ p_i \rangle$.

$p_i\ \Rightarrow\ \langle \exists\ j ::\ p_j \rangle$, predicate calculus
$ss.p_i\ \Rightarrow\ ss.\langle \exists\ j ::\ p_j \rangle$, monotonicity of ss
$\langle \exists\ i ::\ ss.p_i \rangle\ \Rightarrow\ ss.\langle \exists\ i ::\ p_i \rangle$
, disjunction over all i

17. First we show that FP is a solution to the given equation. For arbitrary b

$\langle \forall\ p :$
$\quad \langle \forall q ::\ \textbf{stable}\ p \wedge q \rangle :\ \textbf{stable}\ p \wedge b$
$\rangle$, tautology
stable $\langle \exists p :\ \langle \forall q ::\ \textbf{stable}\ p \wedge q \rangle :\ p \wedge b \rangle$
, disjunction of stables
stable $(\langle \exists p :\ \langle \forall q ::\ \textbf{stable}\ p \wedge q \rangle :\ p \rangle\ \wedge\ b)$
, rewrite
stable $(FP \wedge b)$, use the definition of FP
$\langle \forall b ::\ \textbf{stable}\ (FP \wedge b) \rangle$, bind the free variable b

FP is the weakest solution to $\langle \forall b ::\ \textbf{stable}\ (x \wedge b) \rangle$, because any solution to this equation implies FP.

6
Progress Properties

6.1 Introduction

Safety properties, discussed in chapter 5, allow us to state that "the program does no harm". A trivial program that causes no state change —a program that consists only of a *skip* action, for instance— satisfies all the safety properties. Thus, safety properties alone are insufficient as a basis of program design. Several formal aspects of program design and refinement are seriously affected by the absence of a requirement that the program must guarantee some desirable state changes.

In this chapter, we study a class of properties known as *progress*, or *liveness*. According to Lamport [113], "A liveness property is one which states that something *must* happen". For instance, "I press the switch and then the light is on" is a progress property. A safety property for this system might be "the light never comes on unless the switch is pressed". This safety property is conveniently implemented by smashing the light bulb. Conversely, the given progress property might be implemented by having a light that is permanently on. It is the interplay between the safety and progress properties that determines a nontrivial design.

A progress property may be regarded as a performance guarantee. Performance guarantees typically include numerical time bounds: the light comes on within 10 ms of pressing the switch or a car traveling at 90 kmph stops within 40 meters after the brakes are jammed. Numerical performance guarantees, though desirable, are hard to implement, because such guarantees depend on the speed of the underlying machine or the network,

the scheduling strategy, or even the load on the system —factors that are outside our control during program design. A useful abstraction employed in complexity theory is to specify the rate of growth of the computation time as a function of the input size. This abstraction ignores speed-ups by constant factors. An even coarser abstraction is to classify the rate of growth as being polynomial or nonpolynomial. Unfortunately, we don't yet have a theory to provide such performance guarantees for the asynchronous systems that we consider here. So we abstract further by eliminating the notion of absolute time. We state and prove properties of the following form: once predicate p holds, eventually q will hold in the system. For the lighting problem, p might be "the switch has been pressed" and q might be "the light is on". The time duration between the occurrences of p and q is left unspecified. We develop a logic to state and verify such properties. Even though the logic cannot specify numerical performance measures, it constitutes a useful first step; once we have proved such a property, we may attempt to deduce the performance measures empirically or by using analytic modeling.

Overview of the chapter

In section 6.2, we describe several notions of fairness that are essential for studying progress properties in asynchronous systems. In section 6.3, we introduce *transient* predicates; a transient predicate is guaranteed to be falsified eventually under the given fairness assumption. The primary progress operator, *leads-to*, is introduced in section 6.4. Most progress specifications and deductions are done with *leads-to*. So we give a variety of manipulation rules and show several examples of *leads-to* in section 6.5. Certain theoretical issues are taken up in section 6.6.

Note on the binding powers of operators We introduce logical operators **transient**, **ensures** (abbreviated to **en**), and $\mapsto$ (pronounce *leads-to*) in this chapter. Each of these operators has lower binding power than all arithmetic and predicate calculus operators. Thus,

$p \wedge q$ **en** $r \wedge s$ is to be interpreted as
$(p \wedge q)$ **en** $(r \wedge s)$

A property includes one of these operators or **co**; hence, there is no need to specify priorities among these operators. □

6.2 Fairness

The need for fairness and various flavors of it can be explained by considering the program shown next. The program is a single box that has three actions. Variables x, y, and z are integers.

```
box Fairness
      α ::                  x :=  x + 1
   [] β ::                  y :=  y + 1
   [] γ :: x ≠ y  →         z :=  z + 1
end {Fairness}
```

A fairness condition constrains the order in which the actions, α, β, and γ, are executed. We study the following kinds of fairness: minimal progress, weak fairness, and strong fairness.

6.2.1 Minimal progress

Under minimal progress, the following step is repeated until all guards are *false* : an arbitrary non-*skip* action whose guard is *true* in the current state is executed.

We assert for box *Fairness* that $x + y + z$ will increase eventually under minimal progress (so it will increase without bound). This is because all guards are never *false* —α and β have *true* as their guards— and execution of any action increases $x+y+z$. However, neither x, y, nor z is guaranteed to increase because, for instance, β might be executed indefinitely, thus preserving the values of x and z. Similarly, no guarantee can be made that $x + y$ will increase (γ might be executed forever once $x \neq y$); there is also no guarantee about eventual increase of $x + z$ or $y + z$.

6.2.2 Weak fairness

Under weak fairness, each action is executed infinitely often in any execution. (Executing an action in a state where its guard is *false* causes no state change.)

This fairness condition guarantees that different processes in a multiprocess program will be individually allowed to proceed. The actions representing the various processes constitute the program under consideration. For example, α might belong to one process, and β and γ to another. If α is chosen forever for execution (as in minimal progress), we have effectively blocked the second process permanently; weak fairness prevents such executions.

For the example program, *Fairness*, we assert that x (and y) will increase without bound because each execution of α (or β) will cause x (or y) to increase. We cannot assert that z will increase. For instance, consider the following execution starting in state $x, y = 0, 0$: execute α, β, and γ in this order and repeat the sequence forever. Whenever γ is executed $x = y$, so z is never increased.

Weak fairness is sometimes expressed as follows: if the guard of an action remains continuously *true*, then the action is eventually executed effectively (i.e., in a state where the guard is *true*). This formulation is identical to the given formulation.

6.2.3 Strong fairness

Execution of an action is strongly fair in an execution if the guard of the action is *true* infinitely often, then the action is executed (in a state where the guard is *true*) infinitely often. Execution of an action is strongly fair if it is strongly fair in all executions.

For program *Fairness*, if all three actions are executed in a strongly fair manner, x, y, and z all increase indefinitely. It is easy to see this result for x and y; for z, note that $x \neq y$ is *true* infinitely often since x and y are incremented asynchronously (see exercise 13); therefore, z is incremented infinitely often.

6.2.4 Which is the fairest one?

A traditional sequential program permits no choice in action executions. Therefore, the only pertinent notion of fairness in this case is minimal progress. The significant progress property is termination; it can be stated as "starting in any state that satisfies the initial condition, eventually a state is reached that satisfies *FP*" (*FP* is the fixed point predicate; a state that satisfies *FP* is a terminal state). Termination can be proved by displaying a function whose value decreases eventually as a result of program execution; if there is a lower bound on the function value (specifically, if the function assumes values from a well-founded set), its value cannot decrease forever, so termination is guaranteed. Minimal progress is also useful in concurrent programs for proving "absence of deadlock"; a typical example is if there is a *hungry* philosopher (in a dining philosophers problem), some philosopher will eat.

Minimal progress is not sufficient to guarantee "absence of individual starvation". Even though some philosopher may eat, and eating is performed infinitely often, a particular philosopher may stay *hungry* forever (and starve). In program *Fairness* studied in this section, the system as a whole makes progress by increasing $x + y + z$, but no guarantees can be made about the individual variables.

Weak fairness meets most of the criteria for a useful notion of fairness. It is powerful enough so that starvation-free solutions can be designed, and it is simple enough that a reasonably effective theory for reasoning about it can be developed. Additionally, it has nice compositional properties; see section 8.2.3.

A typical example of the application of strong fairness is in implementing a strong semaphore. It is required that if the semaphore value exceeds zero

infinitely often, every process that is waiting for the semaphore will be granted the semaphore. Under the weak fairness requirement, a waiting process may never be granted the semaphore (because its guard —that the semaphore value exceeds zero— is not continuously *true*, but it is *true* infinitely often.) We take up this example in section 6.5.6.

We may mix different kinds of fairness for the same program; we may group the actions and require different forms of fairness for each group. For instance, if we require minimal progress for $\{\alpha, \beta\}$ and strong fairness for $\{\gamma\}$ in program *Fairness*, then $x+y$ and z increase indefinitely, but neither x nor y can be guaranteed to increase. If $\{\alpha, \beta\}$ have the weak fairness restriction and $\{\gamma\}$ the strong fairness, then all three —x, y, and z— will grow indefinitely.

We can introduce a variety of fairness requirements by attaching different fairness conditions to subgroups within a group. Though these possibilities are theoretically interesting, we do not pursue them in this book. In fact, we develop the theory only for minimal progress and weak fairness and partially for strong fairness.

6.3 Transient Predicate

A predicate is *transient* if it is guaranteed to be falsified by execution of a single action. The formal definition depends on the form of fairness assumed for program execution. This is the only point in our theory where the definition of an operator depends on the form of fairness. Other progress operators are defined using transient predicates; their definitions and derived rules are independent of the underlying fairness. Thus, the progress proofs are largely shielded from having to argue about specific fairness properties of programs.

Recall the following two properties of action systems:

1. Action *skip* is included in every program; hence, each program has at least one action. (We do not show *skip* explicitly in the programs.)

2. Execution of each action terminates. It is easy to check terminations for simple actions, such as the ones represented by assignment statements. For more intricate actions, which include conditional and looping constructs, we have to use the methods of sequential program verification to prove termination; see Gries [79, chapter 12].

For a terminating action s, we have the law of the excluded miracle [61]

$$\frac{\{p\} \quad s \quad \{false\}}{\neg p}$$

i.e., the post-condition of an action is *false* only if the pre-condition is *false*. Using the substitution axiom, this law can be interpreted as "the

resulting state of an action is unreachable only if the action is started in an unreachable state".

6.3.1 Minimal progress

Consider a program in which action i is of the form $g_i \rightarrow s_i$. Predicate p is *transient* if both of the following conditions hold:

1. Whenever p holds, some action has a *true* guard:
   $p \Rightarrow \langle \exists\, i ::\ g_i \rangle$.

2. Executing *any* non-*skip* action that has a *true* guard in a state where p holds falsifies p: (below, i is quantified over non-*skip* actions)
   $\langle \forall\, i ::\ \{p \wedge g_i\}\ \ s_i\ \ \{\neg p\} \rangle$.

Thus, a transient predicate is falsified by execution of any non-*skip* action. This definition may seem overly restrictive; can we not require that a predicate be falsified only after execution of a finite *sequence* of actions? Operator *leads-to* is used to express such facts. Requirement (2), that every non-*skip* action with a *true* guard falsify p, is essential; without such a requirement, a possible execution may consist only of actions that never falsify p.

Example:

We consider the running example from section 6.2, which we reproduce below.

---

box *Fairness*
$\quad \alpha ::\ \ x := x + 1$
$[\!]\ \beta ::\ \ y := y + 1$
$[\!]\ \gamma :: x \neq y \rightarrow\ \ z := z + 1$
end {*Fairness*}

First, we establish that for any integer k,

transient $x + y + z = k$

Following are the proof obligations, and they are easily proved. For any integer k,

1. $x + y + z = k \ \Rightarrow\ true$

2. $\{x + y + z = k\}\ \ x := x + 1\ \ \{x + y + z \neq k\}$
   $\{x + y + z = k\}\ \ y := y + 1\ \ \{x + y + z \neq k\}$
   $\{x + y + z = k \ \wedge\ x \neq y\}\ \ z := z + 1\ \ \{x + y + z \neq k\}$

We can also show that $x = y$ is transient. The proof obligations are similarly obtained. Corresponding to action γ, we have to show

$$\{x = y \ \wedge \ x \neq y\} \ \ z := \ z + 1 \ \ \{x \neq y\}$$

which follows from $\{false\} \ \ s \ \ \{q\}$ for any action s and predicate q.

Next, we attempt to prove that for any k, **transient** $x = k$. We know from operational arguments that this property does not hold. The proof obligations are

1. $x = k \ \Rightarrow \ true$

2. $\{x = k\} \ x := \ x + 1 \ \ \{x \neq k\}$
   $\{x = k\} \ \ y := \ y + 1 \ \ \{x \neq k\}$
   $\{x = k \ \wedge \ x \neq y\} \ \ z := \ z + 1 \ \ \{x \neq k\}$

The last two assertions cannot be established.

Finally, we leave it to the reader to show that neither $x + y = k$ nor $x + z = k$ can be shown transient.

Remark To prove that $x + y + z$ increases eventually, it is sufficient to show: (1) $x + y + z$ is nondecreasing (a safety property that can be established from the program text), and (2) $x + y + z = k$ is transient, for any k. □

6.3.2 Weak fairness

A transient predicate is falsified by *every* "enabled" action under minimal progress. However, under weak fairness it is sufficient to have a *single* action falsify the predicate. Define

$$\textbf{transient} \ p \ \equiv \ \langle \exists \ t :: \ \{p\} \ \ t \ \ \{\neg p\} \rangle$$

where t is over all actions in the system. If t is of the form $g \ \rightarrow \ s$, then $\{p\} \ \ t \ \ \{\neg p\}$ is shown by[1]

$$p \ \Rightarrow \ g \ \text{ and } \ \{p\} \ \ s \ \ \{\neg p\}$$

The following operational argument shows that eventually $\neg p$ holds given that p is transient. Let t be an action that falsifies p. From the weak fairness condition, t is executed eventually. If $\neg p$ holds immediately prior to the execution of t, the desired result has been shown. Otherwise, p holds prior to the execution of t, and from $\{p\} \ \ t \ \ \{\neg p\}$, $\neg p$ holds following the execution of t. (Note that the execution of t terminates.)

---

[1] See appendix A.4.1.

Example:

We consider box *Fairness* of section 6.2. The following predicates can be shown to be transient under weak fairness. For any integer k,

$$x = k \ , \ y = k \ , \ x + y = k \ , \ y + z = k \ , \ x + z = k \ , \ x + y + z = k$$

Predicate $z = k$ cannot be shown transient, because we cannot display an action t such that $\{z = k\} \ \ t \ \ \{z \neq k\}$ holds. The only action that modifies z is

$$\gamma :: \ x \neq y \ \rightarrow \ z := \ z + 1$$

and this action does not satisfy $\{z = k\} \ \ \gamma \ \ \{z \neq k\}$.

We leave it to the reader to show that $x \leq k$ cannot be proved to be transient.

6.3.3 Strong fairness

Transient predicates for strong fairness can be defined recursively using *leads-to*; see [102] for details. We do not plan to consider strong fairness in any detail in this book. However, we show in section 6.5.7 that progress properties under strong fairness can be proved using only the concepts developed for weak fairness. The idea is to show that auxiliary variables can encode the *eventual* operator of temporal logic, and we specify strong fairness requirement by adding axioms that employ these auxiliary variables.

6.3.4 Comparing minimal progress and weak fairness

It is not hard to show that any predicate that is transient under weak fairness is transient under strong fairness. A similar result does *not* hold for minimal progress and weak fairness. To see this, consider a program that consists of the following actions; here, b and t are booleans, and their initial values are immaterial.

$$\begin{array}{|ll} \alpha :: b & \rightarrow \ t := \ \mathit{false} \\ [\!] \ \beta :: \neg b & \rightarrow \ t := \ \mathit{false} \end{array}$$

Under minimal progress t is transient. However, it cannot be shown that t is transient under weak fairness (because there is no action γ such that $\{t\} \ \ \gamma \ \ \{\neg t\}$ holds).

We leave it to the reader to show that a predicate may be transient under weak fairness but not under minimal progress.

6.3.5 Derived rules

We show two derived rules about transient predicates that hold under either minimal progress or weak fairness. These rules are used primarily in proving the derived rules for *leads-to*, not for establishing properties of programs.

- The only predicate that is both stable and transient is *false*.

  $$(\textbf{stable}\ p\ \wedge\ \textbf{transient}\ p)\ \equiv\ \neg p$$

- (strengthening) $\dfrac{\textbf{transient}\ p}{\textbf{transient}\ (p \wedge q)}$

Now we prove the first rule for both forms of fairness. It is easy to see from the definition of transient predicate that *false* is transient, and it is known that *false* is stable. The remaining proof obligation for the first rule is:

$$(\textbf{stable}\ p\ \wedge\ \textbf{transient}\ p)\ \Rightarrow\ \neg p\ .$$

• Proof of $(\textbf{stable}\ p \wedge \textbf{transient}\ p)\ \Rightarrow\ \neg p$ (under minimal progress)

For any action $g_i\ \rightarrow\ s_i$:

$\{p\ \wedge\ g_i\}\ \ s_i\ \ \{p\}$	, **stable** p
$\{p\ \wedge\ g_i\}\ \ s_i\ \ \{\neg p\}$	, **transient** p
$\{p\ \wedge\ g_i\}\ \ s_i\ \ \{\mathit{false}\}$	, conjunction of the above two
$\neg(p\ \wedge\ g_i)$	, from law of the excluded miracle
$p\ \Rightarrow\ \neg g_i$	, simplify the above
$p\ \Rightarrow\ \langle\forall\ i ::\ \neg g_i\rangle$	, conjoin over all i
$p\ \Rightarrow\ \langle\exists\ i ::\ g_i\rangle$	, definition of **transient** p
$\neg p$	, conjoin the above two □

• Proof of $(\textbf{stable}\ p\ \wedge\ \textbf{transient}\ p)\ \Rightarrow\ \neg p$ (under weak fairness)

From the definition of **transient** p, there is an action t such that

$\{p\}\ \ t\ \ \{\neg p\}$	, **transient** p
$\{p\}\ \ t\ \ \{p\}$	, **stable** p
$\{p\}\ \ t\ \ \{\mathit{false}\}$	, conjunction of the above two
$\neg p$	, law of the excluded miracle □

• Proof of the strengthening rule (under minimal progress)

$p\ \Rightarrow\ \langle\exists\ i ::\ g_i\rangle$	, **transient** p
$p \wedge q\ \Rightarrow\ \langle\exists\ i ::\ g_i\rangle$	, predicate calculus

Also, for an action with guard g_i and body s_i

$$\begin{array}{lll} \{p \ \wedge \ g_i\} \ \ s_i \ \ \{\neg p\} & , \textbf{transient } p \\ \{p \ \wedge \ q \ \wedge \ g_i\} \ \ s_i \ \ \{\neg p \ \vee \ \neg q\} & , \text{strengthen lhs, weaken rhs} \end{array}$$

Hence, **transient** $(p \wedge q)$. □

• Proof of the strengthening rule (under weak fairness)

There is an action t such that

$$\begin{array}{lll} \{p\} \ \ t \ \ \{\neg p\} & , \textbf{transient } p \\ \{p \wedge q\} \ \ t \ \ \{\neg p \vee \neg q\} & , \text{strengthen lhs, weaken rhs} \\ \textbf{transient } (p \wedge q) & , \text{definition of transient} \end{array}$$ □

6.3.6 Discussion

The notion of stability (and its generalization in **co**) is fundamental for developing the theory of safety; a stable predicate is guaranteed never to be falsified. The notion of transient, fundamental to a theory of progress, is almost the opposite of stability; a transient predicate is guaranteed to be falsified. A predicate can be established stable by proving facts about *all* actions in a program. Under weak fairness, a predicate can be established transient by proving a fact about *some* action.

It is an interesting research question to identify problem areas where stability and transience are the only useful logical notions. For such problems, special purpose theories may be efficient for deriving system properties.

6.4 ensures, leads-to

Our primary progress operator is *leads-to*. It is defined in terms of another operator, **ensures**, which is abbreviated **en**. It is possible to eliminate **en** from the theory, replacing it by **co**-properties and transient predicates. However, to maintain continuity with [32], we introduce **en** though its role is now considerably diminished.

6.4.1 ensures

The definition of p **en** q is

$$p \ \ \textbf{en} \ \ q \ \equiv \ (p \wedge \neg q \ \ \textbf{co} \ \ p \vee q) \ \wedge \ \textbf{transient } (p \wedge \neg q)$$

It follows from the **co**-property in the above definition that once p holds, it continues to hold as long as q does not. Now we justify in operational

terms that once p holds, q holds eventually ("eventually" includes the present moment). Consider a state in which p holds and q does not. Since $p \wedge \neg q$ is transient, it is eventually falsified. From the **co**-property, whenever $p \wedge \neg q$ is falsified, $p \vee q$ holds. Hence, whenever $p \wedge \neg q$ is falsified, $\neg(p \wedge \neg q) \ \wedge \ (p \vee q)$, i.e., q, holds.

6.4.2 leads-to

The informal meaning of $p \mapsto q$ (read: p *leads-to* q) is "if p holds at any point in the computation, q will hold eventually" (here "will" applies to the current point as well as the future). There is no guarantee, unlike for **en**, that p remains *true* until q holds. The definition of $p \ \mapsto \ q$ is given by a set of inference rules.

- (basis) $\dfrac{p \ \textbf{en} \ q}{p \ \mapsto \ q}$
- (transitivity) $\dfrac{p \ \mapsto \ q \ , \ q \ \mapsto \ r}{p \ \mapsto \ r}$
- (disjunction) $\dfrac{\langle \forall p : \ p \in S : \ p \ \mapsto \ q \rangle}{\langle \exists p : \ p \in S : \ p \rangle \ \mapsto \ q}$, for any set of predicates S

For the basis, we deduce $p \ \mapsto \ q$ from $p \ \textbf{en} \ q$. The transitivity rule is justified as follows. From $p \ \mapsto \ q$, once p holds, q will hold, and from $q \ \mapsto \ r$, once q holds, r will hold. Therefore, once p holds, r will hold. For the disjunction rule, note that a state that satisfies $\langle \exists p : \ p \in S : \ p \rangle$ also satisfies some predicate p in S, so starting from this state, q will eventually be established from $p \ \mapsto \ q$.

The definition of *leads-to* —using inference rules— differs from the way we defined **co**, **transient**, and **en**. The current definition is recursive, and *leads-to* can be understood as an extreme solution (a least fixpoint) of an equation, a topic that we explore in section 6.6.2. The inference rules provide important guidelines for structuring progress proofs: either a proof follows directly from the program text (in the basis case) or it has to be structured as a transitive or disjunctive proof. These rules can also be used to establish derived rules for $\mapsto$ using structural induction over its definition (see sections 6.4.5 and 6.6.1).

In section 6.6.3, we show that the disjunction rule over a finite set can be deduced from the basis and transitivity rules. Therefore, the real power of the disjunction rule lies in its application to an infinite set of predicates.

The only mention of **en** is in the basis rule. If we replace $p \ \textbf{en} \ q$ by its definition, i.e., $(p \wedge \neg q \ \textbf{co} \ p \vee q) \ \wedge \ (\textbf{transient} \ p \wedge \neg q)$, we can eliminate **en** from the theory. The only reason for retaining **en** is for continuity with UNITY-logic [32]. An alternative definition of *leads-to* that eliminates **en** appears in exercise 6. Exercise 22 shows that the transitivity and the disjunction rules may be combined into a single rule.

Note The substitution axiom can be applied, as usual, to the progress properties. That is, an invariant can be replaced by *true* and vice versa in any progress property. □

6.4.3 Examples of specifications with leads-to

In the following, variables x and y are integers and S and T are finite sets of integers.

1. A *hungry* philosopher eats. Let h and e denote, respectively, that a particular philosopher is *hungry* or *eating*.

   $$h \;\mapsto\; e$$

2. Variable x changes eventually. For every integer m,

   $$x = m \;\mapsto\; x \neq m$$

   This can be written equivalently as (see exercise 9d)

   $$\mathit{true} \;\mapsto\; x \neq m$$

3. Variable x grows without bound. For every integer m,

   $$\mathit{true} \;\mapsto\; x > m$$

   This property is an abbreviation for

   $$\langle \forall\, m ::\; \mathit{true} \;\mapsto\; x > m \rangle$$

   It should not be confused with

   $$\mathit{true} \;\mapsto\; \langle \forall\, m ::\; x > m \rangle$$

   which happens to be nonsense; its rhs is *false*.

4. Every integer is eventually added to S. For every integer m,

   $$\mathit{true} \;\mapsto\; m \in S$$

   We cannot conclude that S grows eventually, because items may be removed from S. If items cannot be removed, i.e., for any set U,

   stable $S \supseteq U$

   then we can show that S grows without bound.

5. If values of x and y are different in any state, at least one of these variable values will change eventually.

   $x, y = m, n \ \wedge \ m \neq n \ \mapsto \ \neg(x, y = m, n)$, for all m and n, or
   $\langle \forall \ m, n : \ m \neq n : \ x, y = m, n \ \mapsto \ \neg(x, y = m, n) \rangle$

   There is no guarantee that x and y will ever become equal.

6. Every element common to S and T is eventually removed from both sets.

   $m \in (S \ \cap \ T) \ \mapsto \ m \notin (S \ \cup \ T)$

   It cannot be deduced from above that S and T will eventually become disjoint, because items may be added to both S and T.

7. Predicate p holds infinitely often.

   $true \ \mapsto \ p$ or, equivalently (see exercise 9d)
   $\neg p \ \mapsto \ p$

8. If from some point in the execution p remains *true* forever, q holds eventually (at or beyond that point). Another way of expressing this property is to say that eventually either p is *false* or q is *true*.

   $true \ \mapsto \ \neg p \vee q$

9. If p holds infinitely often (in all executions), then so does q.

   $(true \ \mapsto \ p) \ \Rightarrow \ (true \ \mapsto \ q)$

   This property does *not* say, "In any execution, if p holds infinitely often, so does q". This latter property is stronger than our formulation. This is because if in some execution p holds infinitely often and in some other execution p holds finitely often, the first formulation makes no guarantees about q in any execution; the second formulation requires q to hold infinitely often whenever p does.

10. A given program "terminates"; i.e., starting in any state that satisfies the initial condition, eventually a state is reached that satisfies the fixed point predicate FP.

    initial-condition $\mapsto \ FP$

6.4.4 Derived rules

Effective applications of the following derived rules can shorten progress proofs substantially. The rules are divided into two classes, *lightweight* and *heavyweight*. The former includes rules whose validity are easily established; the latter rules are not entirely obvious. Each application of a heavyweight rule goes a long way toward completing a progress proof.

Lightweight rules

- (implication) $\dfrac{p \;\Rightarrow\; q}{p \;\mapsto\; q}$

- (lhs strengthening, rhs weakening) $\dfrac{p \;\mapsto\; q}{\begin{array}{l} p' \wedge p \;\mapsto\; q \\ p \;\mapsto\; q \vee q' \end{array}}$

- (disjunction) $\dfrac{\langle \forall\ i ::\ p_i \;\mapsto\; q_i \rangle}{\langle \exists\ i ::\ p_i \rangle \;\mapsto\; \langle \exists\ i ::\ q_i \rangle}$

  where i is quantified over an arbitrary set, and p_i, q_i are predicates.

- (cancellation) $\dfrac{p \;\mapsto\; q \vee r \;,\; r \;\mapsto\; s}{p \;\mapsto\; q \vee s}$

We deduce from the implication rule that for any predicate p,

$$p \;\mapsto\; p \quad \text{and} \quad \mathit{false} \;\mapsto\; p$$

The lhs strengthening and the rhs weakening rules —also valid for **co**-properties— are used extensively in proofs. The disjunction rule given previously is slightly more general than the one given here for one special case, an empty set of predicates: the previous rule yields $\mathit{false} \;\mapsto\; q$ for any q, and the current rule yields $\mathit{false} \;\mapsto\; \mathit{false}$. We do not distinguish the two rules by name; it should be obvious in any application which rule is being considered. The cancellation rule played a minimal role in manipulating the **co**-properties; however, it is used heavily in progress proofs. This rule reduces to transitivity when q is *false*. Note that there is no conjunction rule for $\mapsto$ analogous to the one for **co** (see exercise 15c).

Heavyweight rules

- (impossibility) $\dfrac{p \;\mapsto\; \mathit{false}}{\neg p}$

- (PSP) $\dfrac{\begin{array}{l} p \;\mapsto\; q \;, \\ r \;\;\mathbf{co}\;\; s \end{array}}{p \wedge s \;\mapsto\; (q \wedge r) \;\vee\; (\neg r \wedge s)}$

- (induction) Let M be a total function from program states to set W. Let $(W, <)$ be well-founded. Variable m in the following premise ranges over W. Predicates p and q do not contain free occurrences of variable m.

$$\frac{\langle \forall\ m ::\ p\ \wedge\ M = m\ \mapsto\ (p\ \wedge\ M < m)\ \vee\ q \rangle}{p\ \mapsto\ q}$$

- (completion) Let p_i and q_i be predicates where i ranges over a finite set.

$$\frac{\begin{array}{l}\langle \forall\ i ::\\ \quad p_i\ \mapsto\ q_i \vee b\\ \quad q_i\ \textbf{co}\ q_i \vee b\\ \rangle\end{array}}{\langle \forall\ i ::\ p_i \rangle\ \mapsto\ \langle \forall\ i ::\ q_i \rangle\ \vee\ b}$$

The impossibility rule says that a state in which *false* holds is reachable only from an unreachable state (read the consequent of the rule as "**invariant** $\neg p$").

The PSP rule (for Progress-Safety-Progress) is perhaps the most widely used rule in progress proofs. It allows us to structure a progress proof as a safety proof —establishing r **co** s— and a progress proof —establishing $p \mapsto q$— which are then combined. This rule is so important that it should be memorized before attempting serious progress proofs.

Function M in the induction rule is called a *variant function* or a *metric*. The premise of the induction rule says that from any state in which p holds, eventually a state is reached where p still holds and the metric has a lower value, or q is established. Since M takes values from a well-founded set, its value cannot decrease indefinitely. Therefore, q is eventually established. (It is sufficient to require that M's value be in W only when $p \wedge \neg q$ holds.) Some common examples of well-founded relations are $<$ (less-than relation) over positive integers or natural numbers, lexicographic order over tuples (where the tuple entries are well-founded), proper prefix or proper subsequence relation over finite sequences, and proper subset relation over finite sets.

The completion rule is a way to take conjunctions of progress properties. As we remarked earlier, there is no conjunction rule for $\mapsto$ analogous to the rule for **co**-properties (exercise 15c). Under additional assumptions about the predicates in the rhs of the *leads-to*'s, such a conjunction rule is valid; the additional assumptions are given by the **co**-properties. Exercise 20 asks you to show that the rule is *not* valid for an infinite pair of predicates, p_i and q_i; certain generalizations of this rule appear in exercise 21.

6.4.5 Proofs of the derived rules

The lightweight rules can be proved directly from the inference rules for $\mapsto$. Many of the heavyweight rules require induction on the structure of the progress proofs in the premises.

Proofs of the lightweight rules

• (implication) $\dfrac{p \;\Rightarrow\; q}{p \;\mapsto\; q}$

Proof:

$p \wedge \neg q \;\equiv\; \mathit{false}$	, from the premise, $p \;\Rightarrow\; q$
$p \wedge \neg q$ **co** $p \vee q$	, *false* **co** r, for any r
transient $p \wedge \neg q$	, *false* is transient
p **en** q	, from above two, definition of **en**
$p \;\mapsto\; q$	, from basis inference rule for $\mapsto$

• (lhs strengthening, rhs weakening) $\dfrac{p \;\mapsto\; q}{\begin{array}{l} p' \wedge p \;\mapsto\; q\ , \\ p \;\mapsto\; q \vee q' \end{array}}$

Proof:

$p' \wedge p \;\mapsto\; p$	, implication rule (see above)
$p \;\mapsto\; q$	, premise
$p' \wedge p \;\mapsto\; q$	, transitivity on above two

Similarly, $p \;\mapsto\; q \vee q'$ from $p \;\mapsto\; q$ and $q \;\mapsto\; q \vee q'$.

• (disjunction) $\dfrac{\langle \forall\ i ::\ p_i \;\mapsto\; q_i \rangle}{\langle \exists\ i ::\ p_i \rangle \;\mapsto\; \langle \exists\ i ::\ q_i \rangle}$

Proof:
If the range of quantification for i is empty, the conclusion, $\mathit{false} \;\mapsto\; \mathit{false}$, follows from the implication rule. Assume, therefore, that the range of i is nonempty.

$\langle \forall\ i ::\ p_i \;\mapsto\; q_i \rangle$	, premise
$\langle \forall\ i ::\ p_i \;\mapsto\; \langle \exists\ i ::\ q_i \rangle\ \rangle$	, weaken rhs

Applying the disjunction inference rule, the result follows.

• (cancellation) $\dfrac{p \;\mapsto\; q \vee r\ ,\ r \;\mapsto\; s}{p \;\mapsto\; q \vee s}$

Proof:

$r \;\mapsto\; s$	, premise
$q \;\mapsto\; q$	, implication
$q \vee r \;\mapsto\; q \vee s$	, disjunction
$p \;\mapsto\; q \vee r$	, premise
$p \;\mapsto\; q \vee s$	, transitivity on above two

Proofs of the heavyweight rules

In establishing these rules, we often employ structural induction on the proofs of the progress properties in the premise. Suppose, for instance, that $p \mapsto q$ appears in the premise. This property could have been established by either (1) p **en** q, (2) $p \mapsto r$ and $r \mapsto q$ for some r, or (3) a set of properties $r \mapsto q$, for each r in a set S, where $p \equiv \langle \exists\ r :\ r \in S :\ r \rangle$. We have to prove the consequent in each of these three cases. For the first case, we draw on the derived rules of **co** and **transient** (using which **en** is defined); for the remaining two cases, using the induction hypothesis, assume that the rule holds for each of the "smaller" progress properties and include these as premises.

• (impossibility) $\dfrac{p \mapsto \mathit{false}}{\neg p}$

basis:

p **en** *false*	, premise
stable p and **transient** p	, definition of p **en** *false*
$\neg p$	, derived rule (see section 6.3.5)

transitivity:
There is a predicate r such that $p \mapsto r$ and $r \mapsto \mathit{false}$. Hence,

$\neg r$	, induction hypothesis on $r \mapsto \mathit{false}$
$p \mapsto \mathit{false}$	, from $p \mapsto r$ and $\neg r$
$\neg p$	, induction hypothesis

disjunction:
There is a set S of predicates such that $r \mapsto \mathit{false}$ for every r, $r \in S$, and $p \equiv \langle \exists\ r :\ r \in S :\ r \rangle$. Hence, for every r in S,

$r \mapsto \mathit{false}$	, premise
$\neg r$	, induction hypothesis
$\neg p$	, from $p \equiv \langle \exists\ r :\ r \in S :\ \mathit{false} \rangle$ $\square$

• (PSP) $\dfrac{p \mapsto q,\quad r\ \mathbf{co}\ s}{p \wedge s \mapsto (q \wedge r) \vee (\neg r \wedge s)}$

basis:

p **en** q	, premise
$p \wedge \neg q$ **co** $p \vee q$	, from the definition of **en**
r **co** s	, premise
$p \wedge \neg q \wedge r$ **co** $(p \wedge s) \vee (q \wedge s)$	, conjunction of the above two
$p \wedge \neg q \wedge r$ **co** $(p \wedge s) \vee (q \wedge (r \vee (\neg r \wedge s)))$	, weaken rhs

$p \wedge \neg q \wedge r$ **co** $(p \wedge s) \ \vee \ (q \ \wedge \ r) \ \vee \ (\neg r \wedge s)$ (1)
, weaken rhs
transient $p \wedge \neg q$, from the premise p **en** q
transient $p \wedge \neg q \wedge r$, strengthen above (section 6.3.5)
$p \wedge s$ **en** $(q \wedge r) \ \vee \ (\neg r \wedge s)$, from (1) and above using the definition of **en** (use $r \ \Rightarrow \ s$)
$p \wedge s \ \mapsto \ (q \wedge r) \ \vee \ (\neg r \wedge s)$, basis rule for $\mapsto$

transitivity:
There is a predicate b such that $p \ \mapsto \ b$ and $b \ \mapsto \ q$. Hence,

$b \wedge s \ \mapsto \ (q \wedge r) \ \vee \ (\neg r \wedge s)$, induction on $b \ \mapsto \ q$ and r **co** s
$b \wedge r \ \mapsto \ (q \wedge r) \ \vee \ (\neg r \wedge s)$, strengthen lhs using $r \Rightarrow s$
$p \wedge s \ \mapsto \ (b \wedge r) \ \vee \ (\neg r \wedge s)$, induction on $p \ \mapsto \ b$ and r **co** s
$p \wedge s \ \mapsto \ (q \wedge r) \ \vee \ (\neg r \wedge s)$, cancellation on above two

disjunction:
There is a set S of predicates such that $b \ \mapsto \ q$ for every b in S and $p \ \equiv \ \langle \exists \ b : \ b \in S : \ b \rangle$. Hence, for b in S,

$b \ \mapsto \ q$, premise
r **co** s , premise
$b \wedge s \ \mapsto \ (q \wedge r) \ \vee \ (\neg r \wedge s)$, induction hypothesis
$\langle \exists \ b : \ b \in S : \ b \wedge s \rangle \ \mapsto \ (q \wedge r) \ \vee \ (\neg r \wedge s)$
, disjunction
$\langle \exists \ b : \ b \in S : \ b \rangle \wedge s \ \mapsto \ (q \wedge r) \ \vee \ (\neg r \wedge s)$
, predicate calculus
$p \wedge s \ \mapsto \ (q \wedge r) \ \vee \ (\neg r \wedge s)$, $p \ \equiv \ \langle \exists \ b : \ b \in S : \ b \rangle$ □

• (induction) $\dfrac{\langle \forall \ m :: \ p \ \wedge \ M = m \ \mapsto \ (p \ \wedge \ M < m) \ \vee \ q \rangle}{p \ \mapsto \ q}$

where M is a total function from program states to W and $(W, <)$ is well-founded. The premise can be written as follows: for all m in W,

$$p \ \wedge \ M = m \ \mapsto \ \langle \exists \ n : \ n < m : \ p \ \wedge \ M = n \rangle \ \vee \ q$$

In the rest of this proof m and n are quantified over W. We use the abbreviation $A.n$ for $(p \ \wedge \ M = n \ \mapsto \ q)$. For any m in W,

$\langle \forall \ n : \ n < m : \ A.n \rangle$
$\Rightarrow$ {expanding $A.n$ using its definition}
$\langle \forall \ n : \ n < m : \ p \ \wedge \ M = n \ \mapsto \ q \rangle$
$\Rightarrow$ {disjunction}
$\langle \exists \ n : \ n < m : \ p \ \wedge \ M = n \rangle \ \mapsto \ q$
$\Rightarrow$ {cancellation using the premise:
$p \ \wedge \ M = m \ \mapsto \langle \exists \ n : \ n < m : \ p \ \wedge \ M = n \rangle \ \vee \ q$}

$p \ \wedge \ M = m \ \mapsto \ q$
$\Rightarrow$ {using the abbreviation $A.m$ for the above}
$A.m$

We have thus established that for any m in W,
$\langle \forall \ n : \ n < m : \ A.n \rangle \ \Rightarrow \ A.m$
Applying the induction principle (on well-founded sets) we conclude
$\langle \forall \ m :: \ A.m \rangle$.

Now
$\langle \forall \ m :: \ A.m \rangle$
$\Rightarrow$ {expanding $A.m$}
$\langle \forall \ m :: \ p \ \wedge \ M = m \ \mapsto \ q \rangle$
$\Rightarrow$ {disjunction and that m does not appear in p}
$p \ \wedge \ \langle \exists \ m :: \ M = m \rangle \ \mapsto \ q$
$\Rightarrow$ {range of total function M is W, so $\langle \exists \ m :: \ M = m \rangle \ \equiv \ \mathit{true}$}
$p \ \mapsto \ q$ □

- (completion)
$$\frac{\langle \forall \ i : \ 0 \leq i < N : \ p_i \ \mapsto \ q_i \vee b \ \ q_i \ \textbf{co} \ q_i \vee b \ \rangle}{\langle \forall \ i : \ 0 \leq i < N : \ p_i \rangle \ \mapsto \ \langle \forall \ i : \ 0 \leq i < N : \ q_i \rangle \ \vee \ b}$$

We prove the result only for the special case $b \equiv \mathit{false}$. The proof of the general case is left to exercise 21. We prove the following by induction on N.

$$\frac{\langle \forall \ i : \ 0 \leq i < N : \ p_i \ \mapsto \ q_i \quad (1) \ \ \textbf{stable} \ q_i \quad (2) \ \rangle}{\langle \forall \ i : \ 0 \leq i < N : \ p_i \rangle \ \mapsto \ \langle \forall \ i : \ 0 \leq i < N : \ q_i \rangle}$$

Case $N = 0$: The consequent of the inference rule is

$\mathit{true} \ \mapsto \ \mathit{true}$, which follows from the implication rule for $\mapsto$

Case $N + 1$: The consequent of the inference rule is

$\langle \forall \ i : \ 0 \leq i < N + 1 : \ p_i \rangle \ \mapsto \ \langle \forall \ i : \ 0 \leq i < N + 1 : \ q_i \rangle$

Writing P for $\langle \forall \ i : \ 0 \leq i < N : \ p_i \rangle$ and Q for $\langle \forall \ i : \ 0 \leq i < N : \ q_i \rangle$, we have to show

$P \wedge p_N \ \mapsto \ Q \wedge q_N$

Essential to the proof is the notion of *wlt* (see section 6.6.1). From condition (1), $p_N \mapsto q_N$. So there is a predicate r (r is $wlt.q_N$) such that,

$r \mapsto q_N$	, from W3 of section 6.6.1	(3)
$p_N \Rightarrow r$	, from $p_N \mapsto q_N$ and W2 of section 6.6.1	(4)
$q_N \Rightarrow r$	, from W4 of section 6.6.1	(5)
$r \wedge \neg q_N$ **co** r	, from W5 of section 6.6.1	(6)

Then we have

stable q_N	, from (2)	
stable $r \vee q_N$	, disjunction of above and (6)	
stable r	, above and (5)	(7)

The main proof is as follows:

$P \wedge p_N$
$\Rightarrow$ $\{p_N \Rightarrow r$, from (4)$\}$
$P \wedge r$
$\mapsto$ $\{P \mapsto Q$, induction hypothesis
stable r , from (7)
$P \wedge r \mapsto Q \wedge r$, PSP$\}$
$Q \wedge r$
$\mapsto$ $\{r \mapsto q_N$, from (3)
stable Q , conjunction over (2)
$Q \wedge r \mapsto Q \wedge q_N$, PSP$\}$
$Q \wedge q_N$ □

6.4.6 Corollaries of the derived rules

The following corollary, like its namesake for **co**, permits us to conjoin a stable predicate to both sides of a *leads-to* property.

- (stable conjunction) $\dfrac{p \mapsto q \ , \ \textbf{stable}\ r}{p \wedge r \mapsto q \wedge r}$

  Proof: set s to r in the PSP rule. □

- (corollary of PSP) Given r **co** s we have $r \Rightarrow s$. Therefore, strengthening the lhs (by replacing s with r) and/or weakening the rhs (by replacing r with s) in the consequent of the PSP rule yield the following rules. For completeness, we include the original version of the PSP. Henceforth, the hint "PSP" in a proof refers to any of these rules.

$$\frac{\begin{array}{l} p \;\mapsto\; q \\ r \;\textbf{co}\; s \end{array}}{\begin{array}{lll} p \wedge s \;\mapsto\; (q \wedge r) \;\vee\; (\neg r \wedge s) & , \text{PSP} \\ p \wedge r \;\mapsto\; (q \wedge r) \;\vee\; (\neg r \wedge s) & , \text{strengthen lhs} \\ p \wedge s \;\mapsto\; (q \wedge s) \;\vee\; (\neg r \wedge s) & , \text{weaken rhs} \\ p \wedge r \;\mapsto\; (q \wedge s) \;\vee\; (\neg r \wedge s) & , \text{both of the above} \end{array}}$$

- (corollary of induction) In the following corollary, the range of m (in the premise of the induction rule) is restricted by a predicate r. Suppose "$<$" is a well-founded order over the set of values that satisfy r.

$$\frac{\begin{array}{l} \langle \forall\, m :\; r.m : \\ \quad p \;\wedge\; M = m \;\mapsto\; (p \;\wedge\; M < m) \;\vee\; q \\ \rangle \end{array}}{p \;\mapsto\; (p \wedge \neg r.M) \;\vee\; q}$$

  Proof:

$$\begin{array}{ll} p \;\wedge\; M = m \;\wedge\; r.m \;\mapsto\; (p \;\wedge\; M < m) \;\vee\; q & , \text{premise} \\ p \;\wedge\; M = m \;\wedge\; r.M \;\mapsto\; (p \;\wedge\; M < m) \;\vee\; q & , \text{rewrite lhs} \\ p \;\wedge\; M = m \;\wedge\; \neg r.M \;\mapsto\; p \;\wedge\; \neg r.M & , \text{implication} \\ p \;\wedge\; M = m \;\mapsto\; (p \;\wedge\; M < m) \;\vee\; (p \;\wedge\; \neg r.M) \;\vee\; q & , \text{disjunction} \\ p \;\mapsto\; (p \;\wedge\; \neg r.M) \;\vee\; q & , \text{induction } \square \end{array}$$

  An important special case arises when M is integer valued, $r.m$ is of the form $m > L$ for some lower bound L, and $<$ is the standard less-than relation over integers. Then, we conclude from the above corollary:

- (induction over integers)

$$\frac{\begin{array}{l} \langle \forall\, m :\; m > L : \\ \quad p \;\wedge\; M = m \;\mapsto\; (p \;\wedge\; M < m) \;\vee\; q \\ \rangle \end{array}}{p \;\mapsto\; (p \;\wedge\; M \leq L) \;\vee\; q}$$

  If $r.m$ imposes no lower bound on m —for instance, $r.m \;\equiv\; \mathit{true}$— then the conclusion holds for any L.

  Using "greater than" in place of "less than", we have the analogous

- (induction over integers)

$$\frac{\begin{array}{l} \langle \forall\, m :\; m < L : \\ \quad p \;\wedge\; M = m \;\mapsto\; (p \;\wedge\; M > m) \;\vee\; q \\ \rangle \end{array}}{p \;\mapsto\; (p \;\wedge\; M \geq L) \;\vee\; q}$$

A simple corollary of the induction rule, where $<$ is a well-founded order, is

- (simple corollary of induction)

$$\frac{\langle \forall\ m ::\ \ p\ \wedge\ M = m\ \mapsto\ M < m\ \rangle}{true\ \mapsto\ \neg p}$$

6.5 Applications

We consider a few examples in which we employ the progress operators for specifications and deductions of program properties. The emphasis is on converting verbal descriptions to formal specifications and using the derived rules for deductions. In most cases, we also supply the typical verbal arguments that justify the deductions and contrast them with formal proofs.

6.5.1 Non-operational descriptions of algorithms

We consider the algorithm for computing the maximum of a nonempty set S of numbers, as described in section 5.5.1. The safety properties we postulated are as follows. Here, v is the variable in which the maximum is being computed and m is any integer.

$$\textbf{initially}\ v = -\infty \tag{ND1}$$
$$v = m\ \ \textbf{co}\ \ v = m\ \vee\ (v \in S\ \wedge\ v > m) \tag{ND2}$$

We derived a number of safety properties, including

$$\textbf{invariant}\ v \leq M \tag{ND3}$$

where M is the maximum in S, i.e., $M = \langle max\ x :\ x \in S :\ x \rangle$. Now we postulate the following progress property. For all m,

$$m \in S\ \mapsto\ v \geq m \tag{ND4}$$

which says that eventually v is at least m for any m in S. We establish that v will eventually equal M.

• Proof of $true\ \mapsto\ v = M$:

$m \in S\ \mapsto\ v \geq m$	, (ND4)
$M \in S\ \mapsto\ v \geq M$	, instantiating m by M
$true\ \mapsto\ v \geq M$	, substitution axiom on lhs: $M \in S\ \equiv\ true$
$true\ \mapsto\ v = M$	, conjoin invariant (ND3) with rhs □

The following implementation was considered in section 5.5.1. Let S be represented by an array A and s be an index into A. We had

$$\textbf{initially } v, s = -\infty, 0 \text{ and} \tag{ND5}$$
$$v = \langle max\ i :\ 0 \le i < s :\ A[i] \rangle \tag{ND6}$$

We show that v acquires the maximum value in A given that the index s increases as long as the end of the array is not reached. Assume that for all k, $0 \le k < N$ (where A has N, $N \ge 0$, elements),

$$s = k \ \mapsto\ s = k + 1 \tag{ND7}$$

and show,

• Proof of $v, s = -\infty, 0 \ \mapsto\ v = \langle max\ i :\ 0 \le i < N :\ A[i] \rangle$:

$s \le N \ \mapsto\ s = N$, from (ND7) (see Exercise 17a)
$s = 0 \ \mapsto\ s = N$, strengthen lhs of the above; use $N \ge 0$
$s = 0 \ \mapsto\ v = \langle max\ i :\ 0 \le i < N :\ A[i] \rangle$
, substitution axiom; conjoin (ND6) to rhs
$v, s = -\infty, 0 \ \mapsto\ v = \langle max\ i :\ 0 \le i < N :\ A[i] \rangle$
, strengthen lhs □

6.5.2 Common meeting time

The common meeting time problem was discussed in section 5.5.2 where functions f and g mapped non-negative reals to non-negative reals. Now we tighten the requirements on f and g: they map natural numbers to natural numbers. Rewriting the earlier properties, we have

$$m \le n \ \Rightarrow\ f(m) \le f(n) \tag{CMT1}$$
$$m \le n \ \Rightarrow\ g(m) \le g(n)$$

A variable t —previously of type real, now of type natural number— is postulated to satisfy

$$\textbf{initially } t = 0 \tag{CMT2}$$
$$t = m \ \textbf{co}\ t \le max(f(m), g(m)) \tag{CMT3}$$

We had established earlier, from (CMT1–CMT3), that t exceeds no common meeting time:

$$com(n) \ \Rightarrow\ t \le n \tag{CMT4}$$

The essential safety property, (CMT3), can be implemented by program *skip*, which does not change t. To guarantee that t eventually equals the earliest common meeting time (provided that one exists), we add a progress requirement: if t is not a common meeting time, then its value increases eventually.

$$\neg com(t) \ \wedge \ t = m \ \mapsto \ t > m \qquad \text{(CMT6)}$$

We show

$$\langle \exists \ n :: \ com(n) \rangle \ \mapsto \ com(t). \qquad \text{(CMT7)}$$

That is, if there is a common meeting time, t will eventually equal a common meeting time. From (CMT4), t can never exceed any common meeting time. Therefore, if there is a common meeting time, t will eventually equal the earliest common meeting time.

- Proof of (CMT7), $\langle \exists \ n :: \ com(n) \rangle \ \mapsto \ com(t)$:
  For arbitrary natural numbers m and n:

$\neg com(t) \ \wedge \ t = m \ \mapsto \ t > m$	, rewrite (CMT6)
$com(t) \ \wedge \ t = m \ \mapsto \ com(t)$	, implication
$t = m \ \mapsto \ t > m \ \vee \ com(t)$	, disjunction of the above two
$true \ \mapsto \ t > n \ \vee \ com(t)$	, induction over integers
$com(n) \ \mapsto \ \langle com(n) \ \wedge \ t > n \rangle \ \vee \ \langle com(n) \ \wedge \ com(t) \rangle$	, stable conjunction with $com(n)$
$com(n) \ \mapsto \ com(n) \ \wedge \ com(t)$	, (CMT4): first term in rhs is *false*
$com(n) \ \mapsto \ com(t)$	, weaken rhs
$\langle \exists \ n :: \ com(n) \rangle \ \mapsto \ com(t)$	, disjunction over all n □

This proof is invalid if t is real because the induction step is then invalid. The reader can construct a counterexample to (CMT7) by having t increase extremely slowly, say, by $1/2^i$ in step i.

One way to implement progress condition (CMT6) is to increment t by 1 in each step and check if $com(t)$ holds. The monotonicity condition on f and g —given by (CMT1)— is far too weak to permit many other strategies. If the functions are also ascending, i.e., for all natural n

$$n \leq f(n) \text{ and } n \leq g(n),$$

then programs (P1, P2) in section 5.5.2 satisfy

$$\neg com(t) \wedge t = m \ \ \textbf{en} \ \ t > m$$

and hence (CMT6) as well.

6.5.3 Token ring

Safety properties of a token ring were postulated in section 5.5.3.

Notation We write t_i to denote that process i is *thinking*; similarly h_i and e_i stand for *hungry* and *eating*. These predicates are mutually exclusive, and $h_i \vee t_i \vee e_i$ holds. The position of the token is in variable p, i.e., $p = i$ (as in TR5) denotes that process i holds the token. In the following, i ranges over all processes and i' is the right neighbor of i. □

$$\textbf{initially}\ e_i \ \Rightarrow\ p = i \tag{TR0}$$
$$e_i \ \textbf{co}\ e_i \vee t_i \tag{TR1}$$
$$t_i \ \textbf{co}\ t_i \vee h_i \tag{TR2}$$
$$h_i \ \textbf{co}\ h_i \vee e_i \tag{TR3}$$
$$h_i \ \wedge\ p \neq i \ \textbf{co}\ h_i \tag{TR4}$$
$$p = i \ \textbf{co}\ p = i \ \vee\ \neg e_i \tag{TR5}$$

In section 5.5.3, we deduced mutual exclusion, a safety property, from (TR0–TR5). Now, we postulate some progress properties and establish the absence of starvation.

First, we require that a *hungry* token holder transit to *eating*.

$$h_i \ \wedge\ p = i \ \mapsto\ e_i \tag{TR6}$$

Next, we require that the token move from the current token holder to its right neighbor.

$$p = i \ \mapsto\ p = i' \tag{TR7}$$

Note that (TR7) does not require the token to go directly from i to i'.

Our ultimate goal is to establish absence of starvation for process j, $0 \leq j < N$, which is written as

$$h_j \ \mapsto\ e_j \tag{TR8}$$

To prove (TR8), consider an arbitrary j, $0 \leq j < N$. First we show that j eventually holds the token, i.e.,

$$\mathit{true} \ \mapsto\ p = j \tag{TR9}$$

The proof of (TR9) is by induction over (TR7). To apply the induction rule, we define a total order $\prec$ over the processes. Let j be the highest process in this ordering and the processes become successively smaller clockwise along the ring from j. That is,

$$\ldots i' \prec i \ldots \prec j' \prec j$$

Formally, $i' \prec i$ for all process indices i, where $i' \neq j$.

• Proof of (TR9) $\mathit{true} \ \mapsto\ p = j$:

$p = i \ \mapsto\ p \prec i$, for all i where $i' \neq j$
, from (TR7)
$p = i \ \mapsto\ p = j$, for i where $i' = j$
, from (TR7)
$\langle \forall\ i ::\ p = i \ \mapsto\ p \prec i \ \vee\ p = j \rangle$
, from the above two
$\mathit{true} \ \mapsto\ p = j$, induction ($\prec$ is a total order on processes) □

• Proof of (TR8) $h_j \;\mapsto\; e_j$:

$h_j \;\textbf{co}\; h_j \vee e_j$	, from (TR3) using j for i
$true \;\mapsto\; p = j$	, (TR9)
$h_j \;\mapsto\; (h_j \wedge p = j) \vee e_j$	, PSP {use $(\neg h_j \wedge e_j) \equiv e_j$}
$h_j \wedge p = j \;\mapsto\; e_j$	, from (TR6) using j for i
$h_j \;\mapsto\; e_j$	, cancellation on above two □

The standard verbal argument for this problem follows the above proof steps closely. The formalism allows us to combine a few special cases. The role of induction —if every process relinquishes the token eventually, every process acquires the token eventually— is made explicit in the proof. Note that the proof is entirely independent of the kind of fairness assumed in (TR6) and (TR7).

6.5.4 Unordered channel

We consider a directed channel along which messages are sent from one process to another. We design a protocol by which every message sent is eventually delivered, though the order of delivery may be different from the order of transmission. The following scheme implements the protocol of section 4.1.3. Every message sent along the channel is assigned a *sequence number* (a natural number) and the sequence numbers are strictly increasing in the order of transmission. A message that has the lowest sequence number in the channel at any point in the computation is delivered eventually. We claim that this scheme guarantees delivery of every message sent. However, the delivery order may not be monotonic in sequence numbers.

The proposed scheme can be described by the following properties. In this description, s is the set of sequence numbers of the messages in the channel (henceforth, assume that s is a finite set), and x and y are arbitrary sequence numbers. The lowest sequence number in s is $s.min$ (if s is empty $s.min$ is ∞). Therefore, $s.min$ does not decrease as a result of adding a number to s, from the monotonicity of sequence numbers. Also, removing a number from s does not decrease $s.min$; if s becomes empty, $s.min = \infty$. The safety property, (UC1), says that $s.min$ never decreases. The progress property, (UC2), says that the smallest element of s is eventually removed from s. Our goal, (UC3), is to show that every sequence number is eventually outside s.

$$s.min = x \;\textbf{co}\; s.min \geq x \qquad \text{(UC1)}$$
$$s.min = x \;\mapsto\; x \notin s \qquad \text{(UC2)}$$

Show:

$$true \;\mapsto\; y \notin s \qquad \text{(UC3)}$$

The proof is based on the following argument. From (UC1), $s.min$ never decreases. From (UC2), $s.min$ is eventually removed, so $s.min$ eventu-

ally increases. Therefore, $s.min$ increases without bound as long as s is nonempty; so every number is eventually outside s.

• Proof of (UC3), $true \;\mapsto\; y \notin s$:

$s.min = x \;\mapsto\; (x \notin s \;\wedge\; s.min = x) \;\vee\; (s.min \neq x \;\wedge\; s.min \geq x)$
, PSP on (UC1, UC2)
$s.min = x \;\mapsto\; s.min > x$
, simplify the rhs
$true \;\mapsto\; s.min > y$
, induction on integers (use y as the upper bound)
$true \;\mapsto\; y \notin s$
, weaken the rhs using $s.min > y \;\Rightarrow\; y \notin s$ □

6.5.5 Shared counter

A finite nonempty set of processes share a counter *ctr* whose value is a natural number. Each process has three local variables: b (boolean), *old*, and *new* (natural). Henceforth, j denotes a process index, and for local variables of process j, we write b_j, old_j, and new_j. Initially, all b_j are *false*. Process j has three associated actions, shown below, and the complete program includes the actions of all processes.

$\alpha_j ::$	$b_j \;\wedge\; ctr = old_j$	$\rightarrow$	$ctr,\; b_j$	$:=$	$new_j,\; false$
$\beta_j ::$	$b_j \;\wedge\; ctr \neq old_j$	$\rightarrow$	$old_j,\; b_j$	$:=$	$ctr,\; false$
$\gamma_j ::$	$\neg b_j$	$\rightarrow$	$new_j,\; b_j$	$:=$	$old_j + 1,\; true$

It is required to show that the counter value eventually increases; i.e., for any natural number c,

$ctr = c \;\mapsto\; ctr > c$

First, we prove an invariant:

invariant $b_j \;\Rightarrow (new_j = old_j + 1)$, for all j

Let q_j stand for $b_j \;\Rightarrow (new_j = old_j + 1)$. Initially every q_j holds since every b_j is *false*. An effective execution of α_j or β_j preserves q_j because b_j is set to *false*. An effective execution of γ_j sets b_j to *true* and establishes $new_j = old_j + 1$, thus preserving q_j. Actions of processes other than j preserve q_j because q_j names only the local variables of j. We leave the formal proof to the reader.

To show that *ctr* eventually increases, we introduce two auxiliary variables: *nc* is the number of processes j, for which $ctr \neq old_j$, and *nb* is the number of processes for which b_j is *false* :

$nc = \langle +\ j :\ ctr \neq old_j :\ 1 \rangle$
$nb = \langle +\ j :\ \neg b_j :\ 1 \rangle$.

The main observation in the progress proof is that eventually either ctr increases or the pair (nc, nb) decreases lexicographically. Since (nc, nb) cannot decrease forever (lexicographic ordering over pairs of naturals is well-founded), ctr eventually increases. We use the following two predicates, P and Q, in the proof.

$$\begin{array}{lll} P & \equiv & \langle ctr = c \ \wedge\ (nc, nb) = (M, N) \rangle \\ & & \quad \text{, where } M \text{ and } N \text{ are free} \\ Q & \equiv & \langle ctr = c \ \wedge\ (nc, nb) \prec (M, N) \rangle \ \vee\ \langle ctr > c \rangle \\ & & \quad \text{, where } \prec \text{ is the lexicographic order} \end{array}$$

• Proof of $ctr = c \ \mapsto\ ctr > c$:

Effective execution of *any* (non-*skip*) action establishes Q as a post-condition given P as a pre-condition, i.e., for any j,

$$\begin{array}{lll} \{P \wedge\ b_j\ \wedge\ ctr = old_j\} & \alpha_j & \{Q\} \\ \{P \wedge\ b_j\ \wedge\ ctr \neq old_j\} & \beta_j & \{Q\} \\ \{P \wedge\ \neg b_j\} & \gamma_j & \{Q\} \end{array}$$

We leave the formal proofs of these assertions to the reader. Informally: α_j increases ctr (since **invariant** $b_j \Rightarrow new_j = old_j + 1$ holds for this program), β_j decreases nc (though it may increase nb), and γ_j preserves nc and decreases nb.

Next, we use the result of exercise 5. The disjunctions of the guards of α_j, β_j, and γ_j, over all j, is *true*. So we get

$$\begin{array}{ll} P \ \mapsto\ Q & \text{, use exercise 5} \\ \langle ctr = c \ \wedge\ (nc, nb) = (M, N) \rangle & \\ \quad \mapsto\ \langle ctr = c \ \wedge\ (nc, nb) \prec (M, N) \rangle \ \vee\ \langle ctr > c \rangle & \\ & \text{, expand } P \text{ and } Q \\ ctr = c \ \mapsto\ ctr > c & \text{, induction} \end{array}$$ □

6.5.6 Dynamic graphs

In section 5.5.10, a finite directed graph is modified by directing all incident edges on a node toward that node. The effect of this operation is to make that node a *bottom* node (a node that has no outgoing edge). We showed that this operation does not create new paths to non-bottom nodes, i.e., for arbitrary nodes u and v, where $u\ R\ v$ denotes that there is a path from u to v and $v.\ \bot$ that v is bottom.

$$\neg u\ R\ v \ \ \textbf{co}\ \ \neg u\ R\ v \ \vee\ v.\ \bot \qquad \text{(G1)}$$

We showed that one of the consequences of (G1) is that once the graph becomes acyclic, it remains acyclic. Henceforth, we assume that the graph is initially acyclic; therefore, it is always acyclic, i.e., for all u

$$\neg u \; R \; u \tag{G2}$$

Now we add a progress condition. A node is *top* if it has no incoming edge. Hence, a top node remains top as long as edge redirection is not applied to it. We require that edge redirection be applied to every top node eventually; the effect is to make the node a bottom node. Using $v.\top$ to denote that v is a top node, we assume

$$v.\top \;\mapsto\; v.\bot \tag{G3}$$

We show that every node eventually becomes a bottom node, i.e.,

$$\mathit{true} \;\mapsto\; u.\bot \tag{G4}$$

The argument behind the formal proof of (G4) is as follows. For any node u, consider the set of *ancestors* of u, i.e., the nodes that have paths to u. This set does not grow as long as u is non-bottom, a safety property that we establish from G1. Next, since the graph is acyclic, either u is top or u has an ancestor v that is top. In the first case, from G3, u will become bottom. In the second case, from G3, v will become bottom and therefore, cease to be an ancestor of u; hence, the ancestor set of u decreases in size eventually. Since the ancestor set cannot decrease forever, u will become top and then, from G3, eventually become bottom. Next, we formalize this argument.

In the following, $u.an$ denotes the ancestor set of u, i.e., the set of nodes from which there is a path to u.

$$v \in u.an \;\equiv\; v \; R \; u \tag{G5}$$

We use the following facts about top and bottom. A node is top iff its ancestor set is empty (G6) and a bottom node does not belong to any ancestor set (G7).

$$u.\top \;\equiv\; u.an = \emptyset \tag{G6}$$
$$v.\bot \;\Rightarrow\; v \notin u.an \tag{G7}$$

Henceforth, S is any fixed set of nodes, and u and v are arbitrary nodes. We prove that $u.an$ does not grow as long as u remains non-bottom.

Lemma $\quad u.an = S \;\; \mathbf{co} \;\; u.an \subseteq S \;\vee\; u.\bot$ (G8)

Proof:

$\neg v \; R \; u \;\; \mathbf{co} \;\; \neg v \; R \; u \;\vee\; u.\bot$, G1 with u, v interchanged

$v \notin u.an \;\; \mathbf{co} \;\; v \notin u.an \;\vee\; u.\bot$, use G5 to rewrite $\neg v \; R \; u$

$\langle \forall \; v : \; v \notin S : \; v \notin u.an \rangle \;\; \mathbf{co} \;\; \langle \forall \; v : \; v \notin S : \; v \notin u.an \rangle \;\vee\; u.\bot$, conjunction over all v, $v \notin S$

$u.an \subseteq S$ **co** $u.an \subseteq S \ \vee \ u.\bot$, simplify the two sides
$u.an = S$ **co** $u.an \subseteq S \ \vee \ u.\bot$, strengthen the left side □

(G8) states the essential safety property. Next we prove an analogous progress property: the ancestor set of u eventually shrinks as long as u remains non-bottom. (G4) follows by combining the safety and progress results.

Lemma (G4) $true \ \mapsto \ u.\bot$
Proof:

$v.\top \ \mapsto \ v.\bot$
, G3
$u.an = S \ \wedge \ v.\top \ \mapsto \ v \notin u.an$
, strengthen lhs and weaken rhs (using G7)
$u.an = S \ \wedge \ v.\top \ \wedge \ v \in S \ \mapsto \ v \notin u.an \ \wedge \ v \in S$
, conjunction with $v \in S$; $v \in S$ is stable
$u.an = S \ \wedge \ v.\top \ \wedge \ v \in S \ \mapsto \ u.an \neq S$
, weaken rhs
$u.an = S \ \wedge \ \langle \exists \, v :: \ v.\top \ \wedge \ v \in S \rangle \ \mapsto \ u.an \neq S$
, disjunction over v
$u.an = S \ \wedge \ S \neq \emptyset \ \mapsto \ u.an \neq S$
, an acyclic graph S has a top node $\equiv \ (S \neq \emptyset)$
$u.an = S$ **co** $u.an \subseteq S \ \vee \ u.\bot$
, from lemma (G8)
$u.an = S \ \wedge \ S \neq \emptyset \ \mapsto \ u.an \subset S \ \vee \ u.\bot$
, PSP and weaken rhs
$u.an = S \ \wedge \ S = \emptyset \ \mapsto \ u.\bot$
, $u.an = \emptyset \ \Rightarrow \ \{\text{G6}\} \ u.\top \ \mapsto \ \{\text{G3}\} \ u.\bot$
$u.an = S \ \mapsto \ u.an \subset S \ \vee \ u.\bot$
, disjunction of the above two
$true \ \mapsto \ u.\bot$
, induction (subset relation over finite sets is well founded) □

6.5.7 Treatment of strong fairness

In this section we show that progress properties under strong fairness can be proved using only the concepts developed for weak fairness.

A naive approach

A standard example of strong fairness is a binary semaphore shared by two processes. Let x and y be the number of times that the two processes successfully complete their P-operations. A process that holds the semaphore eventually releases it. In the program below, α implements the granting of

the semaphore to the x-process, and β to the y-process; γ implements the V operation.

$$
\begin{array}{llll}
\alpha :: s & \rightarrow & s, x & := \mathit{false}, x+1 \\
[\!] \ \beta :: s & \rightarrow & s, y & := \mathit{false}, y+1 \\
[\!] \ \gamma :: s & & & := \mathit{true}
\end{array}
$$

Under the weak fairness assumption, we can establish only that $x + y$ increases without bound; we can make no such guarantee for either x or y. Now, we impose the following strong fairness condition for action α: if the guard of α (i.e., s) is infinitely often *true*, α is effectively executed infinitely often. We assume weak fairness for the remaining actions, β and γ. The goal is to show that x increases without bound under this strong fairness condition.

The strong fairness condition can be added as a property of the program; for any integer k,

$$(\mathit{true} \ \mapsto \ s) \ \Rightarrow \ (x = k \ \mapsto \ x = k + 1)$$

So we regard the system as consisting of the program (with weak fairness condition) plus the property given above. It is then straightforward to show that for any integer m,

$$\mathit{true} \ \mapsto \ x > m$$

The proof is as follows:

true **en** s	, from the program text
$\mathit{true} \ \mapsto \ s$	, basis rule of $\mapsto$
$x = k \ \mapsto \ x = k + 1$	, use the strong fairness condition
$\mathit{true} \ \mapsto \ x > m$	, induction on integers

This treatment of strong fairness, though sound, is incomplete. To see this, consider the following program where x and y are integers, b is boolean, and only action α is executed under strong fairness.

$$
\begin{array}{llll}
\alpha :: b & \rightarrow & x & := x+1 \\
[\!] \ \beta :: \neg b & \rightarrow & y & := y+1 \\
[\!] \ \gamma :: x = y & \rightarrow & b & := \neg b
\end{array}
$$

We show that $x + y$ increases without bound. First, x and y never decrease. Next, suppose b holds infinitely often in an execution. Then α is executed effectively an infinite number of times, increasing $x + y$ without bound. If b holds finitely often, $\neg b$ holds forever beyond some point in that execution. Every execution of β is effective from then on. Hence, y increases without bound, and so does $x + y$.

Unfortunately, the method we have outlined cannot be used to prove this result; adding the following strong fairness condition to the program

$$(\mathit{true} \;\mapsto\; b) \;\Rightarrow\; (x = k \;\mapsto\; x = k + 1)$$

does not help us to prove that $x+y$ increases. To exploit the strong fairness condition, we need to prove its antecedent:

$$\mathit{true} \;\mapsto\; b$$

We cannot prove this result because, if initially $x, y, b = 0, 1, \mathit{false}$, then $x < y \;\wedge\; \neg b$ persists forever. The trouble is

$$(\mathit{true} \;\mapsto\; b) \;\Rightarrow\; (x = k \;\mapsto\; x = k + 1)$$

says merely that if b holds infinitely often in *all* execution sequences, x will be incremented infinitely often in *all* execution sequences. We need a stronger property: in any execution sequence, if b holds infinitely often, x is incremented infinitely often (in that sequence). We next show how to state such properties by encoding the eventual operator, $\Diamond$, of temporal logic [127, 128] in our theory.

Encoding the Eventual Operator

In temporal logic *eventually* b, written as $\Diamond b$, means that b holds now or will hold in a future state. Henceforth, we write eb for $\Diamond b$ and define it by the following properties:

$$b \;\Rightarrow\; eb$$
$$eb \;\mapsto\; b$$
$$\textbf{stable}\;\; \neg eb$$

Predicate eb may be regarded as an auxiliary variable of the program. More accurately, eb is a prophecy variable [2] whose value in any state depends on the future execution. The value of eb in any state is completely determined by the properties given above. Specifically, eb holds in a state iff b holds in the current or some future state. We validate this claim by considering two cases:

1. eb holds in a state: from $eb \mapsto b$, b holds in the current or some future state.
2. $\neg eb$ holds in a state: from the stability of $\neg eb$, $\neg eb$ holds in the current and all future states, and so does $\neg b$, from $\neg eb \Rightarrow \neg b$.

Let s be the longest suffix that contains only $\neg b$-states. Then, from the above observation, $\neg eb$ holds in only and all states of s. If b holds infinitely often, s is empty, and eb holds in all states. If b holds finitely often, s is nonempty, eb holds throughout the finite prefix preceding s (this prefix is empty when $\neg b$ holds in all states), and $\neg eb$ holds throughout s.

We specify a strong fairness requirement by adding a set of axioms that name prophecy variables like eb and auxiliary variables, as shown next. Let α be an action with guard b that is executed under strong fairness, i.e., if b holds infinitely often, α is effectively executed infinitely often. Introduce an auxiliary boolean variable q and add the assignment statement

$$q := \neg q$$

to the code of α such that whenever α is executed effectively q is changed. Then the strong fairness requirement for α is

$$true \;\mapsto\; q \not\equiv Q \;\vee\; \neg eb \tag{SF}$$

where Q is a free boolean variable. Such a requirement has to be added for each strongly fair action.

To see the validity of (SF), consider two cases: (1) If b holds infinitely often, eb holds in every state; hence, (SF) requires q's value to change eventually, i.e., infinitely often. The value of q changes only through an effective execution of α; hence, α is effectively executed infinitely often if b holds infinitely often. (2) Conversely, if b holds finitely often, $\neg eb$ holds eventually forever. Hence, (SF) is satisfied; i.e., it imposes no additional requirement on the execution of α.

Next, we prove $true \;\mapsto\; x + y > k$ for the given program (page 185). The following properties can be proved directly from the program text; we leave their proofs to the reader.

$$x, q = m, Q \;\; \textbf{co} \;\; x, q = m, Q \;\vee\; (x > m \;\wedge\; q \not\equiv Q) \tag{1}$$
$$\textbf{stable}\; x \geq m \tag{2}$$
$$\textbf{stable}\; y \geq n \tag{3}$$
$$\neg b \;\wedge\; y \geq n \;\; \textbf{en} \;\; b \;\vee\; y > n \tag{4}$$

- Proof of $x = m \;\mapsto\; x > m \;\vee\; \neg eb$: (5)

$x, q = m, Q \;\; \textbf{co} \;\; x, q = m, Q \;\vee\; (x > m \;\wedge\; q \not\equiv Q)$, rewrite (1)
$true \;\mapsto\; q \not\equiv Q \;\vee\; \neg eb$, rewrite (SF)
$x, q = m, Q \mapsto\; x > m \;\vee\; \neg eb$, PSP, weaken rhs
$x = m \mapsto\; x > m \;\vee\; \neg eb$, disjunction over all Q □

- Proof of $\neg eb \wedge\; y \geq n \;\mapsto\; y > n$: (6)

$\neg b \;\wedge\; y \geq n \;\; \textbf{en} \;\; b \;\vee\; y > n$, rewrite (4)
$\neg b \;\wedge\; y \geq n \;\mapsto\; b \;\vee\; y > n$, basis rule of $\mapsto$ on above
$\textbf{stable}\; \neg eb$, definition of eb
$\neg eb \;\wedge\; \neg b \;\wedge\; y \geq n \;\mapsto\; (\neg eb \;\wedge\; b) \;\vee\; (y > n)$, PSP, weaken rhs
$\neg eb \;\wedge\; y \geq n \;\mapsto\; y > n$, definition of eb: $\neg eb \Rightarrow\; \neg b$ □

- Proof of $\mathit{true} \mapsto x + y > k$:

$x = m \mapsto x > m \lor \neg eb$, from (5)
$x, y = m, n \mapsto (x > m \land y \geq n) \lor (\neg eb \land y \geq n)$
, PSP with (3), strengthen lhs
$x, y = m, n \mapsto (x > m \land y \geq n) \lor (y > n)$
, cancellation using (6)
$x, y = m, n \mapsto (x > m \land y \geq n) \lor (x \geq m \land y > n)$
, PSP with (2)
$x, y = m, n \mapsto x + y > m + n$, weaken rhs
$\mathit{true} \mapsto x + y > k$, induction □

6.6 Theoretical Issues

In section 6.6.1, we show the existence of the "weakest predicate that *leads-to*" q, for any q, and we prove some of its properties. (These properties have been used in proving the completion rule in section 6.4.5.) We give a fixpoint characterization of this predicate in section 6.6.2. The role of the disjunction rule is examined in section 6.6.3; we show that the validity of disjunction over a finite set of predicates is derivable from the basis and transitivity rules; therefore, the main use of the disjunction rule is in its application to an infinite set of predicates.

6.6.1 wlt

For predicate q, let $wlt.q$ be the weakest predicate that *leads-to* q. The definition of *wlt* is

$$wlt.q \equiv \langle \exists\, p :\ p \mapsto q :\ p \rangle \tag{W1}$$

We show that $wlt.q$ is indeed the weakest predicate leading to q, i.e.,

- $(p \mapsto q) \equiv (p \Rightarrow wlt.q)$ (W2)

First, observe that $(p \mapsto q) \Rightarrow (p \Rightarrow wlt.q)$:

$p \mapsto q$, assume
$p \Rightarrow wlt.q$, from (W1)

Conversely, we show $(p \Rightarrow wlt.q) \Rightarrow (p \mapsto q)$:

$\langle \forall\, r :\ r \mapsto q :\ r \mapsto q \rangle$, trivially
$\langle \exists\, r :\ r \mapsto q :\ r \rangle \mapsto q$, disjunction
$wlt.q \mapsto q$, using (W1)
$p \mapsto q$, strengthen lhs with $p \Rightarrow wlt.q$ □

We can use (W2) to determine whether $p \mapsto q$ for arbitrary p and q, by computing if $p \Rightarrow wlt.q$ holds. This is the preferred method when $wlt.q$ can be computed efficiently, as in model checking of a finite state program.

Here are some special cases of (W2). Substituting $wlt.q$ for p and using $wlt.q \Rightarrow wlt.q$, we have

$$wlt.q \mapsto q \tag{W3}$$

Also, substituting q for p in (W2) and using $q \mapsto q$, we have

$$q \Rightarrow wlt.q \tag{W4}$$

Next, we show

$$\bullet\ wlt.q \wedge \neg q \ \textbf{co}\ wlt.q \tag{W5}$$

Proof: Given $p \mapsto q$, we show that there is a predicate b that satisfies

$$p \Rightarrow b \tag{B1}$$
$$b \mapsto q \tag{B2}$$
$$b \wedge \neg q \ \textbf{co}\ b \vee q \tag{B3}$$

The proof is by induction on the structure of the proof of $p \mapsto q$.

1. (basis) p **en** q
   Let b be p. Then, we show (B1, B2, B3) as follows:

   $p \Rightarrow b$, trivially, from $b \equiv p$
   $b \mapsto q$, from $p \mapsto q$ and $b \equiv p$
   $b \wedge \neg q \ \textbf{co}\ b \vee q$, from $p \wedge \neg q \ \textbf{co}\ p \vee q$ and $b \equiv p$

2. (transitivity) Suppose $p \mapsto r$ and $r \mapsto q$, for some predicate r.
   We may assume, by induction hypothesis on $p \mapsto r$, that there is a predicate br such that

   $$p \Rightarrow br\ ,\ br \mapsto r\ ,\ br \wedge \neg r \ \textbf{co}\ br \vee r$$

   Similarly, we may assume, by induction hypothesis on $r \mapsto q$, that there is a predicate bq such that

   $$r \Rightarrow bq\ ,\ bq \mapsto q\ ,\ bq \wedge \neg q \ \textbf{co}\ bq \vee q$$

   Define b to be $br \vee bq$. We now establish that b satisfies (B1), (B2), and (B3).

   (a) (Proof of B1) $p \Rightarrow b$:

   $p \Rightarrow br$, induction hypothesis
   $p \Rightarrow br \vee bq$, $br \Rightarrow br \vee bq$
   $p \Rightarrow b$, $b \equiv br \vee bq$

(b) (Proof of B2) $b \;\mapsto\; q$:

$br \;\mapsto\; r$	, given
$bq \;\mapsto\; q$	, given
$br \vee bq \;\mapsto\; r \vee q$	, disjunction
$b \;\mapsto\; r \vee q$	, $b \;\equiv\; br \vee bq$
$b \;\mapsto\; q$	, cancellation using $r \;\mapsto\; q$

(c) (Proof of B3) $b \wedge \neg q \;\; \textbf{co} \;\; b \vee q$:

$br \wedge \neg r \;\; \textbf{co} \;\; br \vee r$	, given
$br \wedge \neg bq \;\; \textbf{co} \;\; br \vee r$	, strengthen lhs using $r \Rightarrow bq$
$br \wedge \neg bq \wedge \neg q \;\; \textbf{co} \;\; br \vee r$	, strengthen lhs
$bq \wedge \neg q \;\; \textbf{co} \;\; bq \vee q$	, given
$(br \vee bq) \wedge \neg q \;\; \textbf{co} \;\; br \vee r \vee bq \vee q$	, disjunction
$(br \vee bq) \wedge \neg q \;\; \textbf{co} \;\; br \vee bq \vee q$	, in the rhs $r \;\Rightarrow\; bq$
$b \wedge \neg q \;\; \textbf{co} \;\; b \vee q$	, $b \;\equiv\; br \vee bq$

3. (disjunction) We are given $r \;\mapsto\; q$ for all r in some set S and that $p \;\equiv\; \langle \exists\; r :: \; r \in S :: \; r \rangle$. By induction hypothesis, for every r in S there is a predicate $b.r$ such that
$r \;\Rightarrow\; b.r,\; b.r \;\mapsto\; q,\; b.r \wedge \neg q \;\; \textbf{co} \;\; b.r \vee q$

Define b to be $\langle \exists\; r :: \; r \in S :: \; b.r \rangle$. For any r, $r \in S$,

(a) $p \;\Rightarrow\; b$:

$r \;\Rightarrow\; b.r$	, given
$p \;\Rightarrow\; b$	, disjunction over r in S

(b) $b \;\mapsto\; q$:

$b.r \;\mapsto\; q$	, given
$\langle \exists\; r :: \; r \in S :: \; b.r \rangle \;\mapsto\; q$	, disjunction
$b \;\mapsto\; q$	, definition of b

(c) $b \wedge \neg q \;\; \textbf{co} \;\; b \vee q$:

$b.r \wedge \neg q \;\; \textbf{co} \;\; b.r \vee q$	, given
$\langle \exists\; r :: \; r \in S :: \; b.r \wedge \neg q \;\; \textbf{co} \;\; b.r \vee q \rangle$	, disjunction
$b \wedge \neg q \;\; \textbf{co} \;\; b \vee q$	, definition of b

We establish (W5) from (B1, B2, B3), as follows. First, $wlt.q \;\mapsto\; q$, from (W3). Using (B1, B2, B3) on this property, with p as $wlt.q$ and q as q, there is a predicate w, analogous to b, such that

$$wlt.q \;\Rightarrow\; w \qquad \text{(B1}'\text{)}$$
$$w \;\mapsto\; q \qquad \text{(B2}'\text{)}$$
$$w \wedge \neg q \;\; \textbf{co} \;\; w \vee q \qquad \text{(B3}'\text{)}$$

Now

$w \;\Rightarrow\; wlt.q$, from (B2′ and W2)
$w \;\equiv\; wlt.q$, from above and B1′
$wlt.q \wedge \neg q \;\; \mathbf{co} \;\; wlt.q \vee q$, replace w by $wlt.q$ in B3′
$wlt.q \wedge \neg q \;\; \mathbf{co} \;\; wlt.q$, simplify the rhs using (W4)

This establishes (W5). □

6.6.2 A fixpoint characterization of wlt

The following fixpoint characterization of *wlt* is under weak fairness. We define a predicate transformer *we* that captures the essence of **en**, and we define *wlt* in terms of *we*. These definitions have been used in automatic verifications of finite state programs where the extreme solutions (i.e., weakest or strongest) can be computed iteratively; see Kaltenbach [104].

Predicate $we.p$ is the weakest predicate such that, starting in any state that satisfies $we.p$, p is eventually established, and $we.p$ holds until p is established. Predicate $we.p$ can be written as a disjunction of several predicates $(we.p)_t$, one predicate for each action t in a given program. Predicate $(we.p)_t$ has the same meaning as $we.p$ but has the additional requirement that p can be established by executing action t with pre-condition $(we.p)_t$. Specifically, $(we.p)_t$ is the weakest solution of (1) in q; here, $wp.s.q$ is the weakest pre-condition of s when the post-condition is q.

$$q \;\equiv\; (\langle \forall\; s ::\; wp.s.q \rangle \wedge wp.t.p) \;\vee\; p \tag{1}$$

Now predicate transformer *we* can be defined. Below, t is quantified over all actions of a program.

$$we.p \;\equiv\; \langle \exists\; t ::\; (we.p)_t \rangle \tag{2}$$

Finally, we define $wlt.q$ as the strongest solution in p of (3).

$$p \;\equiv\; q \;\vee\; we.p \tag{3}$$

The existence of the weakest solution for (1) and the strongest solution for (3) can be established by appealing to the Knaster-Tarski theorem; see [103] for details. The definition of *wlt* given by equation (3) can be shown to be the same as definition (W1) of section 6.6.1. The fixpoint characterization yields a number of properties of *we* and *wlt* (we have seen some of these earlier):

$p \;\Rightarrow\; we.p$
$wlt.p \;\equiv\; p \;\vee\; we.(wlt.p)$
$\langle (p \vee we.q) \;\Rightarrow\; q \rangle \;\Rightarrow\; \langle wlt.p \;\Rightarrow\; q \rangle$
$we.(wlt.p) \;\Rightarrow\; wlt.p$
$p \;\Rightarrow\; wlt.p$
$(p \;\Rightarrow\; q) \;\Rightarrow\; (wlt.p \;\Rightarrow\; wlt.q)$

$wlt.false \equiv false$
$wlt.(wlt.p) \equiv wlt.p$
$wlt.(p \vee wlt.q) \Rightarrow wlt.(p \vee q)$

6.6.3 The role of the disjunction rule

We have defined $\mapsto$ using three inference rules. The basis and transitivity rules are intuitively acceptable. However, the need for the disjunction rule is not easy to see. In this section, we show that the disjunction rule is (1) unnecessary for performing finite disjunctions and (2) necessary for performing infinite disjunctions. One consequence of this observation is that for a finite-state program —where the number of predicates themselves is finite, so disjunction can be performed only over a finite number of predicates— the disjunction rule is unnecessary. Therefore, finite state model checking needs only the basis and transitivity rules.

Let $\models\!\!\Rightarrow$ (a "poor cousin" of $\mapsto$) be the transitive closure of **en**; i.e., $\models\!\!\Rightarrow$ is defined by the following two inference rules.

$$\frac{p \ \textbf{en} \ q}{p \ \models\!\!\Rightarrow \ q}$$

$$\frac{p \ \models\!\!\Rightarrow \ q \ , \ q \ \models\!\!\Rightarrow \ r}{p \ \models\!\!\Rightarrow \ r}$$

We show that $\models\!\!\Rightarrow$ is finitely disjunctive, i.e.,

$$\frac{p \ \models\!\!\Rightarrow \ q \ , \ p' \ \models\!\!\Rightarrow \ q}{p \vee p' \ \models\!\!\Rightarrow \ q}$$

We first show

$$\frac{p \ \models\!\!\Rightarrow \ q}{p \vee r \ \models\!\!\Rightarrow \ q \vee r} \tag{1}$$

We use induction on the structure of the proof of $p \ \models\!\!\Rightarrow \ q$.

(basis) Assume p **en** q
$p \vee r$ **en** $q \vee r$, see Exercise (4d)
$p \vee r \ \models\!\!\Rightarrow \ q \vee r$, definition of $\models\!\!\Rightarrow$

(transitivity) Assume for some b that $p \ \models\!\!\Rightarrow \ b$ and $b \ \models\!\!\Rightarrow \ q$.
$p \vee r \ \models\!\!\Rightarrow \ b \vee r$, induction hypothesis
$b \vee r \ \models\!\!\Rightarrow \ q \vee r$, induction hypothesis
$p \vee r \ \models\!\!\Rightarrow \ q \vee r$, transitivity

• Finite disjunctivity of $\models\!\!\Rightarrow$: given $p \ \models\!\!\Rightarrow \ q$ and $p' \ \models\!\!\Rightarrow \ q$,

$p \vee p' \ \models\!\!\Rightarrow \ q \vee p'$, from the premise $p \ \models\!\!\Rightarrow \ q$ and (1), use p' as r
$q \vee p' \ \models\!\!\Rightarrow \ q$, from the premise $p' \ \models\!\!\Rightarrow \ q$ and (1), use q as r
$p \vee p' \ \models\!\!\Rightarrow \ q$, transitivity on above two

This concludes the proof that $\models\!\Rightarrow$ is finitely disjunctive. Now, we show that $\models\!\Rightarrow$ is not infinitely disjunctive.

We show that if $p \ \models\!\Rightarrow \ q$ holds in a program, there is a natural number k, depending only on p and q, such that for any state that satisfies p there is an execution of length at most k that establishes q.

If $p \ \models\!\Rightarrow \ q$ is proved by $p \ \ \mathbf{en} \ \ q$, there is an action t such that

$$\{p \ \wedge \ \neg q\} \ \ t \ \ \{q\}$$

For a state that satisfies $p \wedge q$, an empty execution sequence establishes q, and for a state that satisfies $p \wedge \neg q$, the sequence that consists of action t establishes q. Thus, the bound k equals 1 in this case. If $p \ \models\!\Rightarrow \ q$ is proved by $p \ \models\!\Rightarrow \ r$ and $r \ \models\!\Rightarrow \ q$, $k_1 + k_2$ is the required bound where k_1 is the bound for $p \ \models\!\Rightarrow \ r$ and k_2 is the bound for $r \ \models\!\Rightarrow \ q$.

The impossibility result is derived by considering a program that consists of a single action; the action decrements an integer variable x by 1. It is straightforward to show that $\mathit{true} \ \mapsto \ x < 0$ holds in this program. However, $\mathit{true} \ \models\!\Rightarrow \ x < 0$ cannot be proved. Because if it can be proved, there is a bound k associated with this proof. Then, for any state —state $x = k$, in particular— we can find an execution sequence of length at most k that establishes $x < 0$, which is impossible.

6.7 Concluding Remarks

The major theme of this chapter is a definition of *leads-to* and the promulgation of its derived rules. The definition uses the auxiliary concept of *transient predicate* (which is used to define *ensures*, which forms the basis for the definition of *leads-to*). Transient predicates are defined directly from the program text for different forms of fairness.

The manipulation rules for *leads-to* consist of about four lightweight and four heavyweight rules. The prominent ones are cancellation, PSP, and induction rules (in addition to transitivity and disjunction rules that appear in the definition of *leads-to*). The examples illustrate how these rules can be applied effectively in practice.

The minimal progress assumption can be used to prove absence of system deadlock and, in some cases, eventual program termination. But it cannot be used to prove absence of individual starvation, e.g., that a transmitted message is eventually delivered. That is why we have discussed weak fairness prominently in this chapter.

6.8 Bibliographic Notes

The treatment of progress properties in this chapter closely follows the original development described in [32]. The only new element is the introduction of transient predicates and the replacement of **unless** by **co** in the derived rules, such as in the PSP and the completion rules. Portions of this chapter appeared earlier in Misra [138]; they are reprinted here with permission from Elsevier Science.

The minimal progress condition is due to Dijkstra [57]. For a comprehensive treatment of fairness, see Francez [72] or Manna and Pnueli [127]. The definition of transient predicate is inspired by Cohen [42]. The definition of **ensures** for minimal progress appears in Jutla and Rao [102]. The definition of transient predicate under weak fairness is inspired by Lehmann, Pnueli, and Stavi's notion [122] of "helpful actions". Operator *leads-to* was introduced in Lamport [113]; its interpretation in linear temporal logic is in Owicki and Lamport [146]. Our definition, using inference rules, has facilitated the proofs of the derived rules using structural induction. Lamport [118] prescribes deducing progress properties from the conjunction of the fairness assumption —expressed as a formula in temporal logic— and the safety properties.

The graph problem in section 6.5.6 is from [134, note 2]; the original inspiration for this problem is from Chandy and Misra [29].

Chandy and Sanders [34] have proposed combining progress with stability; they write $p \hookrightarrow q$ to denote that once p holds in the program, q will eventually hold and will continue to hold thereafter. Operator $\hookrightarrow$ has many pleasing properties, including lhs strengthening, rhs weakening, infinite disjunction, and transitivity. It is particularly interesting that $\hookrightarrow$ is finitely conjunctive.

The notion of *wlt* appears in Knapp [107] and in Jutla, Knapp, and Rao [103]. The latter paper includes the fixpoint characterization of *wlt* given in section 6.6.2. Property (W5) in section 6.6.1 is due to Singh [163]. Several varieties of completeness (and incompleteness) results have been established for *leads-to*. Jutla and Rao [102] contains a thorough exposition of what completeness means in this context. They argue that relative completeness in the sense of Cook [48] is all that we can hope for. Such completeness results appear in Cohen [42], Jutla and Rao [102], Knapp [108], Pachl [148], and Rao [156, 155]. The fact that the disjunction rule in the definition of *leads-to* is unnecessary for finite-state programs is due to van de Snepscheut [167]. It has been exploited by Kaltenbach [104] in implementing a model checker for finite state programs. Operator *leads-to* has been extended to probabilistic programs in [155]. Carruth [27] shows how the progress operators can be extended for real-time programs. His extensions preserve the derived rules stated in this chapter.

The observations in section 6.3.4 are due to Will Adams. The problem in section 6.5.5 was communicated to me by Robert Blumofe, who attributes it to Maurice Herlihy. The proof given here is due to Rajeev Joshi.

Abadi and Lamport [2] introduced prophecy variables in connection with proofs of refinements. Earlier, only auxiliary variables were used in refinements (see [111]), which are inadequate in general. Prophecy variables predict the future, whereas auxiliary variables remember the past. The treatment of strong fairness in section 6.5.7 is due to Cohen.

Model checking, employing some variation of temporal logic, is one of the most important developments in the area of automatic verification; see Clarke and Emerson [39] and Quielle and Sifakis [153] for the pioneering papers on this subject and Clarke, Grumberg, and Peled [40] for a comprehensive survey of the field. Dill [64] and Kaltenbach [104] have developed efficient model checkers for finite state action systems.

Assertional reasoning about safety and liveness properties of concurrent programs, applied to a variety of synchronization and communication problems, appear in Schneider [161]. For a comprehensive survey of verification of concurrent systems see de Roever et al. [53].

6.9 Exercises

1. (Exercise due to C.S. Jutla and J.R. Rao) A program consists of the following actions. Below, p and q are boolean variables.

   $$\begin{array}{lll} \alpha :: & & p := \neg p \\ [\!] \ \beta :: & & q := \neg q \\ [\!] \ \gamma :: q & \rightarrow & p := \mathit{true} \end{array}$$

   Show that

   under minimal progress $\mathit{true} \mapsto (p \vee q)$
   under weak fairness $\mathit{true} \mapsto p$ and $\mathit{true} \mapsto q$
   under strong fairness for all actions $\mathit{true} \mapsto (p \wedge q)$

2. Show that the following property holds for the program *Fairness* of section 6.2, under both minimal progress and weak fairness assumptions. For any integer k

   $$x = k \ \wedge \ y = k \ \ \textbf{en} \ \ x + y > 2 \times k$$

3. (Simulating minimal progress by weak fairness)
   A program that consists of the following actions is executed under minimal progress.

$$\begin{array}{l} \alpha :: x := x+1 \\ [\!] \ \beta :: x := x-1 \end{array}$$

Construct a "simulator" program that operates under weak fairness and mimics the behavior of the above program. Specifically, establish a 1-1 correspondence between the execution sequences of the above program and the simulator, where two executions correspond if they assign the same sequence of values to x.

Hint: introduce a variable in the simulator that determines whether α or β is to be executed.

4. Show the following for **ensures**.

   (a) (implication) $\dfrac{p \ \Rightarrow \ q}{p \ \textbf{en} \ q}$

   (b) (rhs weakening) $\dfrac{p \ \textbf{en} \ q}{p \ \textbf{en} \ q \vee r}$

   (c) (lhs manipulation)

   $$(p \ \textbf{en} \ q) \ \equiv \ (p' \ \textbf{en} \ q)$$
   $$\text{where } p \wedge \neg q \ \Rightarrow \ p' \text{ and } p' \ \Rightarrow \ p \vee q$$

   Apply this rule to show that the following are equivalent.

   $$p \ \textbf{en} \ q \vee r \ , \ p \vee r \ \textbf{en} \ q \vee r \ , \ p \wedge \neg q \ \textbf{en} \ q \vee r$$

   (d) $\dfrac{p \ \textbf{en} \ q}{p \vee r \ \textbf{en} \ q \vee r}$

   (e) $\dfrac{p \vee q \ \textbf{en} \ r}{p \ \textbf{en} \ q \vee r}$

   (f) $(\textbf{transient} \ p) \ \equiv \ (\mathit{true} \ \textbf{en} \ \neg p)$

5. In a given program *effective* execution of *any* non-*skip* action establishes q as a post-condition given p as a pre-condition; i.e., for every non-*skip* action of the form $g_i \ \rightarrow \ s_i$,

   $$\{p \ \wedge \ g_i\} \ \ s_i \ \ \{q\}, \text{ for all } i$$

   Show that $p \ \wedge \ \langle \exists i :: \ g_i \rangle \ \mapsto \ q$ holds under minimal progress as well as weak fairness. Pay special attention to the case where the program has no non-*skip* action.

6. The following alternative definition of *leads-to* eliminates the use of the **ensures** operator; it replaces the original basis rule by the two rules shown next.

(implication) $\dfrac{p \;\Rightarrow\; q}{p \;\mapsto\; q}$

(basis) $\dfrac{p \;\; \textbf{co} \;\; p \vee q \;,\; \textbf{transient}\; p}{p \;\mapsto\; q}$

The transitivity and the disjunction rules are retained. Prove that this definition of *leads-to* is equivalent to the original definition.

7. Consider the program consisting of the actions

$$\begin{array}{|l} \alpha :: b \;\; \rightarrow \;\; x := \; x+1 \\ [\!] \;\; \beta :: \neg b \;\; \rightarrow \;\; x := \; x-1 \end{array}$$

where x is integer and b is boolean. The initial value of b is immaterial. Show that under weak fairness

$$\mathit{true} \;\mapsto\; |x| > m$$

for any integer m. Can this property be deduced if the program had an additional action

$$b := \; \neg b \; ?$$

8. In the following program, variables x, y, and z are integers. Show that $\mathit{true} \;\mapsto\; x, y = 0, 0$ under weak fairness.

$$\begin{array}{|l} x > 0 \;\; \rightarrow \;\; x, y := \; x-1, z \\ [\!] \;\; y > 0 \;\; \rightarrow \;\; y := \; y-1 \\ [\!] \;\; z := \; z+1 \end{array}$$

Can you place a bound, as a function of the initial values of x and y, on the number of assignments to x and y required to establish $x, y = 0, 0$?

9. (Working with the lightweight rules) Show for arbitrary predicates p, q, r, p', and q':

(a) $\dfrac{p \vee q \;\mapsto\; r}{p \;\mapsto\; r}$

(b) $\dfrac{p \wedge q \;\mapsto\; r}{p \;\mapsto\; \neg q \vee r}$

(c) $\dfrac{p \;\mapsto\; q \;,\; p' \;\mapsto\; q'}{p \vee p' \;\mapsto\; (q \wedge \neg p') \;\vee\; q'}$

(d) For any r where

$$p \wedge \neg q \;\Rightarrow\; r \text{ and } r \;\Rightarrow\; p \vee q, \text{ show that}$$
$$(p \;\mapsto\; q) \;\equiv\; (r \;\mapsto\; q)$$

In particular,

$$(p \;\mapsto\; q) \;\equiv\; (p \wedge \neg q \;\mapsto\; q)$$
$$(p \;\mapsto\; q) \;\equiv\; (p \vee q \;\mapsto\; q)$$

(e) $\langle \forall\, i :: p.i \;\mapsto\; q \rangle \;\equiv\; \langle \exists\, i :: p.i \rangle \;\mapsto\; q$

10. Show for arbitrary predicates p, q, and r

(a) $$\frac{p \;\mapsto\; q \;,\; \textbf{stable}\ \neg r \wedge \neg q}{p \;\Rightarrow\; q \vee r}$$

(b) $$\frac{p \;\mapsto\; q}{FP \;\Rightarrow\; (p \;\Rightarrow\; q)}$$ where FP is the fixed point predicate

(c) $$\frac{p \;\mapsto\; \neg p \;,\; \neg p \;\mapsto\; p}{\neg FP}$$

11. For programs F and G whenever p **en** q holds in F, $p \;\mapsto\; q$ holds in G, for all p and q. Show that every *leads-to* property of F is a property of G.

12. Show that boolean variable b is *true* infinitely often in the following program under the weak fairness assumption. Here x and y are integers.

$$\begin{array}{llll} & \alpha :: b & \rightarrow & x := x+1 \\ [\!] & \beta :: \neg b & \rightarrow & y := y+1 \\ [\!] & \gamma :: x \neq y & \rightarrow & b := \neg b \end{array}$$

13. Given that integer variables x and y are not changed synchronously and at least one of them is changed eventually, show that x differs from y infinitely often.

14. For a given program, (1) any step that increases x establishes q, and (2) x increases without bound. Show that q is infinitely often *true*.

15. Show counterexamples to the following:

(a) $$\frac{p \;\textbf{en}\; q \;,\; p' \;\textbf{en}\; q'}{p \vee p' \;\textbf{en}\; q \vee q'}$$

(b) $$\frac{p \wedge \neg q \;\textbf{co}\; p \vee q \;,\; p \;\mapsto\; q}{p \;\textbf{en}\; q}$$

(c) $$\frac{p \;\mapsto\; q \;,\; p' \;\mapsto\; q'}{p \wedge p' \;\mapsto\; q \wedge q'}$$

(d) $\dfrac{p \;\mapsto\; q \;,\; q \;\mapsto\; p}{p \;\equiv\; q}$

16. (Methodology in applying PSP)

    (a) The safety property, "x is nondecreasing", can be written in the following equivalent ways, using free variable m,

    stable $x \geq m$
    $x = m$ **co** $x \geq m$

    Which of these properties is preferable for applying the PSP rule?

    (b) Given $p \mapsto q$, r **co** s and r' **co** s', compute the strongest predicate b that satisfies

    $p \;\wedge\; (r \wedge r') \;\mapsto\; b$

    Does this suggest a methodology for applying the PSP rule?

17. (Induction)

    (a) Show $\dfrac{x = k \;\mapsto\; x = k - 1}{x > 0 \;\mapsto\; x = 0}$

    (b) Let $A[0..N]$ be an array of integers. Define M to be the number of *inversions* in A, i.e.,

    $$M = \langle +\; i, j :\; 0 \leq i < j \leq N \;\wedge\; A[i] > A[j] :\; 1 \rangle$$

    Show that for all m, $m > 0$:

    $$\frac{M = m \;\mapsto\; M < m}{\mathit{true} \;\mapsto\; \langle \forall\, i :\; 0 \leq i < N :\; A[i] \leq A[i+1] \rangle}$$

    (c) If x decreases eventually and y never decreases, show that $x < y$ holds infinitely often. Does **stable** $x < y$ hold?

    (d) Items are added to set s only if p holds as a pre-condition. If s is nonempty, some item is eventually removed from s. Show that eventually either s is empty or p holds.

18. For predicates p and q define p *tracks* q to mean

    1. **invariant** $p \Rightarrow q$
    2. p **co** $\neg q \vee p$
    3. $q \;\mapsto\; \neg q \vee p$

    Predicate p tracks q in the following sense:

    1. if p holds, q holds too
    2. once p holds, it continues to hold until q becomes *false* (then, from $p \Rightarrow q$, p becomes *false*)
    3. if q continues to hold, then p will hold

Show that

(a) *tracks* is a partial order (i.e., reflexive, antisymmetric, transitive relation) over predicates.

(b) $\dfrac{p \ tracks \ q, \ p' \ tracks \ q'}{p \wedge p' \ \ tracks \ q \wedge q'}$

Simplify the definition of *tracks* given **stable** q.

19. (Completion) A process is either *idle* or *active*. An idle process can become active by acquiring a set of resources. The rules for state transition and resource acquisition are the following. Let R be a fixed set of resources.

(a) If a process is idle and holds all resources in R, eventually it becomes active.

(b) A process does not release any resource r, $r \in R$, as long as it is idle.

(c) For any r, $r \in R$, if a process remains idle, it eventually holds r.

Show that a process does not stay idle forever. Note that an idle process may become active without holding all resources in R; the stated properties do not prevent this possibility.

20. Show that the completion rule does not hold for an infinite set of predicate pairs (p_i, q_i).

21. (Generalizations of the completion rule)

(a) Prove, for $0 \leq i < N$ and any set S of indices between 0 and $N-1$,

$$\frac{\langle \forall \ i : \ 0 \leq i < N : \ p_i \ \mapsto \ q_i \ , \ \textbf{stable} \ q_i \rangle}{\begin{array}{l}\langle \forall \ j : \ j \in S : \ p_j \rangle \ \mapsto \\ \qquad \langle \forall \ j : \ j \in S : \ q_j \rangle \ \wedge \ \langle \forall \ j : \ j \notin S : \ p_j \ \Rightarrow \ q_j \rangle\end{array}}$$

(b) Prove, for $0 \leq i < N$,

$$\frac{\begin{array}{c} p_i \ \mapsto \ q_i \vee r_i \\ q_i \ \textbf{co} \ q_i \vee r_i \end{array}}{\langle \forall \ i :: \ p_i \rangle \ \mapsto \ \langle \forall \ i :: \ q_i \rangle \ \vee \ \langle \exists \ i :: \ r_i \rangle}$$

Hint: Let b be $\langle \exists \ i :: \ r_i \rangle$. Weaken the rhs of the premises to get

$$\begin{array}{l} p_i \ \mapsto \ q_i \vee b \\ q_i \ \textbf{co} \ q_i \vee b \end{array}$$

Show, from these premises, that

$$\langle \forall \ i :: \ p_i \rangle \ \mapsto \ \langle \forall \ i :: \ q_i \rangle \ \vee \ b$$

This is the completion rule, stated in section 6.4.5.

(c) Propose and prove a general version combining exercises (21a) and (21b).

22. Prove that transitivity and disjunction rules in the definition of $\mapsto$ can be combined into the following single rule. For any set S of predicates,

$$\frac{p \;\mapsto\; \langle \exists\, q :: \; q \in S :: \; q \rangle \;,\; \langle \forall\, q :: \; q \in S \;:\; q \mapsto \; r \rangle}{p \;\mapsto\; r}$$

23. Show (using W1–W5 of section 6.6.1)

$$\textbf{stable}\ q \;\Rightarrow\; \textbf{stable}\ wlt.q$$

24. Show $\dfrac{p \;\mapsto\; q \;,\; \textbf{stable}\ \neg r \wedge \neg q}{p \wedge \neg q \;\; \textbf{co} \;\; q \vee r}$

Hint: Use the properties of wlt (section 6.6.1).

6.10 Solutions to Exercises

1. (Minimal progress) Prove from the program text that $\neg(p \vee q)$ is transient. Then $true \;\; \textbf{en} \;\; (p \vee q)$; hence, $true \mapsto (p \vee q)$.

   (Weak fairness) Observe that $\{\neg p\}\ \alpha\ \{p\}$. Therefore, $\neg p$ is transient; hence, $true \;\; \textbf{en} \;\; p$ and $true \mapsto p$. Similarly, $true \mapsto q$.

   (Strong fairness) Action β is executed infinitely often because its guard is *true*. Therefore, q holds infinitely often. Hence, γ is executed infinitely often in a state where q holds and $\{q\}\ \gamma\ \{p \wedge q\}$. Therefore, $p \wedge q$ holds infinitely often.

2. (Minimal progress) The proof obligations are

$$\begin{array}{l} x = k \;\wedge\; y = k \;\Rightarrow\; true \\ \{x = k \;\wedge\; y = k\} \;\; x := \; x + 1 \;\; \{x + y > 2 \times k\} \\ \{x = k \;\wedge\; y = k\} \;\; y := \; y + 1 \;\; \{x + y > 2 \times k\} \\ \{x = k \;\wedge\; y = k \;\wedge\; x \neq y\} \;\; z := \; z + 1 \;\; \{x + y > 2 \times k\} \end{array}$$

   These are easily established (the last assertion is of the form $\{false\}\ \ s\ \ \{p\}$).

   (Weak fairness) The proof obligations are

$$\begin{array}{l} x = k \;\wedge\; y = k \;\; \textbf{co} \;\; (x = k \;\wedge\; y = k) \;\vee\; (x + y > 2 \times k) \\ \langle \exists\, t :: \; \{x = k \;\wedge\; y = k\} \;\; t \;\; \{x + y > 2 \times k\} \rangle \end{array}$$

The first proof is easy. For the second proof, use either of the actions α or β for t.

3. Introduce a boolean variable b; initially, b is *true*. The program consists of

$$\begin{array}{l} \gamma :: \textbf{if } b \textbf{ then } x := x+1 \textbf{ else } x := x-1 \textbf{ endif} \\ [\!] \ \delta :: b := \neg b \end{array}$$

Given an execution of this program under weak fairness construct an execution of the program in exercise 3 under minimal progress as follows. An occurrence of γ is replaced by α if b holds, or by β if $\neg b$ holds. Every occurrence of δ is removed. Similarly, given an execution of the program in exercise 3 under minimal progress, construct an execution of the program (under weak fairness) as follows. For each α, if b holds, execute the sequence $\langle \delta;\ \delta;\ \gamma \rangle$, else execute $\langle \delta;\ \gamma \rangle$. Similarly, for β, if b holds, execute $\langle \delta;\ \gamma \rangle$, else execute $\langle \delta;\ \delta;\ \gamma \rangle$. Observe that in the resulting execution sequence both γ and δ appear infinitely often.

4. (a) See the proof of the implication rule for $\mapsto$ (section 6.4.5).

(b)
$$\begin{array}{ll} p \wedge \neg q \ \textbf{ co } \ p \vee q & \text{, from } p \ \textbf{ en } \ q \\ p \wedge \neg q \wedge \neg r \ \textbf{ co } \ p \vee q \vee r & \text{, strengthen lhs, weaken rhs} \qquad (1) \\ \textbf{transient } p \wedge \neg q & \text{, from } p \ \textbf{ en } \ q \\ \textbf{transient } p \wedge \neg q \wedge \neg r & \text{, strengthen} \\ p \ \textbf{ en } \ q \vee r & \text{, from (1) and above} \end{array}$$

(c) From $p \wedge \neg q \ \Rightarrow \ p'$ and $p' \ \Rightarrow \ p \vee q$,

$$p' \wedge \neg q \ \equiv \ p \wedge \neg q \ \text{ and } \ p' \vee q \ \equiv \ p \vee q$$

Replace $p \wedge \neg q$ and $p \vee q$ in the definition of $p \ \textbf{ en } \ q$ by $p' \wedge \neg q$ and $p' \vee q$, respectively, to get the result. The given equivalences can be proved easily using this rule.

(d)
$$\begin{array}{ll} p \ \textbf{ en } \ q \vee r & \text{, from } p \ \textbf{ en } \ q \text{ (use exercise 4b)} \\ p \vee r \ \textbf{ en } \ q \vee r & \text{, from above (use exercise 4c)} \end{array}$$

(e)
$$\begin{array}{ll} p \vee q \ \textbf{ en } \ q \vee r & \text{, weaken rhs of premise (exercise 4b)} \\ p \ \textbf{ en } \ q \vee r & \text{, from above (use exercise 4c)} \end{array}$$

(f)
$$\begin{array}{ll} & \textit{true} \ \textbf{ en } \ \neg p \\ \equiv & \{\text{definition of } \textbf{en}\} \\ & p \ \textbf{ co } \ \textit{true}, \textbf{ transient } p \\ \equiv & \{p \ \textbf{ co } \ \textit{true} \text{ holds for all } p\} \\ & \textbf{transient } p \end{array}$$

5. First, we take care of the case where the program has no non-*skip* action. The result to be proved then is $\mathit{false} \mapsto q$, which follows from the implication rule. Next, we show that $p \wedge g_i$ **en** q, for all i.

   (a) $p \wedge g_i \wedge \neg q$ **co** $(p \wedge g_i) \vee q$:
   For any action, $\alpha_j :: g_j \rightarrow s_j$, we have

   $\{p \wedge g_j\}\ s_j\ \{q\}$
   , given
   $\{(p \wedge g_i \wedge \neg q) \wedge g_j\}\ s_j\ \{(p \wedge g_i) \vee q\}$
   , strengthen lhs and weaken rhs
   $p \wedge g_i \wedge \neg q$ **co** $(p \wedge g_i) \vee q$
   , definition of **co**

   (b) **transient** $p \wedge g_i \wedge \neg q$: The proof under minimal progress is left to the reader; the proof is similar to the one in part (a). Under weak fairness, we have to show the existence of an action β_i such that

   $\{p \wedge g_i \wedge \neg q\}\ \beta_i\ \{\neg(p \wedge g_i \wedge \neg q)\}$

   We show that β_i is α_i, i.e., $g_i \rightarrow s_i$.

   $\{p \wedge g_i\}\ s_i\ \{q\}$
   , given
   $\{p \wedge g_i\}\ \alpha_i\ \{q\}$
   , α_i is $g_i \rightarrow s_i$
   $\{p \wedge g_i \wedge \neg q\}\ \alpha_i\ \{\neg(p \wedge g_i \wedge \neg q)\}$
   , strengthen lhs and weaken rhs

   Now we are ready to prove the main result, $p \wedge \langle \exists i :: g_i \rangle \mapsto q$. For all i,

   $p \wedge g_i$ **en** q , proved
   $p \wedge g_i \mapsto q$, basis rule for $\mapsto$
   $p \wedge \langle \exists i :: g_i \rangle \mapsto q$, disjunction over all i

6. The proof is by mutual implication.

   - original definition $\Rightarrow$ proposed definition:
     The implication rule is proved in section 6.4.5. The transitivity and disjunction rules follow trivially. So it remains to show, under the original definition of $\mapsto$,

     $$\frac{p \textbf{ co } p \vee q \ ,\ \textbf{transient } p}{p \mapsto q}$$

$p \wedge \neg q$ **co** $p \vee q$	, strengthen lhs of **co**-property in the premise
transient $p \wedge \neg q$	, strengthen the premise
p **en** q	, definition of **en**
$p \mapsto q$	, definition of $\mapsto$

- proposed definition $\Rightarrow$ original definition:
  We prove the basis rule of the original definition
  $$\frac{p \textbf{ en } q}{p \mapsto q}$$
  under the new interpretation for $\mapsto$ (the other two rules have been retained).

$p \wedge \neg q$ **co** $p \vee q$	, from p **en** q in the premise
transient $p \wedge \neg q$	, from p **en** q in the premise
$p \wedge \neg q \mapsto q$	, from the basis rule of the proposed definition (using $p \wedge \neg q$ for p)
$p \wedge q \mapsto q$	, from the implication rule in the proposed definition
$p \mapsto q$	, proposed definition's disjunction rule

7. For free variables k and m

$x = k \wedge b$ **en** $x > k \wedge b$	, program text	
$x = k \wedge b \mapsto x > k \wedge b$	, basis for $\mapsto$	
$b \mapsto x > m$	, induction on integers	
$b \mapsto \lvert x \rvert > m$	, weaken rhs	(1)
$\neg b \mapsto x < -m$	, similarly	
$\neg b \mapsto \lvert x \rvert > m$	, weaken rhs $(x < -m \Rightarrow -x > m)$	
$\mathit{true} \mapsto \lvert x \rvert > m$	, disjunction of above and (1)	

   This property cannot be deduced if $\gamma :: b := \neg b$ is an additional action in the program because starting in a state where $\neg b$ holds, the infinite repetition of $\langle \alpha;\ \gamma;\ \beta;\ \gamma \rangle$ never changes x though it is a fair execution.

8. For natural numbers m and n, define $(x, y) \prec (m, n)$ to mean that $x < m \vee (x = m \wedge y < n)$; i.e., (x, y) is lexicographically smaller than (m, n).

$x, y = m, n \wedge m \neq 0$ **en** $(x, y) \prec (m, n)$
, from the program text
$x, y = m, n \wedge n \neq 0$ **en** $(x, y) \prec (m, n)$
, from the program text

$x, y = m, n \ \wedge \ \neg(m, n = 0, 0) \ \mapsto \ (x, y) \prec (m, n)$
, convert above two to *leads-to*s and take disjunction
$x, y = m, n \ \wedge \ m, n = 0, 0 \ \mapsto \ (x, y) = (0, 0)$
, implication
$x, y = m, n \ \mapsto \ (x, y) \prec (m, n) \ \vee \ (x, y) = (0, 0)$
, disjunction on above two
$\mathit{true} \ \mapsto \ x, y = 0, 0$
, induction: lexicographic order over naturals is well-founded

There is no bound on the length of the execution that establishes $x, y = 0, 0$. This is because z can be assigned an arbitrarily large value, so y can be made arbitrarily large whenever x is decreased.

9. (a) Strengthen lhs of the premise to p.

(b)
$p \wedge q \ \mapsto \ r$, premise
$p \wedge \neg q \ \mapsto \ \neg q$, implication
$p \ \mapsto \ \neg q \vee r$, disjunction

(c)
$p \ \mapsto \ q$, premise
$p \ \mapsto \ (q \wedge \neg p') \ \vee \ p'$, weaken the rhs to $q \vee p'$
$p' \ \mapsto \ q'$, premise
$p \ \mapsto \ (q \wedge \neg p') \ \vee \ q'$, cancellation on above two
$p \ \vee \ p' \ \mapsto \ (q \wedge \neg p') \ \vee \ q'$
, disjunction on above two

(d) The proof is by mutual derivation.

Proof of $(p \ \mapsto \ q) \ \Rightarrow \ (r \ \mapsto \ q)$:
$p \ \mapsto \ q$, premise
$q \ \mapsto \ q$, implication
$p \vee q \ \mapsto \ q$, disjunction
$r \ \mapsto \ q$, strengthen lhs (use $r \ \Rightarrow \ p \vee q$)

Proof of $(r \ \mapsto \ q) \ \Rightarrow \ (p \ \mapsto \ q)$: From
$(p \wedge \neg q \ \Rightarrow \ r)$ and $(r \ \Rightarrow \ p \vee q)$
we deduce
$(r \wedge \neg q \ \Rightarrow \ p)$ and $(p \ \Rightarrow \ r \vee q)$

The result follows from the first proof, interchanging the roles of p and r.

(e)
$\langle \forall \ i :: \ p_i \ \mapsto \ q \rangle$, assume
$\langle \exists \ i :: \ p_i \rangle \ \mapsto \ q$, disjunction

Conversely,
$\langle \exists \ i :: \ p_i \rangle \ \mapsto \ q$, premise
$\langle \forall \ j :: \ p_j \ \mapsto \ q \rangle$, strengthen lhs

10. (a)

$p \mapsto q$	, premise
stable $\neg r \wedge \neg q$	, premise
$p \wedge \neg r \wedge \neg q \mapsto \mathit{false}$	, stable conjunction
$\neg(p \wedge \neg r \wedge \neg q)$	, impossibility
$p \Rightarrow q \vee r$	, simplify

(b)

$p \mapsto q$	, premise
stable $FP \wedge \neg q$	, stability at fixed point (section 5.3.2)
$FP \Rightarrow (p \Rightarrow q)$	, proof similar to exercise (10a)

(c)

$FP \Rightarrow (p \Rightarrow \neg p)$	, from exercise (10b), use $p \mapsto \neg p$
$FP \Rightarrow \neg p$	, simplify
$FP \Rightarrow p$	, similarly, from $\neg p \mapsto p$
$FP \Rightarrow \mathit{false}$	, conjunction of the above two
$\neg FP$	, predicate calculus

11. Consider the structure of each *leads-to* property in F. If it is an **en** property, it is a *leads-to* property in G. Otherwise, apply induction on the structure of the *leads-to* property in F to show that it holds in G.

12. Prove (1, 2, 3) from the program text (prove corresponding **ensures** properties).

$$x < y \mapsto b \quad (1)$$
$$x = y \mapsto x < y \vee b \quad (2)$$
$$x > y \mapsto x \leq y \vee b \quad (3)$$

Now,

$x = y \mapsto b$	, cancel in rhs of (2) using (1)
$x \leq y \mapsto b$	, disjunction of above and (1)
$x > y \mapsto b$	, cancel in rhs of (3) using above
$\mathit{true} \mapsto b$	, disjunction on above two

13. Variables x and y are not changed synchronously, and at least one of them is changed eventually. So for free variables m and n,

$$x, y = m, n \ \textbf{co} \ x = m \vee y = n \quad (1)$$
$$x, y = m, n \mapsto x \neq m \vee y \neq n \quad (2)$$

- Proof of $\mathit{true} \mapsto x \neq y$:

$x, y = m, n \mapsto (x = m \wedge y \neq n) \vee (x \neq m \wedge y = n)$	, PSP on (1,2)
$x, y = m, m \mapsto (x = m \wedge y \neq m) \vee (x \neq m \wedge y = m)$	, replace n by m

$x, y = m, m \;\mapsto\; x \neq y$	, weaken rhs
$x = y \;\mapsto\; x \neq y$	, disjunction over all m
$x \neq y \;\mapsto\; x \neq y$	, implication
$\mathit{true} \;\mapsto\; x \neq y$	, disjunction of above two

14. For any m,

$x \leq m \;\; \mathbf{co} \;\; (x > m \Rightarrow q)$	, given safety property
$\mathit{true} \;\mapsto\; x > m$	, given progress property
$x \leq m \;\mapsto\; x > m \;\wedge\; q$	, PSP
$x = m \;\mapsto\; q$	, strengthen lhs, weaken rhs
$\mathit{true} \;\mapsto\; q$	, disjunction over all m

15. (a) This solution is due to Roland Backhouse. In the following program x is an integer variable.

$$\begin{array}{l} \alpha :: x > 0 \;\rightarrow\; x := 0 \\ [\!] \;\; \beta :: x < 0 \;\rightarrow\; x := 0 \end{array}$$

We can show $x > 0 \;\; \mathbf{en} \;\; x = 0$ and $x < 0 \;\; \mathbf{en} \;\; x = 0$. But, we cannot show $x \neq 0 \;\; \mathbf{en} \;\; x = 0$, because there is no action t such that $\{x \neq 0\} \;\; t \;\; \{x = 0\}$.

(b) In the program of exercise (15a) with

$$p \;\equiv\; x \neq 0 \;\text{ and }\; q \;\equiv\; x = 0$$

we can show that $p \wedge \neg q \;\; \mathbf{co} \;\; p \vee q$ (i.e., $x \neq 0 \;\; \mathbf{co} \;\; \mathit{true}$) holds. Also, $p \;\mapsto\; q$ holds because

$x > 0 \;\mapsto\; x = 0$	, from $x > 0 \;\; \mathbf{en} \;\; x = 0$
$x < 0 \;\mapsto\; x = 0$	, from $x < 0 \;\; \mathbf{en} \;\; x = 0$
$x \neq 0 \;\mapsto\; x = 0$	, disjunction
$p \;\mapsto\; q$	, definition of p and q

But as we have shown in exercise 15a, $p \;\; \mathbf{en} \;\; q$ cannot be proved.

(c) Consider a program that has a single boolean variable b that is initially *true*. There is a single action $b := \neg b$. Then

$b \;\mapsto\; \neg b$	, from $b \;\; \mathbf{en} \;\; \neg b$
$b \;\mapsto\; b$	, implication
$b \;\mapsto\; \mathit{false}$	, apply the alleged inference rule
$\neg b$	, impossibility

The conclusion is invalid since it asserts that $\neg b$ is invariant, which contradicts that b holds initially.

(d) In the program shown in exercise (15c), $b \mapsto \neg b$ and $\neg b \mapsto b$. Yet the conclusion, $b \equiv \neg b$, is invalid.

16. (a) Let us attempt applying the PSP rule on $p \mapsto q$ and each of the given properties. The first property, **stable** $x \geq m$, yields

$$p \wedge x \geq m \mapsto q \wedge x \geq m \tag{1}$$

and the second property, $x = m$ **co** $x \geq m$, yields

$$p \wedge x \geq m \mapsto (q \wedge x = m) \vee x > m \tag{2}$$

We can deduce (2) from (1) by weakening the rhs of the latter. However, we cannot deduce (1) from (2). Therefore, the first property is preferable for application of PSP.

(b) We consider two strategies.

i. Apply PSP twice:

$p \mapsto q$, premise
r **co** s , premise
$p \wedge s \mapsto (q \wedge r) \vee (\neg r \wedge s)$, PSP
r' **co** s' , premise
$p \wedge s \wedge s' \mapsto$
$(q \wedge r \wedge r') \vee (\neg r \wedge s \wedge r') \vee (\neg r' \wedge s')$
, PSP

ii. Conjoin the **co** properties and then apply PSP:

r **co** s , premise
r' **co** s' , premise
$r \wedge r'$ **co** $s \wedge s'$, conjunction
$p \mapsto q$, premise
$p \wedge s \wedge s' \mapsto$
$(q \wedge r \wedge r') \vee (\neg r \wedge s \wedge s') \vee (\neg r' \wedge s \wedge s')$
, PSP
$p \wedge s \wedge s' \mapsto$
$(q \wedge r \wedge r') \vee (\neg r \wedge s \wedge r')$
$\vee (\neg r \wedge s \wedge \neg r' \wedge s') \vee (\neg r' \wedge s \wedge s')$
, expand second disjunct using $r' \Rightarrow s'$
$p \wedge s \wedge s' \mapsto$
$(q \wedge r \wedge r') \vee (\neg r \wedge s \wedge r') \vee (\neg r' \wedge s \wedge s')$
, combine the last two disjuncts

The rhs of the conclusion in (b-ii) is stronger than the rhs of the conclusion in (b-i). This suggests that safety properties be computed as far as possible before applying PSP.

17. (a)

$x = k \mapsto x = k - 1$, premise
$x = k \wedge k > 0 \mapsto x = k - 1 \wedge k > 0$
, stable conjunction with $k > 0$

$$
\begin{array}{ll}
x > 0 \ \wedge \ x = k \ \mapsto \ (x > 0 \ \wedge \ x < k) \ \vee \ x = 0 & \\
\qquad\qquad & \text{, rewrite lhs and rhs} \\
x > 0 \ \mapsto \ x = 0 & \text{, induction over integers (use} \\
\qquad\qquad & \quad \text{the range, } k > 0)
\end{array}
$$

(b)
$$
\begin{array}{ll}
\langle \forall \ m : \ m > 0 : \ M = m \ \mapsto \ M < m \rangle & \\
\qquad\qquad & \text{, premise} \\
\mathit{true} \ \mapsto \ M \leq 0 & \text{, induction over integers}
\end{array}
$$

$$
\begin{array}{ll}
M \leq 0 & \\
\equiv \langle + \ i, j : \ 0 \leq i < j \leq N \ \wedge \ A[i] > A[j] : \ 1 \rangle \leq 0 & \\
\equiv \langle \forall \ i, j : \ 0 \leq i < j \leq N \ \wedge \ A[i] \leq A[j] \rangle & \\
\Rightarrow \quad \{\text{instantiating } j = i + 1\} & \\
\quad \langle \forall \ i : \ 0 \leq i < N : \ A[i] \leq A[i+1] \rangle &
\end{array}
$$

(c) For any m and n

$$
\begin{array}{ll}
x = m \ \mapsto \ x < m & \text{, premise} \\
\mathit{true} \ \mapsto \ x < n & \text{, induction over integers} \\
\textbf{stable} \ y \geq n & \text{, premise} \\
y \geq n \ \mapsto \ x < n \ \wedge \ y \geq n & \text{, stable conjunction} \\
y = n \ \mapsto \ x < y & \text{, strengthen lhs, weaken rhs} \\
\mathit{true} \ \mapsto \ x < y & \text{, disjunction over } n
\end{array}
$$

To see that $x < y$ is not stable: even though x is decreased eventually, say from m to n, x may be increased beyond m and (beyond y) some time between these two points in the computation.

(d) For an arbitrary set S:

$$
\begin{array}{ll}
\neg p \ \wedge \ s = S \ \ \textbf{co} \ \ s \subseteq S & \text{, premise} \\
s = S \ \ \wedge \ s \neq \emptyset \ \mapsto \ S - s \neq \emptyset & \text{, premise} \\
\neg p \ \wedge \ s = S \ \wedge \ s \neq \emptyset \ \mapsto & \\
\quad (s \subseteq S \ \wedge \ S - s \neq \emptyset) & \\
\quad \vee (p \ \wedge \ s \subseteq S) \ \vee \ s \subset S & \text{, PSP} \\
\neg p \ \wedge \ s = S \ \wedge \ s \neq \emptyset \ \mapsto \ s \subset S \ \vee \ p & \text{, weaken rhs} \\
p \ \wedge \ s = S \ \wedge \ s \neq \emptyset \ \mapsto \ p & \text{, implication} \\
s = S \ \wedge \ s = \emptyset \ \mapsto \ s = \emptyset & \text{, implication} \\
s = S \ \mapsto \ s \subset S \ \vee \ s = \emptyset \ \vee \ p & \text{, disjunction (above 3)} \\
\mathit{true} \ \mapsto \ s = \emptyset \ \vee \ p & \text{, induction}
\end{array}
$$

18. The proof is left to the reader. See Misra [134, note 30].

19. We write the proof for a generic process; the identity of the process is not shown. Use *idle* and $\neg$*idle* to denote that the process is idle and active, respectively. Let H denote the set of resources that the process holds.

    In the following, r is an arbitrary element of R.

$idle \ \wedge \ R \subseteq H \ \mapsto \ \neg idle$ (a)
$idle \ \wedge \ r \in H \ \ \mathbf{co} \ \neg idle \ \vee \ r \in H$ (b)
$idle \ \mapsto \ \neg idle \ \vee \ r \in H$ (c)

We have to show
$true \ \mapsto \ \neg idle$

Proof: Observe that the rhs of (b) and (c) can be written as
$(idle \ \wedge \ r \in H) \ \vee \ \neg idle$

Apply completion on (b) and (c) over all r, $r \in R$:
$idle \ \mapsto \ \langle \forall \ r : \ r \in R : \ idle \ \wedge \ r \in H \rangle \ \vee \ \neg idle$
$idle \ \mapsto \ (idle \ \wedge \ R \subseteq H) \ \vee \ \neg idle$
, simplify
$idle \ \mapsto \ \neg idle$, cancellation with above and (a)
$true \ \mapsto \ \neg idle$, disjunction with $\neg idle \ \mapsto \ \neg idle$

20. Consider a program that has an integer variable x and a single action $x := \ x + 1$. For any integer k, $k \geq 0$,

$x = 0 \ \mapsto \ x > k$
stable $x > k$

Yet $\langle \wedge \ k :: \ x = 0 \rangle \ \mapsto \ \langle \wedge \ k :: \ x > k \rangle$, which is equivalent to $x = 0 \ \mapsto \ false$, is invalid.

21. (a) The following proof is due to Ernie Cohen.

For k, $0 \leq k \leq N$, let A_k be $\langle \forall \ i : \ 0 \leq i < k : \ p_i \ \Rightarrow \ q_i \rangle$. We show for all k, $0 \leq k \leq N$, $true \ \mapsto \ A_k$. The proof is by induction on k.

Case $k = 0$**:** $true \ \mapsto \ A_0$, trivially from $A_0 \ \equiv \ true$.
Case $k + 1, 0 \leq k < N$**:**
$true$
$\mapsto$ {induction hypothesis}
A_k
$\mapsto$ {weaken}
$(A_k \wedge \neg p_k) \ \vee \ p_k$
$\mapsto$ {cancellation using $p_k \ \mapsto \ q_k$}
$(A_k \wedge \neg p_k) \ \vee \ q_k$
$\mapsto$ {conjunction of $true \ \mapsto \ A_k$ and **stable** q_k yields $q_k \ \mapsto \ A_k \wedge q_k$. Apply cancellation.}
$(A_k \wedge \neg p_k) \ \vee \ (A_k \wedge q_k)$
$\equiv$ {definition of A_{k+1}}
A_{k+1}

Now we prove the desired result. Henceforth, $0 \leq i < N$ unless the quantification of i is shown explicitly.

$true \;\mapsto\; \langle \forall\, i ::\; p_i \;\Rightarrow\; q_i \rangle$
, from $true \;\mapsto\; A_k$, setting $k = N$
stable $\langle \forall\, i :\; i \in S :\; q_i \rangle$
, stable conjunction of the premise
$\langle \forall\, i :\; i \in S :\; q_i \rangle \;\mapsto$
$\langle \forall\, i ::\; p_i \;\Rightarrow\; q_i \rangle \;\wedge\; \langle \forall\, i :\; i \in S :\; q_i \rangle$
, stable conjunction of the above two
$\langle \forall\, i :\; i \in S :\; p_i \rangle \;\mapsto\; \langle \forall\, i :\; i \in S :\; q_i \rangle$
, completion rule on the premises
$\langle \forall\, i :\; i \in S :\; p_i \rangle \;\mapsto$
$\langle \forall\, i ::\; p_i \;\Rightarrow\; q_i \rangle \;\wedge \langle \forall\, i :\; i \in S :\; q_i \rangle$
, transitivity on the above two
$\langle \forall\, i :\; i \in S :\; p_i \rangle \;\mapsto$
$\langle \forall\, i :\; i \notin S :\; p_i \;\Rightarrow\; q_i \rangle \;\wedge\; \langle \forall\, i :\; i \in S :\; q_i \rangle$
, rewrite rhs

(b) Henceforth, $0 \leq i \leq N$ unless the quantification of i is shown explicitly. As suggested in the hint, let b be $\langle \exists\, i ::\; r_i \rangle$. We show that

$$\langle \forall\, i ::\; p_i \rangle \;\mapsto\; \langle \forall\, i ::\; q_i \rangle \;\vee\; b$$

This is the completion rule, stated in section 6.4.5. We use the same notation and the proof structure as in that proof.

Case $N = 0$**:** We have to show $true \;\mapsto\; true$, which follows from the implication rule for $\mapsto$.

Case $N + 1$**:** Writing P for $\langle \forall\, i :\; 0 \leq i < N :\; p_i \rangle$ and Q for $\langle \forall\, i :\; 0 \leq i < N :\; q_i \rangle$, we have to show

$$P \wedge p_N \;\mapsto\; (Q \wedge q_N) \;\vee\; b\ .$$

Rewriting the premises for i, $0 \leq i \leq N$,

$$p_i \;\mapsto\; q_i \vee b \tag{1}$$
$$q_i \;\;\mathbf{co}\;\; q_i \vee b \tag{2}$$

Using the facts about *wlt* (section 6.6.1), and the given premise $p_N \;\mapsto\; q_N \;\vee\; b$, there is a predicate r (r is $wlt.q_N$) such that

$$r \;\mapsto\; q_N \vee b \tag{3}$$
$$p_N \;\Rightarrow\; r \tag{4}$$
$$(q_N \vee b) \;\Rightarrow\; r \tag{5}$$
$$r \;\wedge\; \neg(q_N \vee b) \;\;\mathbf{co}\;\; r \tag{6}$$

By the induction hypothesis
$P \;\mapsto\; Q \vee b$
The proof of $P \wedge p_N \;\mapsto\; (Q \wedge q_N) \;\vee\; b$ is as follows.

$$
\begin{array}{lll}
& P \wedge p_N & \\
\Rightarrow & \{4\} & \\
& P \wedge r & \\
\mapsto & \{q_N \ \ \mathbf{co} \ \ q_N \vee b & \text{, from (2)} \\
& \ \ r \wedge q_N \wedge \neg b \ \ \mathbf{co} \ \ q_N \vee b & \text{, strengthen lhs} \\
& \ \ r \wedge \neg q_N \wedge \neg b \ \ \mathbf{co} \ \ r & \text{, from (6)} \\
& \ \ r \wedge \neg b \ \ \mathbf{co} \ \ r \vee q_N \vee b & \text{, disjunction} \\
& \ \ r \wedge \neg b \ \ \mathbf{co} \ \ r & \text{, } (q_N \vee b) \Rightarrow r \text{ from (5)} \\
& \ \ P \ \mapsto \ Q \vee b & \text{, induction hypothesis} \\
& \ \ P \wedge r \wedge \neg b \ \mapsto \ (Q \wedge r) \ \vee \ b & \text{, PSP, weaken rhs} \\
& \ \ P \wedge r \wedge b \ \mapsto \ b & \text{, implication} \\
& \ \ P \wedge r \ \mapsto \ (Q \wedge r) \ \vee \ b & \text{, disjunction} \\
& \ \} & \\
& (Q \wedge r) \ \vee \ b & \\
\mapsto & \{r \ \mapsto \ q_N \ \vee \ b & \text{, from (3)} \\
& \ \ Q \ \ \mathbf{co} \ \ Q \vee b & \text{, conjoin (2), } 0 \leq i < N \\
& \ \ Q \wedge r \ \mapsto \ (Q \wedge q_N) \ \vee \ b & \text{, PSP} \\
& \ \ b \ \mapsto \ b & \text{, implication} \\
& \ \ (Q \wedge r) \ \vee \ b \ \mapsto \ (Q \wedge q_N) \ \vee \ b & \text{, disjunction: above two} \\
& \ \} & \\
& (Q \wedge q_N) \ \vee \ b &
\end{array}
$$

(c)
$$
\frac{\begin{array}{c} p_i \ \mapsto \ q_i \ \vee \ r_i \\ q_i \ \ \mathbf{co} \ \ q_i \ \vee \ r_i \end{array}}{\begin{array}{l} \langle \forall \ i : \ i \in S : \ p_i \rangle \ \mapsto \ (\langle \forall \ i : \ i \in S : \ q_i \rangle \ \wedge \\ \qquad \langle \forall \ i : \ i \notin S : \ p_i \ \Rightarrow \ q_i \rangle) \ \vee \ \langle \exists \ i :: \ r_i \rangle \end{array}}
$$

The proof is left to the reader.

22. Introduce a new symbol $\rightsquigarrow$ for the *leads-to* defined in the exercise.

- First, we show that

  $p \ \rightsquigarrow \ q \ \Rightarrow \ p \ \mapsto \ q$

  The proof is by induction on the structure of the proof of $p \ \rightsquigarrow \ q$.

  1. $p \ \rightsquigarrow \ q$ is proved by $p \ \ \mathbf{en} \ \ q$: then, from the basis rule, $p \ \mapsto \ q$.
  2. $p \ \rightsquigarrow \ q$ is proved by displaying a set S of predicates where
     $p \ \rightsquigarrow \ \langle \exists \ r : \ r \in S : \ r \rangle$ and $\langle \forall \ r : \ r \in S : \ r \ \rightsquigarrow \ q \rangle$

  Now,

  $$
  \begin{array}{lll}
  & \langle \forall \ r : \ r \in S : \ r \ \mapsto \ q \rangle & \text{, induction hypothesis} \\
  & \langle \exists \ r : \ r \in S : \ r \rangle \ \mapsto \ q & \text{, disjunction rule for } \mapsto \\
  & p \ \mapsto \ \langle \exists \ r : \ r \in S : \ r \rangle & \text{, induction hypothesis} \\
  & p \ \mapsto \ q & \text{, transitivity of } \mapsto \text{ on above two}
  \end{array}
  $$

- The remaining proof obligation is

  $$p \;\mapsto\; q \;\Rightarrow\; p \;\leadsto\; q$$

  which may be proved by induction on the structure of the proof of $p \;\mapsto\; q$.

23.

$wlt.q \wedge \neg q$ **co** $wlt.q$	, (W5) of section 6.6.1
q **co** q	, premise
$wlt.q \vee q$ **co** $wlt.q \vee q$	, disjunction
stable $wlt.q$	, $q \;\Rightarrow\; wlt.q$ (W4 of section 6.6.1)

24.

$wlt.q \;\mapsto\; q$	, from (W3) of section 6.6.1	
stable $\neg r \wedge \neg q$	, premise	
$wlt.q \;\Rightarrow\; q \vee r$	, use $wlt.q$ for p in exercise (10a)	(1)
$p \;\Rightarrow\; wlt.q$	, use $p \mapsto q$ and (W2) (section 6.6.1)	(2)
$wlt.q \wedge \neg q$ **co** $wlt.q$	, from (W5) of section 6.6.1	
$p \wedge \neg q$ **co** $wlt.q$	, strengthen lhs using (2)	
$p \wedge \neg q$ **co** $q \vee r$	, weaken rhs using (1)	

7
Maximality Properties

7.1 Introduction

Traditionally, a program specification is given by *safety* and *progress* properties, as explained in chapters 5 and 6. A safety property —e.g., no two neighbors eat simultaneously in a dining philosophers solution— is used to exclude certain undesirable execution sequences. A specification with safety properties alone can be implemented by a program that does nothing; that is, the safety constraints are implemented by excluding all nontrivial executions. Therefore, it is necessary to specify progress properties —e.g., some *hungry* philosopher eats eventually— which guarantee that some execution sequence will be included. Safety and progress requirements are sufficient for specifying nontrivial sequential programming tasks, but they are not sufficient for concurrent program design because, for instance, the dining philosophers solution may allow only one philosopher to eat at a time, thus eliminating all concurrency. We propose a new class of properties, called *maximality*, to permit inclusion of *all* executions that satisfy a specification. Thus, the sequential solution to the dining philosophers problem can be excluded by requiring that the solution be maximal for the appropriate specification.

Typically, program design involves constructing a program P that *implements* a given specification S; that is, the set $|P|$ of executions of P is a subset of the set $|S|$ of executions that satisfy S. For instance, given a specification to generate an infinite sequence of natural numbers, a program that generates a sequence of zeroes implements the specification. So

does the program that generates the natural numbers in order. In many cases, we seek a program P that not only implements S —i.e., $|P| \subseteq |S|$— but for which $|P| = |S|$. Then every execution that satisfies specification S is a possible execution of P; we call P *maximal* for specification S. For example, the program that generates a stream of zeroes is *not* maximal for the specification to generate an infinite sequence of natural numbers, nor is the solution that allows only one philosopher to eat at a time.

There are at least three reasons why we are interested in maximal solutions. First, as remarked above, we exploit maximality to eliminate undesirable solutions for a given specification, ones that restrict concurrency, for instance. With an appropriate specification, a maximal solution admits maximal concurrency.

Second, we often simulate an artifact by a program, and the latter has to simulate all behaviors of the former; in this case, the simulation program has to be maximal for the specification of the artifact. In sections 4.1.6 and 7.6, we consider a faulty channel that may lose, duplicate or reorder messages sent along it, as is the case in the alternating bit protocol [159]. To study the correctness of senders and receivers that communicate over such channels, the program that simulates this device has to be maximal. For instance, to apply model checking [41] for the verification of this protocol, it has to be shown that the programs that describe the sender, the receiver, and the faulty channel satisfy certain properties when they are composed together. For such a proof, we require that the program used for the faulty channel be maximal for its specification.

The third reason for designing a maximal solution is that we often develop (and prove correct) such a solution and then refine it —by eliminating a certain degree of nondeterminism, for instance— to obtain a program that is actually implemented. This strategy may be easier than developing the implemented program directly. A single maximal program for a problem may be the basis for a family of interrelated programs, each of which may be appropriate for a different computing platform. In section 11.3, we consider the problem of task scheduling in Seuss; we show a maximal solution and several refinements of it.

Typically, a maximal solution is nondeterministic; in many cases, the nondeterminism is unbounded.

Overview of the chapter

In this chapter, we suggest a method for proving the maximality of a program with respect to a given specification. Given a program P that is to be proved maximal with respect to a specification, we have to show that each sequence σ of states meeting the specification arises in a possible execution of the program. We first construct a *constrained* program P' from P and σ; the constrained program retains the structure of P, but its actions are restricted by guards and augmented by assignments to certain auxiliary

variables. Next, we show that *all* fair executions of P' produce σ and that any fair execution of P' corresponds to a fair execution of P; hence, σ is a possible sequence of states in an execution of P.

Even though we prove facts about possible executions of programs, there is no need to appeal to branching-time logics; we employ the logic developed in chapters 5 and 6. The method seems to be quite effective in practice, resulting in concise proofs for nontrivial examples such as the fair unordered channel of section 7.5 and the task scheduler of section 11.3. The proposed proof method may also serve as a guide in constructing maximal programs from specifications.

7.2 Notion of Maximality

To illustrate the notion of maximality, consider a box in which a method *fnat* returns a natural number in each call and a positive natural number eventually. The solution given below is maximal in the sense that any sequence of natural numbers that obey these constraints may be returned by the method *fnat*.

---

box *FairNatural*
  integer $n = 0$;
  total action :: $n := n + 1$
  total method *fnat*(x: integer):: $x, n := n, 0$
end {*FairNatural*}

The action system corresponding to this box is given below. In the program below, α corresponds to the total action and β to method *fnat*.

---

program *FairNatural*
  integer x;
  integer $n = 0$;

    α:: $n := n + 1$
  [] β:: $x, n := n, 0$
end {*FairNatural*}

The safety and progress proof obligations for this program are that x is assigned only non-negative numbers as values and x is infinitely often positive; i.e.,

invariant $x \geq 0$
true $\mapsto$ $x > 0$

These properties are proved easily using the methods of chapters 5 and 6. Next, we argue that the given program is maximal with respect to these two properties; a formal proof appears in section 7.3.4.

Given an infinite sequence of natural numbers X, where X contains nonzero elements infinitely often, we show an execution of *FairNatural* such that x takes on successive values from X. Consider the following sequence of actions, where X_j is the item j of X, $j \geq 0$:

$$\alpha^{X_0}\beta\,\alpha^{X_1}\beta\ \ldots\alpha^{X_j}\beta\ \ldots, \text{ for all } j,\ j \geq 0$$

Here, α^{X_j} stands for X_j repetitions of α; if $X_j = 0$, then α^{X_j} is an empty sequence. We claim that (1) this sequence of actions corresponds to a fair execution of the program, in the sense that both α and β appear infinitely often in this sequence, and (2) the sequence of values assigned to x is X. To see the first claim, observe that X contains an infinite number of nonzero elements (from the specification), and that each nonzero element contributes at least one α to the sequence; there are an infinite number of βs by construction. For the second claim, note that $\alpha^{X_j}\beta$ sets x to X_j, for each j.

Henceforth, we use the action system model of chapter 2 in developing the theory of maximality.

7.2.1 Definition of maximality

Given a program P and specification S we can show that P *satisfies* S (i.e., P has all the properties in S) using the logic in chapters 5 and 6. To prove maximality, we have to show that any sequence that satisfies S may be obtained from an execution of P in the sense described below. First, we define when an infinite sequence σ of states, $\sigma = \sigma_0, \sigma_1, \ldots$, satisfies S. We consider only the following types of properties in S: **initially** p, p **co** q and $p \mapsto q$. In the following description, $p(\sigma_i)$ means that predicate p holds in state σ_i.

σ satisfies	**initially** p	means	$p(\sigma_0)$ holds.
σ satisfies	p **co** q	means	$\langle \forall i :: p(\sigma_i) \Rightarrow q(\sigma_{i+1}) \rangle$.
σ satisfies	$p \mapsto q$	means	$\langle \forall i :: (\exists j :\ i \leq j :\ p(\sigma_i) \Rightarrow q(\sigma_j)) \rangle$.

Sequence σ satisfies S if it satisfies each property in S, as described above.

Definition of program execution A program *execution* is an infinite sequence of the form $\tau_0 A_0 \tau_1 \ldots \tau_i A_i \tau_{i+1} \ldots$ where each τ_i is a program state and A_i is an action; τ_0 satisfies the initial condition and $(\tau_i, \tau_{i+1}) \in A_i$. (Recall that action A_i is a binary relation over states.) Each execution satisfies the following fairness requirement: each action appears infinitely often in an execution. □

Let δ be an execution of a program, V a subset of program variables and σ an infinite sequence of states over V (a state over V is an assignment of values to the variables in V). The *projection* of δ over V is the (infinite) sequence of states obtained by constructing from each δ_i a state in which the values assigned only to the variables in V are retained. Henceforth, we say that δ *reduces* to σ over V if the projection of δ over V is equivalent to σ up to finite stuttering. That is, if we remove the action names from δ, retain the values only in variables V in each state and remove some number of states from each finite segment of repeating states, then we get σ. For program *FairNatural*, the fragment of the execution sequence

$$(1,0)\alpha(1,1)\alpha(1,2)\beta(2,0)\alpha(2,1)\beta(1,0)$$

where each state is a pair of values of (x, n), reduces to (finite) sequence $\langle 121 \rangle$ and also to $\langle 1121 \rangle$ over the set of variables $\{x\}$.

Definition of *maximal* Program P is *maximal* for specification S with respect to V provided that P satisfies S and for any infinite sequence σ that satisfies S some execution δ of P *reduces to* σ over V. □

Henceforth, whenever V is understood we omit any mention of it; we write "δ *reduces to* σ" and "P is maximal for specification S".

7.3 Proving Maximality

7.3.1 Constrained program

Let σ be a sequence of states that satisfies S; we have to show that some execution of P reduces to σ. The strategy is to construct a *constrained* program P' such that *all* executions of P' reduce to σ and each execution of P' corresponds to some fair execution of P in the sense to be defined in section 7.3.2.

The constrained program P' is constructed from P as follows.

1. Variables of P are retained in P'; they are called *original* variables.
2. New variables, called *chronicles*, are introduced in P'. Chronicles are like history variables: they encode the given state sequence σ. They are not altered in the constrained program; their values are only read. There may be several chronicles, one corresponding to each variable of P, to encode the sequence of values that a variable assumes in an execution.
3. Auxiliary variables are introduced in P'. In our examples, we use a special auxiliary variable, which we call a *point*, to show the position in the chronicles that matches the current state of P'. Auxiliary variables may be read and written in P'.

4. An action α of P is modified to

$$\alpha' :: \quad g \ \rightarrow \ \alpha \ ; \ \beta$$

where g is a guard that may name any variable of P' and β, which is optional, may assign only to the auxiliary variables. Action α' is an *augmented* action corresponding to α and g is the *augmenting* guard of α'. Augmenting an action may eliminate some of the executions of P. If α is a guarded command of the form $h \ \rightarrow \ \gamma$, then α' is $g \wedge h \ \rightarrow \ \gamma; \beta$.

5. Constrained program P' may also include additional actions of the form $g \ \rightarrow \ \beta$, where g names any variable of P' and β assigns only to the auxiliary variables.

6. Initialization in P' assigns the same values to the original variables as in P. Additionally, auxiliary variables and other variables of P (that are not initialized in P) may be assigned values.

In summary, in P' no assignment is made to the chronicles. Auxiliary variables appear only in guards, tests, and assignments to themselves. Therefore, auxiliary variables do not affect the values of the original variables. Original variables of P are assigned values exactly as they were assigned in P, except that some of the variables that were uninitialized in P may be initialized in P'. We will show that the sequence of values assumed by the original variables in P' is the same as the chronicle and each fair execution of P' corresponds to some (fair) execution of P.

Example

Consider program *FairNatural* of section 7.2.1. To prove its maximality for the specification

invariant $x \geq 0$
true $\ \mapsto \ x > 0$

pick an arbitrary sequence X that satisfies the specification. That is, X satisfies the following conditions.

$X_0 \geq 0$
$\langle \forall i :: \ X_i \geq 0 \ \Rightarrow \ X_{i+1} \geq 0 \rangle$
$\langle \forall i :: \ (\exists j : \ i \leq j : \ X_j > 0) \rangle$

Now construct a constrained program *FairNatural*$'$ that includes chronicle X and an auxiliary variable j that denotes the *point*. The augmented actions corresponding to α and β are α' and β'.

```
program FairNatural′
  integer x = X_0;
  integer j = 1;
  integer n = 0;

    α′:: n < X_j  →  n := n + 1
  [] β′:: n = X_j  →  x, n := n, 0;  j := j + 1
end {FairNatural′}
```

We claim that in every fair execution of *FairNatural′*, the sequence of values assigned to x is X; i.e., **invariant** $j > 0 \wedge x = X_{j-1}$. We also show that any fair execution of *FairNatural′* corresponds to a fair execution of *FairNatural*. So X is the outcome of some execution of *FairNatural*.

Remarks on the constrained program

The following example shows that a constrained program may not be executable. Consider the specification: output a sequence of integers where each element is one more than the preceding element with, possibly, one exception where the element is one *less* than the preceding element. Given below is a program that is maximal for this specification. The program has an integer variable x and a boolean b. Neither variable is initialized. If b is *true* initially, x only increases, and if b is *false* initially, eventually x decreases once and then increases forever. The output is the sequence of values assigned to x.

```
program choice
  integer x;
  boolean b;

    α:: x := x + 1
  [] β:: ¬b  →  x := x − 1; b := true
end {choice}
```

To prove maximality of this program given a possible output sequence X, we have to construct a constrained program in which the initial value of b depends on X. However, no finite prefix of X can tell us how to initialize b; b has to be set to *true* if and only if X is an increasing sequence. That is, initially,

$$b \;\equiv\; \langle \forall i :\; i \geq 0 :\; X_{i+1} > X_i \rangle$$

Therefore, the constrained program is not executable.

7.3.2 Proving maximality

We describe the proof steps required to establish the maximality of a program for a given specification. The constrained program inherits all safety properties of the original program since assignments to the original variables are not modified. We have to establish the following facts in the constrained program.

1. **Chronicle correspondence:** Show that every fair execution of the constrained program assigns a sequence of values to the original variables which is same as the values in the chronicle.

   - (Safety) Show that the values of the original variables are identical to those of the chronicle at the current *point* (recall that the *point* is given by an auxiliary variable, such as j in *FairNatural*). This proof obligation is stated as an invariant of the constrained program.
   - (Progress) The current value of the *point* is incremented eventually. That is, longer sequences of the chronincle are made to match the values assigned to original variables. (This property often follows from the progress proof for execution correspondence; see below.)

2. **Execution correspondence:** Show that every *fair* execution of the constrained program corresponds to a *fair* execution of the original program such that both executions compute the same values in the original variables.

   - (Safety) The truth of the augmenting guard of each action is preserved by all other actions. That is, the augmenting guard of α' may be falsified by executing α' only.

     This condition is met trivially if all augmenting guards are pairwise disjoint; no guard can then be falsified by the execution of another action because the latter's guard is then *false* and its execution has no effect.
   - (Progress) Show that each augmenting guard is *true* infinitely often.

Example

For *FairNatural*$'$ the proof obligations are as follows.

1. Chronicle correspondence:

   (Safety) **invariant** $j > 0 \ \wedge \ x = X_{j-1}$.

   (Progress) $j = J \mapsto j = J + 1$, for any natural J.

2. Execution correspondence:

   (Safety) The augmenting guard of α', $n < X_j$, is preserved by β', and that of β', $n = X_j$, is preserved by α' . (These follow because the guards are disjoint.)

   (Progress) *true* $\mapsto n < X_j$, *true* $\mapsto n = X_j$.

7.3.3 Justification for the proof rules

The chronicle correspondence rule establishes that the computation of the constrained program P' matches the given chronicle. The safety requirement guarantees the match at the current *point*, and the progress requirement guarantees that successively longer prefixes of the chronicle will be computed.

Given that the execution correspondence conditions hold, we argue that for any fair execution τ of P', $\tau = \tau_0 A_0 \tau_1 \ldots \tau_i A_i \tau_{i+1} \ldots$, there is a fair execution γ of P, $\gamma = \gamma_0 B_0 \gamma_1 \ldots \gamma_i B_i \gamma_{i+1} \ldots$ such that τ reduces to the sequence of states $\gamma_0 \gamma_1 \ldots \gamma_i \gamma_{i+1} \ldots$ over the variables of P.

We modify τ by removing certain actions and states from it as follows. For each action A_i in τ that has an augmenting guard g, if $g(\tau_i)$ does not hold, then ($\tau_i = \tau_{i+1}$ in this case) remove $\tau_i A_i$ from τ. We show that the resulting sequence τ' is an infinite sequence, so it is an execution.

From the progress condition of execution correspondence, the augmenting guard g of an augmented action α' is *true* infinitely often; from the safety condition of execution correspondence, g remains *true* as long as α' is not executed. Each action α' is executed infinitely often in a fair execution of P'. Therefore, α' is infinitely often executed in a state where its augmenting guard g is *true*. Actions whose guards were *false* at the time of their execution have been removed from τ. Therefore, τ' contains every augmented action infinitely often, and the corresponding guard is then *true*. In a state where the augmenting guard g of α' holds, α' has the same effect on the original variables as the action α that it corresponds to. (The superposed actions do not modify the original variables.) Therefore, τ' is an execution of the constrained program, and it corresponds to a fair execution γ of the original program such that the sequence of states for the original variables in τ, τ' and γ are identical.

Not all executions of the constrained program P' have counterparts in P, the original program. In particular, if X is a sequence of zeroes, *FairNatural*$'$ computes X by executing the sequence of actions $(\alpha'\beta')^\omega$; in this execution, α' has no effect and β' computes the next value. However, the corresponding sequence, $(\alpha\beta)^\omega$ in *FairNatural*, does not compute X. The execution correspondence rule ensures that every *fair* execution of P' corresponds to a *fair* execution of P that computes the same sequence of states (in the original variables of P). In *FairNatural*$'$, the guard of α',

$n < X_i$, does not hold infinitely often if X is a sequence of zeroes, so the execution correspondence rule does not apply.

7.3.4 Proof of maximality of program FairNatural

We prove the maximality of *FairNatural* (page 217) using *FairNatural′* (page 221) as the constrained program. We first state certain properties of *FairNatural′* that are required in the maximality proof; these properties follow from the program text. Then we give a complete proof of maximality. In the following, J and K are free variables of type natural.

$$\textbf{invariant } j > 0 \ \wedge \ n \leq X_j \ \wedge \ x = X_{j-1} \tag{P1}$$
$$j = J \ \textbf{co} \ j = J \ \vee \ (j = J + 1 \ \wedge \ n = 0) \tag{P2}$$
$$n \leq X_j \ \wedge \ X_j - n = K \ \textbf{en} \ (n \leq X_j \ \wedge \ X_j - n < K) \ \vee \ n = X_j \tag{P3}$$
$$n = X_j \ \wedge \ j = J \ \textbf{en} \ j = J + 1 \ \wedge \ n = 0 \tag{P4}$$

We also have the following properties of X from the specification of *FairNatural*.

$$\langle \forall i :: \ X_i \geq 0 \ \Rightarrow \ X_{i+1} \geq 0 \rangle$$
$$\langle \forall i :: \ (\exists j : \ i \leq j : \ X_j > 0) \rangle$$

Property (P5), below, follows from the properties of X. Here, $f(i)$ is the next position beyond i where $X_{f(i)}$ is positive. Such a position exists because $\langle \forall i :: \ (\exists j : \ i \leq j : \ X_j > 0) \rangle$.

There is a function f, $f : naturals \ \rightarrow \ naturals$, such that
$f(i) > i$ and $X_{f(i)} > 0$, for all i. (P5)

Proof of chronicle correspondence

- (Safety) **invariant** $j > 0 \ \wedge \ x = X_{j-1}$ follows from (P1).
- (Progress) $j = J \mapsto j = J + 1$, for any natural J: (1)

$$\begin{array}{ll}
true \mapsto n = X_j & \text{, execution corresp.(3), see below} \\
j = J \ \textbf{co} \ j = J \ \vee \ (j = J + 1 \ \wedge \ n = 0) & \\
 & \text{, P2} \\
j = J \ \mapsto \ (n = X_j \ \wedge \ j = J) \ \vee \ (j = J + 1 \ \wedge \ n = 0) & \\
 & \text{, PSP applied to above two} \\
n = X_j \ \wedge \ j = J \ \mapsto j = J + 1 \ \wedge \ n = 0 & \\
 & \text{, from (P4), use basis rule of } \mapsto \\
j = J \ \mapsto \ j = J + 1 \ \wedge \ n = 0 & \text{, cancellation on above two} \quad (2) \\
j = J \ \mapsto \ j = J + 1 & \text{, weaken rhs} \quad \square
\end{array}$$

Proof of execution correspondence

- (Safety): The guards $n < X_j$ and $n = X_j$ are disjoint.

- (Progress) $\mathit{true} \mapsto n = X_j$: (3)

  $n \leq X_j \ \wedge \ X_j - n = K \ \mapsto \ (n \leq X_j \ \wedge \ X_j - n < K) \ \vee \ n = X_j$
  , from P3, use basis rule of $\mapsto$
  $n \leq X_j \mapsto \ n = X_j$, induction
  $\mathit{true} \ \mapsto n = X_j$, substitution axiom:
  invariant $n \leq X_j$ from (P1) □

- (Progress) $\mathit{true} \ \mapsto n < X_j$:

  $j = J \mapsto j = J + 1$, chronicle correspondence (1), above
  $j = J \ \mapsto \ j = f(J) - 1$, induction on above
  $j = f(J) - 1 \ \mapsto \ j = f(J) \wedge n = 0$
  , let J be $f(J) - 1$ in (2) of chronicle correspondence proof
  $j = J \ \mapsto \ j = f(J) \wedge n = 0$, transitivity on the above two
  $j = J \ \mapsto \ n < X_j$, $j = f(J) \ \Rightarrow \ X_j > 0$ from (P5)
  $\mathit{true} \ \mapsto n < X_j$, disjunction over all J □

7.4 Random Assignment

A maximal solution is typically highly nondeterministic. In our previous example, *FairNatural*, we exploited the nondeterminacy of action execution; an arbitrary natural number is computed because n is incremented an indeterminate number of times. In many cases, it is convenient to have nondeterminacy in the code itself. To this end, we introduce a "random assignment" statement that assigns a random value to a variable; see Apt and Olderog [12, section 9.4] for an axiomatic treatment of random assignment. We show the additional proof steps required to prove the constrained program when random assignments are replaced by specific assignments. As an example, we treat a fair unordered channel in which random assignments are essential in constructing the solution.

7.4.1 The form of random assignment

A random assignment statement is of the form

$x :=?$

and execution of this statement assigns a random value of the appropriate type to x. There is no notion of fairness in this assignment; repeated execution of this statement may always assign the same value to x.

Random assignment is convenient for programming maximal solutions. However, it can be simulated using the existing features of our programming model. For instance, in the following program, similar to *FairNatural*, every execution of γ stores a random natural number in x. The program is also maximal: any sequence of natural numbers may be assigned to x.

---

program *RandomNatural*
  integer x;
  integer $n = 0$;

    α:: $n := n + 1$
  ▯ β:: $n > 0 \ \rightarrow \ n := n - 1$
  ▯ γ:: $x := n$
end {*RandomNatural*}

Note on the maximality of program RandomNatural

We omit the proof of maximality of *RandomNatural*; the proof is similar to *FairNatural*'s proof. Note that augmenting α, β and γ by the guards $n < X_j,\ n > X_j,\ n = X_j$, where X is a given sequence of natural numbers as in *FairNatural*, is not sufficient for the proof of maximality. If X is an increasing sequence, for instance, $n > X_j$ never holds, and execution correspondence cannot be proved. Create a constrained program in which the codes of the augmented actions α' and β' are executed at least once following each execution of γ', as follows. Let c be an auxiliary variable, $c \in \{0, 1, 2\}$, where $c = 1$ if the last executed action is γ', and then α' is executed and c is set to 2; if $c = 2$, then β' is executed and c is set to 0; when $c = 0$ any of α', β', γ' may be executed.

---

program *RandomNatural′*
  integer $x = X_0$; **integer** $j = 1$;
  integer $n = 0$;
  enum $(0, 1, 2)$ $c = 0$;

    α':: $(c = 0 \ \wedge \ n < X_j) \ \vee \ c = 1 \ \rightarrow$
          $n := n + 1$; **if** $c = 1$ **then** $c := 2$ **endif**
  ▯ β':: $(c = 0 \ \wedge \ n > X_j) \ \vee \ c = 2 \ \rightarrow$
          $n > 0 \ \rightarrow \ n := n - 1;\ c := 0$
  ▯ γ':: $c = 0 \ \wedge \ n = X_j \ \rightarrow$
          $x := n;\ c := 1;\ j := j + 1$
end {*RandomNatural′*}

Note The augmenting guard of β' implies $n > 0$, since $n > X_j \Rightarrow n > 0$, and it can be shown that $(c = 2 \;\Rightarrow\; n > 0)$ is invariant. Hence, the second guard of β', $n > 0$, may be dropped. □

General form of random assignment

We use a more general form of random assignment

$$x :=? \;\; \textbf{st} \;\; p$$

where variable x is assigned any value such that predicate p holds after the assignment. For instance, for integer x,

$$x :=? \;\; \textbf{st} \;\; \langle \exists i :: \; x = 2 \times i \rangle$$

assigns any even number to x, and

$$y := x;\; x :=? \;\; \textbf{st} \;\; x > y$$

increases the value of x arbitrarily. It is the programmer's responsibility to ensure that such a random assignment is feasible.

Constraining random assignments

In constructing a constrained program, a random assignment is replaced by a specific assignment. If

$x :=? \;\; \textbf{st} \;\; p$ is replaced by
$x := e$

it has to be shown that p holds after the assignment $x := e$.

There is one caveat in constructing these proofs. Earlier, we had said that a constrained program inherits all safety properties of the original program. This is true only if the random assignments have been correctly constrained. Therefore, it cannot be assumed that the constrained program inherits the safety properties until the correctness of these assignments in the constrained program have been shown. In particular, the proof of correctness of these assignments cannot assume any safety properties of the original program; any such assumption has to be proved explicitly in the constrained program.

7.5 Fair Unordered Channel

To illustrate proofs with random assignments, we take the example of a channel interposed between a sender and a receiver. A first-in–first-out (fifo) channel guarantees that the order of delivery of messages is the same as the order in which they were put into the channel. In this section we

consider a fair unordered channel in which (1) the messages are delivered in random order and (2) every message sent is eventually delivered. We propose a maximal solution. A fifo channel implements both requirements, but it is not maximal.

This problem, which is couched as a message transmission problem, has a number of other applications. In particular, the solution can be used to output all natural numbers, where any order of output is a possibility; the sender simply sends the natural numbers in sequence and the channel then delivers them in any possible order. The solution can also be used as a scheduler for programs that have an infinite number of actions, and each action has to be scheduled eventually. This can be implemented by using the sequence created by the channel just described; schedule action i when the channel outputs i.

A maximal program for this problem appears in section 4.1.3. Here, we consider the following simplification of the problem. A program has an infinite input sequence x, and it has to generate a sequence y that is a permutation of x; any permutation is a possible output. We assume further that the items in x are distinct, which can be assured by appending a unique sequence number to each item of x. Then, every item in y corresponds to a unique item in x and vice versa. The specification of the program is as follows: the safety conditions state that every item in y is from x and that the elements in y are unique; the progress condition states that every item of x eventually appears in y.

$$\langle \forall j :: \ (\exists i :: \ x_i = y_j) \rangle$$
$$\langle \forall i, j :: \ y_i = y_j \ \Rightarrow \ i = j \rangle$$
$$\langle \forall i :: \ \mathit{true} \ \mapsto \ (\exists j :: \ x_i = y_j) \rangle$$

7.5.1 Maximal solution for fair unordered channel

The solution shown below consists of two actions, *read* and *write*. In the *read* action, an item is removed from x and stored in set z; in the *write* action an item from z is removed and appended to sequence y. The *write* action permits removal of any item from z, yet it is not sufficient to remove an arbitrary item because the progress property does not hold if some item is left in z forever. Therefore, we associate a *height*, a natural number, with each item that is placed into z, and in the *write* action we remove any item with the smallest height from z. When an item is added to z it is assigned a height that is at least the value of variable t; we describe below how t is computed.

In the following program, the heights of the items are stored in array H; in particular, $H[c]$ and $H[d]$ are the heights of items c and d, respectively. Since the items are all distinct this representation is unambiguous. Variables i and j are the numbers of items read from x and written to y, respectively. Items in a sequence are indexed starting at 0.

---

program *FUnCh*
  integer $i, j, t = 0, 0, 0$;
  item c;
  seq(item) $y = \langle\rangle$;
  set(item) $z = \emptyset$;

    read:: $c := x_i$; $H[c] :=?$ **st** $H[c] \geq t$;
          $z := z \cup \{c\}$; $i := i + 1$

  [] *write*:: $z \neq \emptyset \ \rightarrow$
          $c :=?$ **st** $c \in z \ \wedge \ \langle \forall d : \ d \in z : \ H[c] \leq H[d] \rangle$;
          $t, \ y_j, \ z, \ j := H[c] + 1, \ c, \ z - \{c\}, \ j + 1$
end {*FUnCh*}

The following properties hold in *FUnCh*:

$\langle \forall j :: \ (\exists i :: \ x_i = y_j) \rangle$
$\langle \forall i :: \ true \ \mapsto \ (\exists j :: \ x_i = y_j) \rangle$

The first property says that every item of y is some item in x, and the second property states that every item of x is eventually appended to y. We leave it to the reader to prove these properties. For the progress property, it has to be shown that each item u in z is selected eventually, as c, in *write*. We give an outline of this proof.

Let n be the number of items in z whose height is less than t. For a specific item u in z consider the pair $(H[u] + 1 - t, n)$, where $H[u]$ is the height of u. Both components of the pair are non-negative. This pair is unaffected by the execution of *read*, because *read* does not change $H[u]$, t or n. The pair decreases lexicographically whenever an item is removed from z. Since the pair cannot be decreased indefinitely, eventually u is removed.

7.5.2 The constrained program

Let Y be any permutation of x; i.e.,

$\langle \forall j :: \ (\exists i :: \ x_i = Y_j) \rangle$
$\langle \forall i, j :: \ Y_i = Y_j \ \Rightarrow \ i = j \rangle$
$\langle \forall i :: \ (\exists j :: \ x_i = Y_j) \rangle$

We show that Y is a possible output of the program *FUnCh*. For the proof, we create a constrained program using the transformations described in section 7.3.1. Additionally, we replace the random assignments of *FUnCh* by specific assignments. In particular, the height assigned to an item is its position in Y.

---

program *FUnCh′*
  integer $i, j, t = 0, 0, 0$;
  item c;
  seq(item) $y = \langle\rangle$;
  set(item) $z = \emptyset$;

    read′:: $Y_j \notin z \ \rightarrow$
      $c := x_i$; $H[c] := k$ **st** $c = Y_k$;
      $z := z \cup \{c\}$; $i := i + 1$

  [] *write′*:: $Y_j \in z \ \rightarrow$
      $z \neq \emptyset \ \rightarrow$
        $c := Y_j$;
        $t,\ y_j,\ z,\ j := H[c] + 1,\ c,\ z - \{c\},\ j + 1$
end {*FUnCh′*}

Note The assignment to $H[c]$ in *read′* is not a random assignment; there is a unique value Y_k that matches c, i.e., x_i.

The augmenting guard of *write′*, $Y_j \in z$, implies the original guard, $z \neq \emptyset$. □

7.5.3 Proof of maximality: invariants

We write $x_{0:i}$ for the set $\{x_0, x_1, \ldots, x_{i-1}\}$; thus, $x_{0:0}$ is the empty set. The proofs of the following properties of *FUnCh′* are left to the reader.

invariant $x_{0:i} = z \cup y_{0:j}$ (P1)
invariant $y_{0:j} = Y_{0:j}$ (P2)
invariant $\langle \forall d :\ d \in z :\ d = Y_{H[d]} \ \wedge \ j \leq H[d] \rangle$ (P3)
invariant $t = j$ (P4)

The proofs of (P1, P2) are straightforward; these proofs use the fact that the items in z are distinct. The proof of (P3) needs some explanation. Action *read′* adds c to z where $H[c] = k \wedge c = Y_k$; hence, $c = Y_{H[c]}$. To see that $j \leq H[c]$ in *read′*: it follows from (P1) that $x_i \notin y_{0:j}$; hence, $c = x_i = Y_k$, where $j \leq k$, i.e., $j \leq H[c]$. Action *write′* removes c from z provided that $H[c]$ is the smallest height. From (P3), all heights are distinct because all items in Y are distinct; further, each height is at least j. From the guard, $Y_j \in z$, the height of Y_j is the lowest, and all other items in z have height exceeding j. Therefore, the incrementation of j in *write′* preserves $j \leq H[d]$ for each d in z. The proof of (P4) is similar.

7.5.4 Correctness of implementation of random assignments

We have to show

1. in $read'$: $H[c] := k$ **st** $c = Y_k$ implements $H[c] :=?$ **st** $H[c] \geq t$.
2. in $write'$: $c := Y_j$ implements

$$c :=? \ \textbf{st} \ \ c \in z \ \wedge \ \langle \forall d : \ d \in z : \ H[c] \leq H[d] \rangle$$

• Proof of (1):
In $read'$, prior to the assignment we have, from invariant (P1),

$$\begin{array}{ll} & x_{0:i} = z \cup y_{0:j} \\ \Rightarrow & \{\text{From (P2), } y_{0:j} = Y_{0:j}\text{; } x \text{ is a permutation of } Y\} \\ & x_i \notin Y_{0:j} \wedge \langle \exists k :: \ x_i = Y_k \rangle \\ \Rightarrow & \{\text{Predicate calculus}\} \\ & \langle \exists k : \ k \geq j : \ x_i = Y_k \rangle \\ \Rightarrow & \{k \text{ above is unique since items of } Y \text{ are distinct; } c = x_i\} \\ & H[c] := k \ \textbf{st} \ c = Y_k \text{ implements } H[c] :=? \ \textbf{st} \ H[c] \geq j \\ \Rightarrow & \{\text{From P4, } j = t\} \\ & H[c] := k \ \textbf{st} \ c = Y_k \text{ implements } H[c] :=? \ \textbf{st} \ H[c] \geq t \end{array} \qquad \square$$

• Proof of (2):
We have to show after the assignment $c := Y_j$ that

$$c \in z \ \wedge \ \langle \forall d : \ d \in z : \ H[c] \leq H[d] \rangle$$

holds. Applying the axiom of assignment (see appendix A.4.1), we have to show before the assignment that

$$Y_j \in z \ \wedge \ \langle \forall d : \ d \in z : \ H[Y_j] \leq H[d] \rangle$$

holds. The first conjunct, $Y_j \in z$, follows from the guard of $write'$. The second conjunct is proved next. From (P3),

$$\begin{array}{ll} & \langle \forall d : \ d \in z : \ d = Y_{H[d]} \ \wedge \ j \leq H[d] \rangle \\ \Rightarrow & \{\text{each item of } z \text{ is a unique item of } Y\} \\ & \langle \forall d, k : \ d \in z : \ (d = Y_k \ \equiv \ H[d] = k) \ \wedge \ j \leq H[d] \rangle \\ \Rightarrow & \{Y_j \in z \text{ from the guard of } write'\} \\ & \langle \forall k : \ Y_j = Y_k \ \equiv \ H[Y_j] = k \rangle \ \wedge \ \langle \forall d : \ d \in z : \ j \leq H[d] \rangle \\ \Rightarrow & \{\text{setting } k \text{ to } j\} \\ & H[Y_j] = j \ \wedge \ \langle \forall d : \ d \in z : \ j \leq H[d] \rangle \\ \Rightarrow & \{\text{predicate calculus}\} \\ & \langle \forall d : \ d \in z : \ H[Y_j] \leq H[d] \rangle \end{array}$$

7.5.5 Proof of chronicle and execution correspondence

Proof of chronicle correspondence

- (Safety) We have to show that $y_{0:j} = Y_{0:j}$, which follows from (P2). □
- (Progress) We have to show that $j = J \mapsto j = J + 1$ for any natural J. Each execution of $write'$ increments j. From the progress proof of $write'$ under execution correspondence (see below) the code of $write'$ is executed arbitrarily many times. Therefore, j increases without bound. □

Proof of execution correspondence

- (Safety) The augmenting guards $Y_j \notin z$ and $Y_j \in z$ are disjoint.
- (Progress of $read'$) $true \mapsto Y_j \notin z$:

$$
\begin{array}{lll}
Y_j \in z \ \ \textbf{en} \ \ Y_j \notin z & & \text{, from the text of } FUnCh' \\
Y_j \in z \ \mapsto \ Y_j \notin z & & \text{, basis rule of } \mapsto \\
Y_j \notin z \ \mapsto \ Y_j \notin z & & \text{, implication rule of } \mapsto \\
true \ \mapsto \ Y_j \notin z & & \text{, disjunction of above two}
\end{array}
$$ □

- (Progress of $write'$) $true \mapsto Y_j \in z$:

  There is a unique k such that $Y_j = x_k$. For any n,

$$
\begin{array}{lll}
Y_j \notin z \wedge k - j = n \ \ \textbf{en} \ \ k - j < n & & \text{, from the text of } FUnCh' \\
Y_j \notin z \wedge k - j = n \ \mapsto \ k - j < n & & \text{, basis rule of } \mapsto \\
Y_j \notin z \mapsto \ Y_j \in z & & \text{, induction} \\
Y_j \in z \ \mapsto \ Y_j \in z & & \text{, implication rule of } \mapsto \\
true \mapsto Y_j \in z & & \text{, disjunction of above two}
\end{array}
$$ □

7.6 Faulty Channel

In section 4.1.6, we have considered a faulty channel that may lose messages, duplicate any message an unbounded (though finite) number of times and permute the order of messages. For any point in the computation, it is given that not all messages beyond this point will be lost; otherwise, there can be no guarantee of any message transmission at all. This is similar to the fault model of a channel assumed in the alternating bit protocol [159] (the difference being that in the latter, the channel does not reorder messages). Such a protocol can be proved correct by encoding the communication between the *sender* and the *receiver* using a maximal solution for the faulty channel. As we remarked earlier, it is essential to have a maximal solution

in this case, because a protocol must cope with any possible behavior of the channel. We leave the correctness and maximality proof of the program in section 4.1.6 to the reader. The maximality proof is similar to that for *FUnCh* of section 7.5.1; below, we sketch an outline of the proof.

Consider an infinite sequence S each of whose elements is of the form (put, y) or (get, y). As we did for the unordered channel, we assume that all messages are distinct. A sequence is *valid* if (1) every (get, y) is preceded by exactly one (put, y) and (2) every (put, y) is followed by a finite number —possibly zero— of (get, y)s. Given a valid sequence, we construct an execution as follows. Whenever the next element of S is (put, y), we simulate a call of *put* with argument y. Let d be the number of (get, y) appearing in S. From the definition of a valid sequence, d is well defined. The step in *put* that needs a natural number calls *fnat*, and we require that the value returned by *fnat* be d. Whenever the next element of S is (get, y), we assign y to x; this is possible because $y \in b$ at that point.

7.7 Concluding Remarks

We described the notion of maximality, which rules out implementations with insufficient nondeterminism. A maximal program for a given specification has (up to finite stuttering [114]) all the behaviors admitted by the specification. Although we described techniques for proving maximality only, our proof method may be used to show that a program admits a specific set of executions.

7.8 Bibliographic Notes

Notions similar to maximality have been studied elsewhere in the literature, e.g., *bisimulation* due to Milner [132]. However, unlike bisimulation, which relates two programs (i.e., agents of a process algebra), our notion of maximality relates a program written using guarded commands with a specification written in a UNITY-like temporal logic. The work described in this chapter appears in [101].

8
Program Composition

8.1 Introduction

The goal of a program composition theory is to answer questions of the following form:

- Given a program consisting of one or more components, how do we deduce properties of the program from the specifications of its components?
- Given the specification of a program, how do we partition its design among its components?
- When does a program "inherit" a property of one of its components?

The first question is fundamental for program analysis. The second question is identical to the first except that it involves a design question: *how* to partition a specification. We ignore the heuristic aspects of design in this chapter. Therefore, our answer to the first two questions is to propose inference rules that permit deductions of program properties from those of its components; these rules can also provide guidelines for postulating the specifications of components from a given program specification. The third question is motivated by the possibility of adopting the following "ideal" methodology for program design: partition the specification of the program (to be designed) so that each part in the partition is the specification of a component. If each component meets its specification, and the program

inherits the properties of its components, then the program meets its specification. Each component can then be designed in a similar manner, until the specifications are so trivial that they can be implemented directly. Such a methodology, though attractive, poses a number of formidable technical challenges. We develop a theory in chapter 9 that qualifies the kinds of properties that can be inherited by a program from its components.

We have been intentionally vague about what constitutes a component of a system. Traditionally, a component represents a function or procedure that, upon invocation, returns a result; or it includes the representation of a data object and the methods by which the representation may be manipulated; or it is a process that carries out a portion of the overall computation autonomously. We make no specific assumptions about the nature of a component; therefore, our theory includes each of these interpretations as a special case. Each component is an action system, and we expect the components to operate on global as well as local data. We use "program" to designate a single box as well as unions of several boxes.

A common misconception in program structuring is that a *process* — whose code can be executed on a single processor, or which can be viewed as a unit of computation, as in a transaction processing system, for instance— constitutes a "natural" decomposition of a system; therefore, it is argued that a system should be understood (i.e., specified) process by process. In many cases, it is best to understand a system in a manner orthogonal to the way it is structured as a set of processes. As an example, Fig. 8.1 shows a communication protocol structured as a hierarchy of three components: bit communication, data transfer, and connection management. Each component addresses a specific concern in the given application. There are two processes —a *sender* and a *receiver*— in this system. Each of the components is partitioned between the *sender* and the *receiver*. However, the protocol is best understood component by component, not process by process.

A compositional theory is best applied if the components are "loosely coupled", i.e., their interactions take place through a few shared variables, and there is a strict discipline in the manner in which the shared variables are accessed (read or written). In such cases, concise specifications can be written for the components and their interactions. By contrast, "tight" coupling —as in a mutual exclusion algorithm, for instance— involves interactions that cannot be expressed succinctly, component by component. In a typical mutual exclusion algorithm, for instance, specification of a process is often its implementation. It is not useful to design or verify such systems by considering one component at a time.

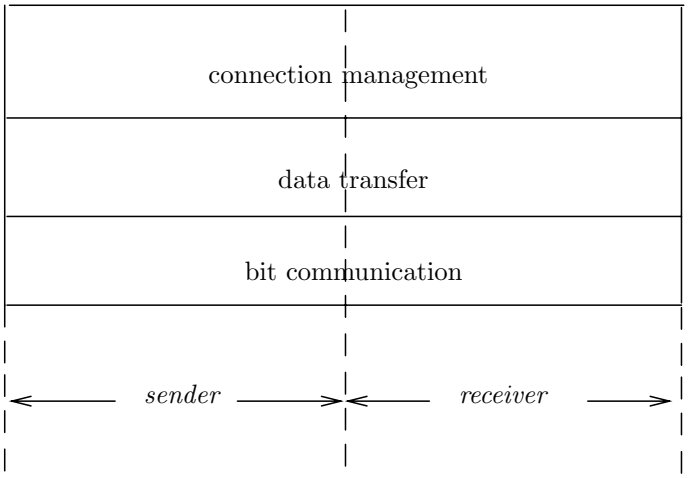


Figure 8.1: Components in a data transfer protocol

8.2 Composition by Union

We begin the study of program composition by considering the *union* operator (written $[\!]$), which is used for compositions of asynchronously operating components. An informal description of the union of programs F and G, denoted by $F \,[\!]\, G$, is as follows: $F \,[\!]\, G$ is a program in which

- The initial conditions of both F and G hold.
- In each step, an action is chosen from F or G for execution, and each action (from both F and G) is chosen eventually.

Union may be applied to any finite set of programs; for technical reasons, we require that this set of programs be nonempty.

Restriction Throughout this chapter, we assume weak fairness in program execution. All boxes are action systems (see chapter 2 for a description of action systems). □

8.2.1 Definition of union

For program composition, it is particularly important to specify which variables named in a program are *local* to that program, i.e., those that can be neither read nor written by any other program; the remaining variables named in that program are *global* (i.e., they may possibly be read or written by other programs). We prefix the declaration of a variable in a program by *local* or *global*.

Note on name clashes Local variables of different programs may have identical names. Similarly, a local variable of one program may have the same name as a global variable declared elsewhere. We assume that the local variables are renamed, if necessary, to avoid such name clashes. □

It is useful to regard $F \parallel G$ as a single action system that is constructed out of F and G as follows. The local (global) variables of $F \parallel G$ are the ones that are declared local (global) in F or in G. Declarations of the same global variable in two different programs should not be conflicting, e.g., if a global variable x is declared in F, it is either undeclared (so not accessed) in G, or similarly declared in G. Also, the initial conditions of F and G should not conflict in order for their union to be defined. The initial condition of $F \parallel G$ is the conjunction of the initial conditions of F and G. (It is useful for theoretical purposes to define $F \parallel G$ even when the initial conditions of F and G conflict; we do not pursue this possibility.) The body of $F \parallel G$ consists of all the actions from F and G.

Example:

We show two boxes —*send* and *receive*— and their union.

---

box *send*
  global boolean
    $b = true$;
  local integer $ns = 0$;

    $b \ \rightarrow \ ns, b := \ ns + 1, false$
end $\{send\}$

box *receive*
  global boolean
    $b = true$;
  local integer $nr = 0$;

    $\neg b \ \rightarrow \ nr, b := \ nr + 1, true$
end $\{receive\}$

Box *send* $\parallel$ *receive* has global variable b and local variables ns and nr. The variables are initialized as in the components. The set of actions of *send* $\parallel$ *receive* is the union of the ones from *send* and *receive*.

---

box *send* $\parallel$ *receive*
  global boolean $b = true$;
  local integer $ns, nr = 0, 0$;

    $b \ \rightarrow \ ns, b := \ ns + 1, false$
  $\parallel$ $\neg b \ \rightarrow \ nr, b := \ nr + 1, true$
end $\{send \parallel receive\}$

Notation So far, we have been writing properties such as p **co** q, **transient** p, p **en** q, $p \ \mapsto \ q$, fixed point predicate (FP), and initial

condition (IC) without explicit mention of the program to which these apply. This is because we always had a single program under consideration. Now we use explicit program names, for instance,

p **co** q in F

to denote that p **co** q holds in program F and

$F.FP$ and $F.IC$

to denote the fixed point predicate and the initial condition, respectively, of program F. □

8.2.2 Hierarchical program structures

From the definition of union, each global variable of F and of G is global in $F \parallel G$. It is often useful to declare that a variable shared between F and G cannot be accessed by any other program, i.e., we would like to hide the variable. For instance, if F and G communicate by messages, the channels between them are local to any system of which both F and G are components. For the example given on page 238, we may wish to specify that variable b cannot be accessed by any program other than *send* and *receive*.

We propose a notation that permits us to hide variables. It also allows hierarchical program construction using union. Consider the following example.

---

program H
  global integer $x = 0$;
  local integer y;

  program F
    global integer x, y;
    local integer z;
    {body of F}
  end $\{F\}$

$\parallel$ **program** G
    global integer x, y;
    local integer u;
    {body of G}
  end $\{G\}$

end $\{H\}$

Here H is $F \parallel G$ with the restriction that the set of local *and* global variables of H is the set of global variables of $F \parallel G$. Thus, variable x can be accessed outside H, whereas y cannot be accessed outside H; only F and G may communicate through y. Additionally, we permit H to initialize its local and global variables as long as it does not conflict with the initial condition of $F \parallel G$; see the initialization of x.

Properties of H can be deduced from the properties of $F \parallel G$; any property of $F \parallel G$ that names only its global variables, i.e., the (local and global) variables of H, is a property of H. Further, we may use the initial condition of H to deduce additional properties, such as invariants, that do not hold in $F \parallel G$.

The declaration mechanism we have proposed hides variables. Hence, it can be used for hierarchical program constructions. For instance, programs F and G may themselves hide certain variables of the components out of which they are built. In this form of hierarchy, only the access rights are constrained. Unlike sequential program hierarchy, flow of control is unaffected; each action of every component program is chosen eventually for execution.

8.2.3 Union theorem

Fundamental to the study of union is the union theorem. It permits us to deduce certain properties of a program from those of its components. We state the theorem for $\langle \parallel i :: F_i \rangle$, where index i assumes values from a nonempty finite set.

Union theorem

1. $\langle \parallel i :: F_i \rangle.IC \equiv \langle \forall i :: F_i.IC \rangle$
   $\langle \parallel i :: F_i \rangle.FP \equiv \langle \forall i :: F_i.FP \rangle$

2. p **co** q in $\langle \parallel i :: F_i \rangle \equiv \langle \forall i :: p$ **co** q in $F_i \rangle$

3. **transient** p in $\langle \parallel i :: F_i \rangle \equiv \langle \exists i ::$ **transient** p in $F_i \rangle$ □

Condition (1) gives a method to compute the initial conditions and fixed point predicates of a system from those of its components. From condition (2), a **co**-property of a system is a **co**-property of each component and vice versa. Condition (3) is the counterpart of condition (2) for transient predicates.

Corollaries

Corollaries (2,3,4), below, show that **stable**, **constant**, and **ensures** properties are inherited by a program from its components. Note that we have

an equivalence (not just an implication) in each case. Corollaries (5,6,7) show that certain properties of a box —**co**, **invariant**, and **en**— are inherited by a program in which the box is a component provided that the predicate in the lhs is stable in the other components of the program.

$$\frac{\begin{array}{l} p \ \ \textbf{co} \ \ q \ \ \text{in } F \\ p' \ \ \textbf{co} \ \ q' \ \ \text{in } G \end{array}}{p \wedge p' \ \ \textbf{co} \ \ q \vee q' \ \ \text{in } F \ [\!] \ G} \tag{1}$$

$$\textbf{stable } p \ \ \text{in } \langle [\!] \ i :: \ F_i \rangle \ \equiv \ \langle \forall \ i :: \ \textbf{stable } p \ \ \text{in } F_i \rangle \tag{2}$$

$$\textbf{constant } e \ \ \text{in } \langle [\!] \ i :: \ F_i \rangle \ \equiv \ \langle \forall \ i :: \ \textbf{constant } e \ \ \text{in } F_i \rangle \tag{3}$$

$$\begin{array}{l} p \ \ \textbf{en} \ \ q \ \ \text{in } \langle [\!] \ i :: \ F_i \rangle \\ \equiv \langle \forall \ i :: \ p \wedge \neg q \ \ \textbf{co} \ \ p \vee q \ \ \text{in } F_i \rangle \ \wedge \ \langle \exists \ i :: \ p \ \ \textbf{en} \ \ q \ \ \text{in } F_i \rangle \end{array} \tag{4}$$

$$\frac{\begin{array}{l} \textbf{stable } p \ \ \text{in } F \\ p \ \ \textbf{co} \ \ q \ \ \text{in } G \end{array}}{p \ \ \textbf{co} \ \ q \ \ \text{in } F \ [\!] \ G} \tag{5}$$

$$\frac{\begin{array}{l} \textbf{stable } p \ \ \text{in } F \\ \textbf{invariant } p \ \ \text{in } G \end{array}}{\textbf{invariant } p \ \ \text{in } F \ [\!] \ G} \tag{6}$$

$$\frac{\begin{array}{l} \textbf{stable } p \ \ \text{in } F \\ p \ \ \textbf{en} \ \ q \ \ \text{in } G \end{array}}{p \ \ \textbf{en} \ \ q \ \ \text{in } F \ [\!] \ G} \tag{7}$$

8.2.4 Proof of the union theorem and its corollaries

Proof of the union theorem, part (1)

The initial condition of $\langle [\!] \ i :: \ F_i \rangle$ is $\langle [\!] \ i :: \ F_i \rangle.IC$, which is the conjunction of the initial conditions of the components, F_is, by definition. Similarly, for the FP,

$$\begin{array}{ll} & \langle [\!] \ i :: \ F_i \rangle.FP \\ \equiv & \quad \{s.FP \text{ is the fixed-point predicate for action } s. \text{ Definition of } FP\} \\ & \langle \forall s : \ s \ \in \ \langle [\!] \ i :: \ F_i \rangle : \ s.FP \rangle \\ \equiv & \quad \{\text{actions of } \langle [\!] \ i :: \ F_i \rangle \text{ is the union of the actions of the } F_i\text{s}\} \\ & \langle \forall i :: \ \langle \forall s : \ s \ \in \ F_i : \ s.FP \rangle \ \rangle \\ \equiv & \quad \{\text{using the definition of } F_i.FP\} \\ & \langle \forall i :: \ F_i.FP \rangle \end{array}$$

Proof of the union theorem, part (2)

$$\begin{array}{ll} & p \ \ \textbf{co} \ \ q \ \ \text{in } \langle [\!] \ i :: \ F_i \rangle \\ \equiv & \quad \{\text{definition of } \textbf{co}\} \end{array}$$

$\langle \forall s :\ s\ \in\ \langle [\!]\ i ::\ F_i \rangle :\ \{p\}\ \ s\ \ \{q\} \rangle$
$\equiv$ {actions of $\langle [\!]\ i ::\ F_i \rangle$ is the union of the actions of the F_is}
$\langle \forall i ::\ \langle \forall s :\ s\ \in\ F_i :\ \{p\}\ \ s\ \ \{q\} \rangle\ \rangle$
$\equiv$ {definition of **co**}
$\langle \forall i ::\ p\ \ \mathbf{co}\ \ q\ \ \text{in}\ F_i \rangle$

Proof of the union theorem, part (3)

transient p in $\langle [\!]\ i ::\ F_i \rangle$
$\equiv$ {definition of transient}
$\langle \exists\ s :\ s\ \in\ \langle [\!]\ i ::\ F_i \rangle :\ \{p\}\ \ s\ \ \{\neg p\} \rangle$
$\equiv$ {actions of $\langle [\!]\ i ::\ F_i \rangle$ is the union of the actions of the F_is}
$\langle \exists\ i ::\ \langle \exists\ s :\ s\ \in\ F_i :\ \{p\}\ \ s\ \ \{\neg p\} \rangle\ \rangle$
$\equiv$ {definition of transient}
$\langle \exists\ i ::$ **transient** p in $F_i \rangle$

Proofs of the corollaries

1. p **co** q in F , premise
$p \wedge p'$ **co** $q \vee q'$ in F , strengthen lhs, weaken rhs
$p \wedge p'$ **co** $q \vee q'$ in G , similarly
$p \wedge p'$ **co** $q \vee q'$ in $F\ [\!]\ G$, union theorem (part 2) □

2. Set q to p in part (2) of the union theorem. □

3. **constant** e in $\langle [\!]\ i ::\ F_i \rangle$
$\equiv$ {definition of constant}
$\langle \forall\ m ::$ **stable** $e = m$ in $\langle [\!]\ i ::\ F_i \rangle\ \rangle$
$\equiv$ {Corollary 2}
$\langle \forall\ m ::\ \langle \forall\ i ::$ **stable** $e = m$ in $F_i \rangle\ \rangle$
$\equiv$ {predicate calculus}
$\langle \forall\ i ::\ \langle \forall\ m ::$ **stable** $e = m$ in $F_i \rangle\ \rangle$
$\equiv$ {definition of constant}
$\langle \forall\ i ::$ **constant** e in $F_i \rangle$ □

4. p **en** q in $\langle [\!]\ i ::\ F_i \rangle$
$\equiv$ {definition of **en**}
$(p \wedge \neg q\ \ \mathbf{co}\ \ p \vee q\ \ \text{in}\ \langle [\!]\ i ::\ F_i \rangle) \wedge$
(**transient** $p \wedge \neg q$ in $\langle [\!]\ i ::\ F_i \rangle$)
$\equiv$ {union theorem, parts 2 and 3}
$\langle \forall\ i ::\ p \wedge \neg q\ \ \mathbf{co}\ \ p \vee q\ \ \text{in}\ F_i \rangle \wedge$
$\langle \exists\ i ::$ **transient** $p \wedge \neg q$ in $F_i \rangle$
$\equiv$ {definition of **en**}
$\langle \forall\ i ::\ p \wedge \neg q\ \ \mathbf{co}\ \ p \vee q\ \ \text{in}\ F_i \rangle\ \wedge\ \langle \exists\ i ::\ p\ \ \mathbf{en}\ \ q\ \ \text{in}\ F_i \rangle$ □

5. p **co** p in F , premise
   p **co** q in G , premise
   p **co** $p \vee q$ in $F \parallel G$, Corollary 1
   p **co** q in $F \parallel G$, $p \Rightarrow q$ from p **co** q in G □

6. **invariant** p in G , premise
   $G.IC \Rightarrow p$, definition of invariant
   $(F \parallel G).IC \Rightarrow G.IC$, part (1) of the union theorem
   $(F \parallel G).IC \Rightarrow p$, from the above two

   Also,
   stable p in G , from the premise, **invariant** p in G
   stable p in F , premise
   stable p in $F \parallel G$, from Corollary 2

   Hence, from $(F \parallel G).IC \Rightarrow p$ and **stable** p in $F \parallel G$,
   invariant p in $F \parallel G$. □

7. p **co** p in F , premise
   $p \wedge \neg q$ **co** $p \vee q$ in G , from p **en** q in G
   $p \wedge \neg q$ **co** $p \vee q$ in $F \parallel G$, Corollary 1 on the above two (*)
   transient $p \wedge \neg q$ in G , from p **en** q in G
   transient $p \wedge \neg q$ in $F \parallel G$, union theorem (part 3)
   p **en** q in $F \parallel G$, above and (*), use definition of **en** □

8.2.5 Locality axiom

A program can change only the variables declared in it; this is the essence of the locality axiom. For a predicate p, let x be the variables named in it that are accessible (i.e., declared as local or global variable) in program F. Execution of an action in F can change p only by changing x; if F does not change x, it does not change p either.

Locality axiom $p \wedge x = m$ **co** $(x = m) \Rightarrow p$ in F. □

If x is empty, $x = m$ should be treated as *true*. If p names no variable declared in F, from the locality axiom, p is stable in F. Applying the same argument, $\neg p$ is also stable in F. Hence, p is constant in F in this case.

As an application of the locality axiom, we prove

Lemma

$$\frac{p \circ q \text{ in } F}{p \wedge x = m \circ q \vee x \neq m \text{ in } F \parallel G}$$

where (1) $\circ$ is any of the operators, **co**, **en**, or $\mapsto$, (2) p and q name only the variables accessible in F, and (3) x denotes the variables shared between F and G.
Proof: We prove the result for each operator separately.

• Assume p **co** q in F:

$p \wedge x = m$ **co** $p \vee x \neq m$ in G, locality axiom
$p \wedge x = m$ **co** $q \vee x \neq m$ in G, weaken rhs: $p \Rightarrow q$ from p **co** q
$p \wedge x = m$ **co** $q \vee x \neq m$ in F
, strengthen lhs, weaken rhs of the premise
$p \wedge x = m$ **co** $q \vee x \neq m$ in $F \parallel G$
, union theorem

• Assume p **en** q in F:

p **en** q in F , given
$p \wedge x = m$ **en** $q \vee x \neq m$ in F , property of **en** (1)
(chapter 6, exercises 4b, 4c)
$(p \wedge \neg q \wedge x = m)$ **co** $(p \wedge x = m) \vee q \vee x \neq m$ in F
, from above (2)
$(p \wedge x = m)$ **co** $(x = m) \Rightarrow p$ in G , locality axiom
$(p \wedge \neg q \wedge x = m)$ **co** $(p \wedge x = m) \vee q \vee x \neq m$ in G
, strengthen lhs, weaken rhs
$p \wedge x = m$ **en** $q \vee x \neq m$ in $F \parallel G$, corollary 4 on (1,2, above)

• Assume $p \mapsto q$ in F:
We apply induction on the structure of the proof of $p \mapsto q$ in F. We have already proved the result for p **en** q in F; the remaining two cases are treated below.

$p \mapsto r$ in F and $r \mapsto q$ in F:
$p \wedge x = m \mapsto r \vee x \neq m$ in $F \parallel G$, induction hypothesis
$p \wedge x = m \mapsto (r \wedge x = m) \vee x \neq m$ in $F \parallel G$
, rewrite rhs
$r \wedge x = m \mapsto q \vee x \neq m$ in $F \parallel G$, induction hypothesis
$p \wedge x = m \mapsto q \vee x \neq m$ in $F \parallel G$, cancellation on above two

$\langle \forall i :: p.i \mapsto q \rangle$ in F where $p \equiv \langle \exists i :: p.i \rangle$:
$p.i \wedge x = m \mapsto q \vee x \neq m$ in $F \parallel G$, induction hypothesis
$\langle \exists i :: p.i \wedge x = m \rangle \mapsto q \vee x \neq m$ in $F \parallel G$
, disjunction
$p \wedge x = m \mapsto q \vee x \neq m$ in $F \parallel G$, rewrite lhs

8.2.6 Union theorem for progress

The union theorem of section 8.2.3 permits us to deduce elementary progress properties in a compositional manner: a predicate is transient in a system if it is transient in some component. Our ultimate goal is to deduce the *leads-to* properties in a similar fashion. Unfortunately, there is no simple compositional rule for *leads-to* similar to that for transient. Plausible inference rules —if $p \mapsto q$ holds in both F and G, then $p \mapsto q$ in $F \parallel G$, or if **stable** p in F and $p \mapsto q$ in G, then $p \mapsto q$ in $F \parallel G$— are all invalid. The following example illustrates the difficulty.

Example Let F and G share a global integer variable x. Program F has a single action that increments x; G has a single action that decrements x. For any k, we can show

$$\begin{array}{l} \mathit{true} \mapsto |x| > k \text{ in } F \\ \mathit{true} \mapsto |x| > k \text{ in } G \end{array}$$

However, $\mathit{true} \mapsto |x| > k$ in $F \parallel G$ does not hold because alternate steps by F and G starting in state $x = 0$ guarantees $|x| \leq 1$ at all times. □

This example lends new meaning to the proverb "Too many cooks spoil the broth": even though either one of F or G could establish a property, their interleaved execution may not; some of the proof steps of $p \mapsto q$ in F may be invalidated by G. For instance, $p \mapsto q$ may have been established in F by

$$p \mapsto r \text{ in } F \quad \text{and} \quad r \mapsto q \text{ in } F$$

Suppose that program G always falsifies r eventually. Then the following segment of an execution of $F \parallel G$ starting in a state that satisfies p,

$$\{p\} \text{ execution of } F \quad \{r\} \text{ execution of } G \quad \{\neg r\}$$

establishes $\neg r$, and there is no guarantee that q will ever be established later during this execution. Program G is said to have *interfered* with F's execution.

Avoiding interference

We may so restrict the behavior of program G that it never *interferes* with F's execution; in its extreme form, we may prohibit G from writing into a variable that F reads. Then every progress proof in F is unaffected by the steps of G. But this observation is not too useful because the result applies only to programs with severely restricted forms of variable sharing. We propose a less severe restriction on G that preserves the progress properties of F.

Let x denote all the variables shared between F and G and $<$ be a well-founded ordering relation among the possible values of x. Let predicates p and q name only the variables accessible from F (including variables in x).

Union theorem for progress

$$\frac{\begin{array}{c} p \;\mapsto\; q \;\text{ in } F\,, \\ p \;\wedge\; \neg q \;\wedge\; x = m \;\;\textbf{co}\;\; (p \;\wedge\; x \leq m) \;\vee\; q \;\text{ in } F \,[\!]\, G \end{array}}{p \;\mapsto\; q \;\text{ in } F \,[\!]\, G}$$

The progress condition in the hypothesis requires $p \;\mapsto\; q$ to be established in one component (in this case, F). The safety condition requires that both F and G may only "decrease" x along the well-founded order (as long as $p \wedge \neg q$ holds). A consequence of the safety condition is that F and G preserve p as long as q is not established; that is, $p \wedge \neg\, q \;\;\textbf{co}\;\; p \vee q$ holds.

To see the validity of this theorem in operational terms, consider any execution of $F \,[\!]\, G$ starting in a state where p holds. If G changes the value of x an infinite number of times in this execution, from

$$p \;\wedge\; \neg\, q \;\wedge\; (x = m) \;\;\textbf{co}\;\; (p \;\wedge\; x \leq m) \;\vee\; q$$

eventually q will hold (because as long as $p \wedge \neg\, q$ holds, the value of x decreases each time it is changed, from the safety hypothesis; from well-foundedness, x cannot decrease forever, so q will be established). If G changes x only a finite number of times and q has not been established by the time G last changes x, then p holds at that point (again from the given safety property); G no longer interferes by changing x; so, from $p \mapsto q$ in F, eventually q is established.

Proof of the Union Theorem for Progress

$p \;\mapsto\; q$ in F	, premise of the theorem
$p \;\wedge\; x = m \;\mapsto\; q \;\vee\; x \neq m$ in $F \,[\!]\, G$	, above and lemma (page 243)
$p \;\wedge\; \neg\, q \;\wedge\; x = m \;\;\textbf{co}\;\; (p \;\wedge\; x \leq m) \;\vee\; q$ in $F \,[\!]\, G$	, premise of the theorem
$p \;\wedge\; \neg\, q \;\wedge\; x = m \;\mapsto\; (p \;\wedge\; x < m) \;\vee\; q$ in $F \,[\!]\, G$	, PSP and simplify
$p \;\wedge\; q \;\wedge\; x = m \;\mapsto\; q$	, implication rule for $\mapsto$
$p \;\wedge\; x = m \;\mapsto\; (p \;\wedge\; x < m) \;\vee\; q$ in $F \,[\!]\, G$	, disjunction of above two
$p \;\mapsto\; q$ in $F \,[\!]\, G$	, induction on the above

8.3 Examples of Program Union

Program union is a central concept of our theory. So we consider a few examples involving union in some depth. We show the component boxes and derive some of their properties employing the union theorem and its corollaries.

8.3.1 Parallel search

The following program contains the essence of parallel state-space search. In a parallel search, a given state space is partitioned and a process assigned to search each part independently. The processes accumulate their results in certain global variables.

We consider a very simple case: count the number of zeroes in an array, $A[0..N]$, $N \geq 0$. We partition the elements of A into those with even and those with odd indices. Box *Even* counts the number of zeroes in the part with even indices; box *Odd* works analogously. The total count is stored in a global variable z.

---

box *Even*
  global integer $z = 0$;
  global array$[0..N]$**(integer)** A;
  local integer $i = 0$;

    $A[i] = 0 \ \wedge \ i \leq N \ \rightarrow \ i, \ z := i + 2, \ z + 1$
  $[\!]$ $A[i] \neq 0 \ \wedge \ i \leq N \ \rightarrow \ i, \ z := i + 2, \ z$
end $\{Even\}$

box *Odd*
  global integer $z = 0$;
  global array$[0..N]$**(integer)** A;
  local integer $j = 1$;

    $A[j] = 0 \ \wedge \ j \leq N \ \rightarrow \ j, \ z := j + 2, \ z + 1$
  $[\!]$ $A[j] \neq 0 \ \wedge \ j \leq N \ \rightarrow \ j, \ z := j + 2, \ z$
end $\{Odd\}$

It is straightforward that z is the number of zeroes in A when the state satisfies the fixed point predicates of both boxes (further, both boxes will eventually satisfy their *FP*s). Observe that each action in F or G is an "atomic" action in $F \,[\!]\, G$. Thus, assignments to i and z in *Even* are carried out together, without preemption by any action of *Odd*.

Note A better methodology for parallel search is for each box to store the result of its search in a local variable. On completion of its search it updates the global variable in which the result of the entire search is stored, using the value of its local variable. Thus, *Even* and *Odd* may store their zero-counts in local variables ez and oz, respectively, and on completion of their searches, they may increase z by ez and oz. □

Derivations of program properties

We prove the following properties of $Even \parallel Odd$. In the following, *even.i* holds iff i is *even*; similarly, *odd.j*.

1. **stable** $z \geq m$ for any integer m
2. **invariant** $even.i \ \wedge \ i \leq N+2$
   invariant $odd.j \ \wedge \ j \leq N+2$
3. **stable** $i > N$, **stable** $j > N$
4. **invariant** $z =$
   $\langle + \ k : \ 0 \leq k < i \ \wedge \ even.k \ \wedge \ A[k] = 0 : \ 1 \rangle +$
   $\langle + \ k : \ 0 \leq k < j \ \wedge \ odd.k \ \wedge \ A[k] = 0 : \ 1 \rangle$
5. $true \ \mapsto \ i > N$, $true \ \mapsto \ j > N$
6. $true \ \mapsto \ z = \langle + \ k : \ 0 \leq k \leq N \ \wedge \ A[k] = 0 : \ 1 \rangle$

In the following proofs, we do not explicitly show the proofs of **co**- or **en**-properties in the boxes *Even* and *Odd*. Such properties are deduced from the program texts using the techniques of chapters 5 and 6.

- Proof of (1), **stable** $z \geq m$ in $Even \parallel Odd$:

  stable $z \geq m$ in $Even$, from the text of $Even$
  stable $z \geq m$ in Odd , from the text of Odd
  stable $z \geq m$ in $Even \parallel Odd$, union theorem corollary □

- Proof of (2), **invariant** $even.i \ \wedge \ i \leq N+2$ in $Even \parallel Odd$:

  constant i in Odd , locality axiom (section 8.2.5)
  stable $even.i \ \wedge \ i \leq N+2$ in Odd , constant predicate is stable
  invariant $even.i \ \wedge \ i \leq N+2$ in $Even$
  , from the text of $Even$
  invariant $even.i \ \wedge \ i \leq N+2$ in $Even \parallel Odd$
  , union theorem corollary

  Proof of **invariant** $odd.j \ \wedge \ j \leq N+2$ is similar. □

- Proof of (3): similar to proof of (2). □

- Proof of (4):
  Let q be the predicate

  $z = \langle + \ k : \ 0 \leq k < i \ \wedge \ even.k \ \wedge \ A[k] = 0 : \ 1 \rangle$
  $+ \langle + \ k : \ 0 \leq k < j \ \wedge \ odd.k \ \wedge \ A[k] = 0 : \ 1 \rangle$

We show **invariant** q in $Even \parallel Odd$. From the union theorem, initially,

$$
\begin{array}{ll}
 & i, j, z = 0, 1, 0 \ \text{ in } Even \parallel Odd \\
\Rightarrow & \{\text{predicate calculus}\} \\
 & z = \langle + \ k : \ 0 \leq k < i \ \wedge \ even.k \ \wedge \ A[k] = 0 : \ 1 \rangle \\
 & \quad + \langle + \ k : \ 0 \leq k < j \ \wedge \ odd.k \ \wedge \ A[k] = 0 : \ 1 \rangle
\end{array}
$$

It remains to show that **stable** q in $Even \parallel Odd$. From the union theorem corollary, we have to show that this expression is stable in both $Even$ and Odd, which follow from the texts of $Even$ and Odd. □

- Proof of (5), $true \ \mapsto \ i > N$ in $Even \parallel Odd$:

$i = k$ **en** $i > k$ in $Even$	, from the text of $Even$
stable $i = k$ in Odd	, locality axiom
$i = k$ **en** $i > k$ in $Even \parallel Odd$	, union theorem corollary
$i = k \ \mapsto \ i > k$ in $Even \parallel Odd$	, definition of $\mapsto$
$true \ \mapsto \ i > N$ in $Even \parallel Odd$	, induction on integers

Proof of $true \ \mapsto \ j > N$ is similar. □

Note the structure of progress proof (5): an **en** property is first established for the system using the union theorem; then the *leads-to* property is established and induction applied. Conversely, if we had first established

$$true \ \mapsto \ i > N \ \text{ in } Even$$

then union theorem could not have been used directly in proving the desired result.

- Proof of (6),
  $true \ \mapsto z = \langle + \ k : \ 0 \leq k \leq N \ \wedge \ A[k] = 0 : \ 1 \rangle$ in $Even \parallel Odd$:

The following proof is over $Even \parallel Odd$.

$$
\begin{array}{ll}
 & true \\
\mapsto & \{\text{completion rule using (3) and (5)}\} \\
 & i > N \ \wedge \ j > N \\
\Rightarrow & \{\text{use (2) and the substitution axiom}\} \\
 & i > N \ \wedge \ j > N \\
 & \wedge \ even.i \ \wedge \ i \leq N + 2 \ \wedge \ odd.j \ \wedge \ j \leq N + 2 \\
\Rightarrow & \{\text{use (4)}\} \\
 & z = \langle + k : \ 0 \leq k \leq N \ \wedge \ A[k] = 0 : \ 1 \rangle
\end{array}
$$
□

The proof of this program is given in excruciating detail to highlight the application of the union theorem. In each case, the proof is carried out at two levels: (1) deducing properties directly from the text of a program component, and (2) deducing properties by applying the union theorem and other inference rules. We have shown the second kind of deduction in

detail. This proof structure permits us to carry out deductions where the components are specified by their properties and not by their codes. We study another such proof in section 8.3.3.

8.3.2 Handshake protocol

Consider the program given as an example in section 8.2.1. Box *send* sends a message and waits for acknowledgment before sending another message. Box *receive* sends an acknowledgment after receiving a message. Therefore, the two actions of box *send* $[\!]$ *receive* are, in effect, executed alternately. The shared boolean variable, b, determines which action is effectively executed next: if b holds, then *send* can send a message, and if $\neg b$ holds, *receive* can receive a message (and transmit an acknowledgment). Each action inverts b to allow the other action to be executed. We do not explicitly show message send and receive; variables ns and nr are the number of messages sent and received, by *send* and *receive*, respectively.

Here, we consider a variation of the protocol: boolean b is eliminated, replaced by $nr \geq ns$. We focus attention on one of the components, *send*, given below.

---

box *send*
  global integer $nr, ns = 0, 0$;

  $nr \geq ns \;\rightarrow\; ns := ns + 1$
end $\{send\}$

Program F with which *send* is to be composed should have the following properties: (1) it should not modify ns (see F1 below), (2) it may only increase nr and only up to ns (see F1 below), and (3) if $nr < ns$, it should increase nr to establish $nr = ns$ (see F2 below). We can directly establish these properties for program *receive* of section 8.2.1 from its code (substituting $\neg b$ by $nr < ns$). However, our interest is in establishing properties of F $[\!]$ *send* for any F that satisfies the specification, among them program *receive*.

We postulate the following safety and progress properties of F. Henceforth, m and n are arbitrary natural numbers.

- (F1) $nr, ns = m, n$ **co** $ns = n \;\wedge\; (nr = m \;\vee\; m \leq nr \leq n)$ in F.
- (F2) $nr < ns \;\mapsto\; nr = ns$ in F.

From the properties (F1) and (F2) of F and the text of *send* we will establish that (1) nr and ns differ by at most 1 (see FG1), so *send* and F, in effect, are executed alternately, and (2) ns increases without bound (see FG2); therefore, so does nr.

- (FG1) **invariant** $nr \leq ns \leq nr + 1$ in $F \parallel send$.
- (FG2) $true \mapsto ns > n$ in $F \parallel send$.

Derivations of Program Properties

First, we prove a number of safety properties of F, from its specification (F1), and *send*, from its code.

- (F3) **constant** ns in F: In this proof, all properties are in F.

$nr, ns = m, n$ **co** $ns = n$	, weaken rhs of F1
$ns = n$ **co** $ns = n$	, disjunction over all m
constant ns	, definition of constant

- (F4) **stable** $nr \geq m$ in F: use elimination theorem on F1.
- (F5) **stable** $nr \leq ns$ in F: use elimination theorem on F1.

From its text, we can establish the following properties in *send*:

- (F6) **stable** $ns \geq n$ in *send* .
- (F7) **stable** $nr \geq m$ in *send* .
- (F8) **stable** $nr \leq ns$ in *send* .

We deduce

- (FG3) **stable** $ns \geq n$ in $F \parallel send$: union theorem on (F3, F6).
- (FG4) **stable** $nr \geq m$ in $F \parallel send$: union theorem on (F4, F7).
- (FG5) **stable** $nr \leq ns$ in $F \parallel send$: union theorem on (F5, F8).

We now establish the safety property (FG1) of the system.

- Proof of (FG1), **invariant** $nr \leq ns \leq nr + 1$ in $F \parallel send$:

Proof: All the properties below are in $F \parallel send$.

initially $nr \leq ns$	, initially $nr, ns = 0, 0$; from *send*'s text
stable $nr \leq ns$	, from FG5
invariant $nr \leq ns$	, from above two

The proof of **invariant** $ns \leq nr + 1$ in $F \parallel send$ is similar. □

Next, we prove two progress properties in preparation for the proof of (FG2).

• Proof of (FG6), $nr < ns \;\mapsto\; nr = ns$ in $F \,[\!]\, send$:
Proof: Given F2, this result follows from the union theorem for progress (section 8.2.6), provided that the following is a property in $F \,[\!]\, send$:

$$nr < ns \;\wedge\; (nr, ns) = (m, n) \;\; \textbf{co}$$
$$\langle nr < ns \;\wedge\; (nr, ns) \preceq (m, n)\rangle \;\vee\; \langle nr = ns\rangle$$

From the union theorem, this property (with a suitable partial order $\prec$) has to be proved in both F and *send*. We use the following partial order $\prec$ over pairs of naturals:

$$(p, q) \prec (r, s) \;\equiv\; |p - q| < |r - s|$$

In *send* the property can be shown from its text. The proof in F follows by applying the elimination theorem on F1. □

• Proof of (FG7), $nr = ns \;\mapsto\; nr < ns$ in $F \,[\!]\, send$:
Proof: similar to that of FG6. □

Now we establish the main progress property of the system, FG2.

• Proof of (FG2), $true \;\mapsto\; ns > n$ in $F \,[\!]\, send$, for any natural n:
In the following proof all properties are in $F \,[\!]\, send$.

stable $ns \geq n \;\wedge\; nr \geq n$
, conjuction of (FG3, FG4), using n for m
$nr = ns \;\wedge\; ns \geq n \;\wedge\; nr \geq n \;\mapsto\; nr < ns \;\wedge\; ns \geq n \;\wedge\; nr \geq n$
, stable conjuction of FG7 and above
$nr = ns \;\wedge\; ns \geq n \;\mapsto\; ns > n$
, rewrite lhs and weaken rhs of above
$nr < ns \;\wedge\; ns \geq n \;\mapsto\; nr = ns \;\wedge\; ns \geq n$
, stable conjuction of FG6 and FG3
$nr \leq ns \;\wedge\; ns \geq n \;\mapsto\; ns > n$
, transitivity, disjunction on above two
$ns \geq n \;\mapsto\; ns > n$, substitution axiom: $nr \leq ns$ is *true*
$true \;\mapsto\; ns > n$, induction □

8.3.3 Semaphore

The purpose of the next example is to illustrate how specifications of some components can be combined with the codes of other components to derive properties of a system. We choose an example, *semaphore*, that is treated in great detail in section 4.9. Our goal here is not to gain additional insight into the semaphore problem but to illustrate the application of the union theorem on a well-understood problem so that the specifics of the problem do not obscure the treatment of program composition.

We show a program *sem* that implements a binary semaphore. Programs F and G interact with *sem* by requesting a P- or a V-operation. Program

F (or G) requests a P-operation by setting a global boolean variable pf (or pg) to *true*. Program *sem* responds to a request for P-operation by granting the semaphore to F or G (if the semaphore is available); the response is indicated by changing the corresponding variable —pf or pg— from *true* to *false*. (Thus, as long as pf is *true* F is "waiting" for the semaphore.) Similarly, F and G request V-operations by setting global variables vf and vg, respectively, to *true* and the *sem* program sets them to *false* after completion of the respective V-operations.

Remark: The protocol for requesting an operation, P or V, and receiving a response is cumbersome in the model of action systems. However, this is the best we can do if components communicate through shared variables. The model of Seuss, in which the procedure call is a primitive, was motivated by the need to simplify interactions among the components. □

It is readily seen that program *sem* can be regarded as the union of two programs *semf* and *semg*, which respond to requests from F and G, respectively. The value of the semaphore has to be shared between *semf* and *semg*; boolean variable s stores this value. Fig. 8.2 shows the decomposition of *sem*. Codes for *semf* and *semg* are as follows. Program *sem* is *semf* $[\!]$ *semg*.

---

box *semf*
  global boolean $pf, vf = false, false$;
  global boolean $s = true$;

    $pf \ \wedge \ s \ \rightarrow \ pf, s := \ false, false$
  $[\!]$ $vf \ \rightarrow \ vf, s := \ false, true$
end $\{semf\}$

box *semg*
  global boolean $pg, vg = false, false$;
  global boolean $s = true$;

    $pg \ \wedge \ s \ \rightarrow \ pg, s := \ false, false$
  $[\!]$ $vg \ \rightarrow \ vg, s := \ false, true$
end $\{semg\}$

Another version of semaphore

The protocol for changing pf, vf, pg, and vg is too liberal; it permits program F to set pf and vf to *true* simultaneously, and this is illegal according to our interpretation. We can restrict the behavior of F by imposing ex-

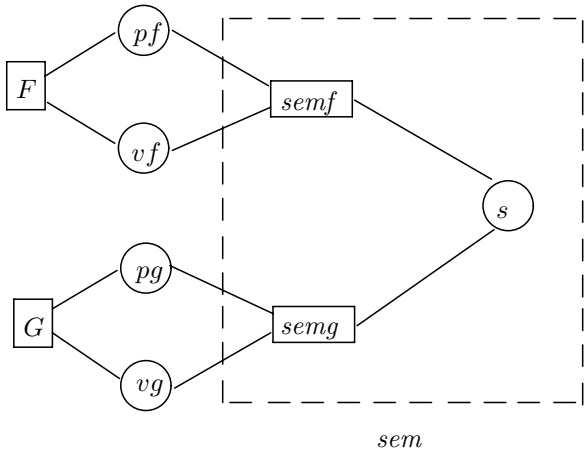


Figure 8.2: Decomposition of program *sem* into *semf* and *semg*. The semaphore value s is shared between *semf* and *semg*.

plicit constraints in its specification. A better strategy is to restrict the interface syntactically in such a way that few illegal actions are permitted.

Introduce a global boolean variable *rf* that F sets to *true* to request a P-operation or a V-operation. The ambiguity —whether the requested operation is a P- or a V— can be resolved by the semaphore program because it can retain information regarding which program holds the semaphore; if program F holds the semaphore, *rf* being *true* signals a V-operation, and if F does not hold the semaphore, *rf* being *true* signals a request for a P-operation. The semaphore program sets *rf* to *false* to indicate completion of the requested operation. Similarly, program G communicates its intention using a global boolean variable *rg*.

This proposal simplifies the interface considerably. The semaphore program now has to retain information regarding who holds the semaphore. Let *sh* be a variable that assumes the following values:

F: Program F holds the semaphore.

G: Program G holds the semaphore.

N: Neither program holds the semaphore.

Now we no longer need variable s (which stores the semaphore value) because

$$s \equiv (sh = \mathrm{N}).$$

Introduction of *sh* prevents both programs from holding the semaphore simultaneously; thus, the proof obligation for mutual exclusion has been discharged by syntactic means.

As before, we implement the semaphore program as the union of two programs, *semf1* and *semg1* in this case.

---

box *semf1*
  global boolean rf;
  global enum (F,G,N) $sh = \mathrm{N}$;

  P:: $rf \ \wedge \ sh = \mathrm{N} \ \rightarrow \ rf, sh := \mathit{false}, \mathrm{F}$
$[\!]$ V:: $rf \ \wedge \ sh = \mathrm{F} \ \rightarrow \ rf, sh := \mathit{false}, \mathrm{N}$
end $\{semf1\}$

box *semg1*
  global boolean rg;
  global enum (F,G,N) $sh = \mathrm{N}$;

  P:: $rg \ \wedge \ sh = \mathrm{N} \ \rightarrow \ rg, sh := \mathit{false}, \mathrm{G}$
$[\!]$ V:: $rg \ \wedge \ sh = \mathrm{G} \ \rightarrow \ rg, sh := \mathit{false}, \mathrm{N}$
end $\{semg1\}$

Program *semf1* $[\!]$ *semg*1 implements a "weak" semaphore; F may request the semaphore but it may never hold it because G is being granted the semaphore arbitrarily many times (and G releases the semaphore arbitrarily many times). This program has the following progress property:

$$rf \vee rg \ \mapsto \ sh \neq \mathrm{N} \ \text{ in } F \ [\!] \ G \ [\!] \ semf1 \ [\!] \ semg1$$

A program for strong semaphore

A "strong" semaphore is one in which every request for a P-operation is eventually honored, provided that every program holding the semaphore eventually releases it. That is, under the assumption of eventual release of the semaphore,

$$rf \ \mapsto \ sh = \mathrm{F} \ \text{ in } F \ [\!] \ G \ [\!] \ semf2 \ [\!] \ semg2$$
$$rg \ \mapsto \ sh = \mathrm{G} \ \text{ in } F \ [\!] \ G \ [\!] \ semf2 \ [\!] \ semg2$$

To implement a strong semaphore, we introduce a boolean variable pr that indicates which process has the priority in acquiring the semaphore: variable pr is *true* iff F is to be given priority whenever there is contention for the semaphore. The initial value of pr is immaterial. Whenever F is granted the semaphore pr is set to *false* —thus giving priority to G in future contentions— and similarly, pr is set to *true* whenever G is granted the semaphore.

As before, we introduce two component programs, *semf2* and *semg2*. Variables rf, rg, and pr are shared between these two programs.

---

box *semf2*
  global boolean rf, rg, pr;
  global enum (F,G,N) $sh = \mathrm{N}$;

  P:: $rf \ \wedge \ sh = \mathrm{N} \ \wedge \ (pr \vee \neg rg) \ \rightarrow \ rf, sh, pr := \mathit{false}, \mathrm{F}, \mathit{false}$
$[\!]$ V:: $rf \ \wedge \ sh = \mathrm{F} \ \rightarrow \ rf, sh := \mathit{false}, \mathrm{N}$
end $\{semf2\}$

box *semg2*
  global boolean rf, rg, pr;
  global enum (F,G,N) $sh = \mathrm{N}$;

  P:: $rg \ \wedge \ sh = \mathrm{N} \ \wedge \ (\neg pr \vee \neg rf) \ \rightarrow \ rg, sh, pr := \mathit{false}, \mathrm{G}, \mathit{true}$
$[\!]$ V:: $rg \ \wedge \ sh = \mathrm{G} \ \rightarrow \ rg, sh := \mathit{false}, \mathrm{N}$
end $\{semg2\}$

We show below that

$rf \ \mapsto \ sh = \mathrm{F}$ in $F \,[\!]\, G \,[\!]\, semf2 \,[\!]\, semg2$
$rg \ \mapsto \ sh = \mathrm{G}$ in $F \,[\!]\, G \,[\!]\, semf2 \,[\!]\, semg2$

Derivations of program properties

We have four boxes under consideration: F, G, *semf2*, and *semg2*. This is an example where codes of some of the components —F and G— are unavailable. We prove properties of the program using the specifications of F and G.

The composite program, $F \,[\!]\, G \,[\!]\, semf2 \,[\!]\, semg2$, is designated by *sys*. A safety property of *sys* is that at most one program —F or G— holds the semaphore at any time; this is vacuously established because sh identifies the holder of the semaphore, if any. A progress property of *sys* is that any program, F or G, which requests the semaphore is eventually granted the semaphore, provided that every semaphore holder requests a V-operation eventually. Specifically, our goal is to establish

$rf \ \mapsto \ sh = \mathrm{F}$	in *sys* and analogously,	(PrF)
$rg \ \mapsto \ sh = \mathrm{G}$	in *sys* assuming	(PrG)
$sh = \mathrm{F} \ \mapsto \ rf$	in *sys* and	(s0)
$sh = \mathrm{G} \ \mapsto \ rg$	in *sys*	(s1)

Assume that programs F and G obey the following protocol in modifying the global variables:

stable rf in F	(s2)
stable rg in G	(s3)

Also, the only global variable which is accessible in F is rf; similarly, G can access only rg. Therefore, from the locality axiom,

$$\textbf{constant } rg, pr, sh \text{ in } F \tag{s4}$$
$$\textbf{constant } rf, pr, sh \text{ in } G \tag{s5}$$

We can establish the following safety properties of sys:

$$\textbf{invariant } sh = \mathrm{F} \Rightarrow \neg pr \tag{s6}$$
$$\textbf{invariant } sh = \mathrm{G} \Rightarrow pr \tag{s7}$$
$$rf \ \wedge \ sh \neq \mathrm{F} \quad \textbf{co} \quad rf \ \vee \ sh = \mathrm{F} \tag{s8}$$
$$\neg rg \ \wedge \ sh = \mathrm{G} \quad \textbf{co} \quad sh = \mathrm{G} \tag{s9}$$

Properties (s6–s9) are proved by applying the union theorem. We prove (s9) for illustration.

- Proof of (s9), $\neg rg \ \wedge \ sh = \mathrm{G} \quad \textbf{co} \quad sh = \mathrm{G}$ in sys:

$$\begin{array}{lll}
\textbf{stable } sh = \mathrm{G} \text{ in } G & \text{, from s5} & \\
\neg rg \ \wedge \ sh = \mathrm{G} \quad \textbf{co} \quad sh = \mathrm{G} \text{ in } G & \text{, strengthen above lhs} & (1) \\
\textbf{stable } \neg rg \ \wedge \ sh = \mathrm{G} \text{ in } F & \text{, from s4} & (2) \\
\textbf{stable } \neg rg \ \wedge \ sh = \mathrm{G} \text{ in } semf2 & \text{, text of } semf2 & (3) \\
\textbf{stable } \neg rg \ \wedge \ sh = \mathrm{G} \text{ in } semg2 & \text{, text of } semg2 & (4) \\
\neg rg \ \wedge \ sh = \mathrm{G} \quad \textbf{co} \quad sh = \mathrm{G} \text{ in } sys & & \\
\qquad \text{, union theorem corollary on (1–4)} & & \square
\end{array}$$

The following progress properties capture the essence of the system operation: (s10) says that a request for a V-operation is eventually honored, and then the program other than the requester has priority in acquiring the semaphore; (s11) says that if the program that requests the semaphore has priority and the semaphore is available, then the program is eventually granted the semaphore; (s12) says that a request for a P-operation when the semaphore is available is granted unless the other program requests the semaphore.

In sys:

$$rg \ \wedge \ sh = \mathrm{G} \ \mapsto \ pr \ \wedge \ sh = \mathrm{N} \tag{s10}$$
$$rf \ \wedge \ pr \ \wedge \ sh = \mathrm{N} \ \mapsto \ sh = \mathrm{F} \tag{s11}$$
$$rf \ \wedge \ sh = \mathrm{N} \ \mapsto \ (sh = \mathrm{F}) \vee (rf \ \wedge \ rg \ \wedge \ sh = \mathrm{N}) \tag{s12}$$

Note Each of the properties (s6–s12) has a dual that is obtained by replacing one predicate in the following pairs by the other: (rf, rg), $(pr, \neg pr)$, $(sh = \mathrm{F}, sh = \mathrm{G})$, $(sh \neq \mathrm{F}, sh \neq \mathrm{G})$, and $(sh = \mathrm{N}, sh = \mathrm{N})$. For example, the dual of (s11) is $rg \ \wedge \ \neg pr \ \wedge \ sh = \mathrm{N} \ \mapsto \ sh = \mathrm{G}$. $\square$

The proofs of the progress properties (s10–s12) are similar. We establish these properties as **ensures** properties in sys by showing that an appropriate predicate is transient in $semg2$ for (s10), in $semf2$ for (s11), and

in *semf2* for (s12). Next we prove corresponding safety properties for the other components; they are derived from (s2–s5) and the texts of *semf2* and *semg2*. A typical proof is given below.

- Proof of (s12),
  $rf \ \wedge \ sh = \mathrm{N} \ \mapsto \ (sh = \mathrm{F}) \ \vee \ (rf \ \wedge \ rg \ \wedge \ sh = \mathrm{N})$ in *sys*:

  stable $rf \ \wedge \ sh = \mathrm{N}$ in G , from s5
  $rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N}$ **co** $(sh = \mathrm{F}) \ \vee \ (rf \ \wedge \ sh = \mathrm{N})$ in G (1)
  , strengthen lhs, weaken rhs
  stable $rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N}$ in F (2)
  , from (s2, s4)
  stable $rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N}$ in *semg*2 (3)
  , text of *semg*2
  $rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N}$ **co** $(sh = \mathrm{F}) \ \vee \ (rf \ \wedge \ sh = \mathrm{N})$ in *semf2* (4)
  , text of *semf2*
  $rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N}$ **co** $(sh = \mathrm{F}) \ \vee \ (rf \ \wedge \ sh = \mathrm{N})$ in *sys* (5)
  , union theorem on (1–4)
  transient $rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N}$ in *semf2* (6)
  , text of *semf2*
  $rf \ \wedge \ sh = \mathrm{N}$ **en** $(sh = \mathrm{F}) \ \vee \ (rf \ \wedge \ rg \ \wedge \ sh = \mathrm{N})$ in *sys*
  , union theorem on (5–6)
  $rf \ \wedge \ sh = \mathrm{N} \ \mapsto \ (sh = \mathrm{F}) \ \vee \ (rf \ \wedge \ rg \ \wedge \ sh = \mathrm{N})$ in *sys*
  , from above □

Properties (s2–s12) have the information we need to establish the desired progress properties, (PrF) and (PrG). Thus, we establish certain elementary properties from the program text (or the specifications of the components) and then apply the derived rules to establish additional properties.

- Proof of (PrF), $rf \ \mapsto \ sh = \mathrm{F}$ in *sys*:
In the following proof all properties are in *sys*. The result follows by taking disjunction of the following three.

  $rf \ \wedge \ sh = \mathrm{G} \ \mapsto \ sh = \mathrm{F}$, proved below (s13)
  $rf \ \wedge \ sh = \mathrm{N} \ \mapsto \ sh = \mathrm{F}$, proved below (s14)
  $rf \ \wedge \ sh = \mathrm{F} \ \mapsto \ sh = \mathrm{F}$, implication □

- Proof of (s13), $rf \ \wedge \ sh = \mathrm{G} \ \mapsto \ sh = \mathrm{F}$:

  $\neg rg \ \wedge \ sh = \mathrm{G} \ \mapsto \ rg \ \wedge \ sh = \mathrm{G}$, PSP on (s1, s9)
  $\neg rg \ \wedge \ sh = \mathrm{G} \ \mapsto \ pr \ \wedge \ sh = \mathrm{N}$, transitivity on above and s10
  $sh = \mathrm{G} \ \mapsto \ pr \ \wedge \ sh = \mathrm{N}$, disjunction on above and s10
  $rf \ \wedge \ sh = \mathrm{G} \ \mapsto \ (rf \ \wedge \ pr \ \wedge \ sh = \mathrm{N}) \ \vee \ (sh = \mathrm{F})$
  , PSP on (above, s8)
  $rf \ \wedge \ sh = \mathrm{G} \ \mapsto \ sh = \mathrm{F}$, cancellation (above, s11) □

- Proof of (s14), $rf \ \wedge \ sh = \mathrm{N} \ \mapsto \ sh = \mathrm{F}$:

  $rg \ \wedge \ \neg pr \ \wedge \ sh = \mathrm{N} \ \mapsto \ sh = \mathrm{G}$, dual of (s11)
  $rf \ \wedge \ rg \ \wedge \ \neg pr \ \wedge \ sh = \mathrm{N} \ \mapsto \ (rf \ \wedge \ sh = \mathrm{G}) \vee \ (sh = \mathrm{F})$
  , PSP on (above, s8)
  $rf \ \wedge \ rg \ \wedge \ \neg pr \ \wedge \ sh = \mathrm{N} \ \mapsto \ sh = \mathrm{F}$, cancellation (above, s13)
  $rf \ \wedge \ rg \ \wedge \ pr \ \wedge \ sh = \mathrm{N} \ \mapsto \ sh = \mathrm{F}$, strengthen lhs on s11
  $rf \ \wedge \ rg \ \wedge \ sh = \mathrm{N} \ \mapsto \ sh = \mathrm{F}$, disjunction on above two
  $rf \ \wedge \ sh = \mathrm{N} \ \mapsto \ sh = \mathrm{F}$, cancellation (above, s12) □

8.3.4 Vending machine

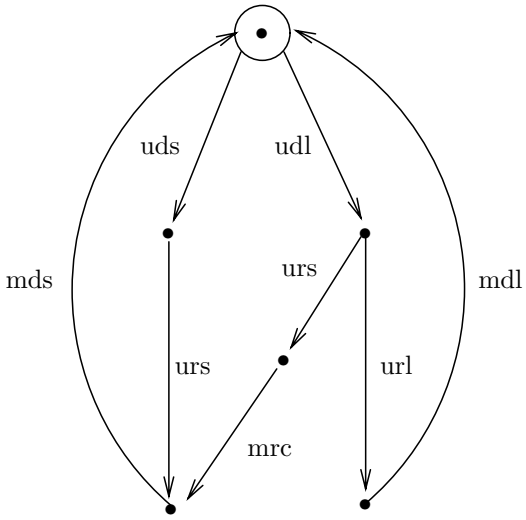
This example is inspired by Hoare [92]. A vending machine has a coin slot that accepts either a small coin or a large coin from a *user*. The machine has two buttons, one for requesting a small chocolate bar and the other for a large chocolate bar. If a button is pressed after depositing an adequate amount of money —a small coin for a small bar, a large coin for a large bar— then the machine dispenses the appropriate bar. If a large coin has been deposited and the button is pressed for a small bar, the machine dispenses a small bar and returns the appropriate change.

The behavior of the machine can be described by the finite state diagram of Fig. 8.3. Each node represents a state and each edge a transition; the initial state is denoted by a circle. The label on an edge is the name of the event that causes the transition. An event can be caused by the *user* —*uds*, for instance, denotes the *user* depositing a small coin— or the event can be generated by the vending machine —*mdl* denotes the machine dispensing a large chocolate bar. The caption in Fig. 8.3 gives the meaning of each event in the diagram.

Fig. 8.3 provides a succinct description of the allowable execution sequences. Thus, we can observe that $\langle udl \ url \ mdl \rangle$ is an allowable execution sequence whereas $\langle uds \ mdl \rangle$ is not. The small size of the specification makes it possible to trace all individual execution sequences and be assured of the "correctness of the specification".

Fig. 8.3 should be regarded as the specification of the entire system, consisting of *both* the *user* and the vending machine. Whenever components of a system are "tightly coupled" —an action in one component is followed by an action in another component, for instance— their behaviors are often best described by a single state diagram. In such cases, there is little advantage in separating the specifications of the individual components. However, for systems with even a few components the state space can become extraordinarily large. It is generally preferable to specify each component individually so that the state spaces remain manageable. For this example, we specify the *user* and the vending machine individually.

The interface between the *user* and the vending machine (henceforth called *vm*) consists of the following variables.



uds: user deposits small coin
udl: user deposits large coin
urs: user requests small chocolate
url: user requests large chocolate
mrc: machine returns change
mds: machine dispenses small chocolate
mdl: machine dispenses large chocolate

Figure 8.3: The state transitions in the vending machine.

cin: (s, l, ϕ) The value of *cin* is the amount of the coin in the input box of the machine. Here, s denotes a small coin, l a large coin, and ϕ the absence of any coin. This variable can be increased (from ϕ to s or l) by *user* alone; it may be decreased (to ϕ) by *vm* alone.

req: (s, l, ϕ) The value of *req* is the outstanding request for a chocolate bar; s is for a small bar, l for a large one, and ϕ denotes that there is no outstanding request for chocolate. Similar to *cin*, the value of *req* can be increased only by *user* and decreased only by *vm*.

dis: (s, l, ϕ) The value of *dis* is the size of the chocolate in the dispenser-tray: s for small, l for large, and ϕ for the absence of chocolate. This variable is increased (from ϕ to s or l) by *vm* and decreased (to ϕ) by *user* removing the dispensed chocolate.

cout: (boolean) The value of *cout* indicates if there is money in the coin return box. It is set to *true* only by *vm* and to *false* only by *user*.

Note We have simplified the original description by replacing two chocolate buttons by one. This abstraction prevents requesting a small and a large chocolate bar simultaneously. □

Define the following partial order over $\{\phi, s, l\}$:

$$\phi < s \quad \text{and} \quad \phi < l$$

This partial order is often called a "flat order"; here, ϕ is the "bottom" element and it is "smaller" than every other element; the remaining elements are unrelated by the partial order. The relations $>$, $\leq$, and $\geq$ have the appropriate meaning.

In the following, variables m and n take arbitrary values from $\{\phi, s, l\}$. The specification of *user* merely states how each of the interface variables —*cin*, *req*, *dis*, *cout*— can be changed by *user*. There is no progress requirement for *user*.

In *user* ::
stable $cin \geq m$
stable $req \geq m$
stable $dis \leq m$
stable $\neg cout$

The safety specification of *vm* is obtained similarly.

In *vm* ::
stable $cin \leq m$
stable $req \leq m$
stable $dis \geq m$
stable $cout$

Additionally, we have the progress requirement for *vm* that it has to dispense the appropriate chocolate and change. For all C and B from $\{\phi, s, l\}$

$$cin, req, dis, cout = C, B, \phi, false \ \wedge \ B \neq \phi \ \wedge \ C \geq B \ \mapsto$$
$$cin, req, dis, cout = \phi, \phi, B, C > B \ \text{ in } vm$$

Note on the specification The proposed specification is quite crude. The progress property of *vm* poses considerable technical difficulties; we explain the difficulties and propose a different specification later in this section. We modeled the chocolate dispenser tray by variable *dis*; this disallows *vm* from dispensing another chocolate until the previous one has been removed. A general dispenser tray can be modeled by a bag whose elements are s or l; *user* removes an element from the bag and *vm* adds to the bag. Similar remarks apply to *cout*. The given specification allows *user* to set the interface variables in arbitrary order. In particular, *user* may first request a chocolate and then deposit a coin. The specification in Fig. 8.3 disallows this possibility. □

Derivations of program properties

The main purpose of this subsection is to derive certain properties of the vending machine system by applying the union theorem. A secondary purpose is to show that certain intuitively plausible properties, justified by the informal description, cannot be established from the given formal specification.

First, observe that the progress property of *vm* —a *leads-to* property— is in a form that is not amenable to manipulation by the union theorem. Therefore, we postulate a stronger specification; we strengthen the property to an **ensures** property: for all C and B from $\{\phi, s, l\}$

$$\begin{array}{l} \mathit{cin}, \mathit{req}, \mathit{dis}, \mathit{cout} = C, B, \phi, \mathit{false} \ \wedge \ B \neq \phi \ \wedge \ C \geq B \ \textbf{en} \\ \qquad \mathit{cin}, \mathit{req}, \mathit{dis}, \mathit{cout} = \phi, \phi, B, C > B \ \text{ in } \mathit{vm} \end{array}$$

We establish that a small chocolate bar will be dispensed provided that a small coin is inserted, a small bar is requested, and both the chocolate dispenser tray and the output coin box are empty.

- Proof of $\mathit{cin}, \mathit{req}, \mathit{dis}, \mathit{cout} = s, s, \phi, \mathit{false} \ \mapsto \ \mathit{dis} = s$ in $\mathit{vm} \parallel \mathit{user}$:

  stable $\mathit{cin}, \mathit{req}, \mathit{dis}, \mathit{cout} = s, s, \phi, \mathit{false}$ in user
  , each conjunct is stable in user
  $\mathit{cin}, \mathit{req}, \mathit{dis}, \mathit{cout} = s, s, \phi, \mathit{false}$ **en**
  $\mathit{cin}, \mathit{req}, \mathit{dis}, \mathit{cout} = \phi, \phi, s, \mathit{false}$ in vm
  , progress property of vm with $C, B := s, s$
  $\mathit{cin}, \mathit{req}, \mathit{dis}, \mathit{cout} = s, s, \phi, \mathit{false}$ **en**
  $\mathit{cin}, \mathit{req}, \mathit{dis}, \mathit{cout} = \phi, \phi, s, \mathit{false}$ in $\mathit{vm} \parallel \mathit{user}$
  , union theorem corollary on above two
  $\mathit{cin}, \mathit{req}, \mathit{dis}, \mathit{cout} = s, s, \phi, \mathit{false} \ \mapsto \ \mathit{dis} = s$ in $\mathit{vm} \parallel \mathit{user}$
  , definition of $\mapsto$ and weaken rhs □

An intuitively plausible property of $\mathit{vm} \parallel \mathit{user}$ is

$$\mathit{cin}, \mathit{req} = s, s \ \mapsto \ \mathit{dis} = s$$

That is, a small chocolate bar will be dispensed if a small coin is inserted and a small bar is requested. However, this property does not hold in our system. In particular, if the dispenser tray already contains a large bar, i.e., $\mathit{dis} = l$, then *vm* is prevented from dispensing a small bar, i.e., setting *dis* to s, because

stable $\mathit{dis} \geq m$ for all m in $\{\phi, s, l\}$

Thus, *user* can break the system by not removing a single chocolate. We have discussed how to model the dispenser tray by a bag (see Note on the specification on page 261); the reader may wish to prove this progress property in the revised model.

A more involved property is

stable $cin, req = s, l$

That is, if *user* inserts a small coin and requests a large chocolate bar, the system is deadlocked! This is plausible because (1) *user* cannot insert additional coins nor can he cancel the request and (2) *vm* cannot honor this request. However, this property is not provable. The specification of *vm* does not prevent it from removing an inserted coin or canceling a button push. If desired, *vm*'s specification may be modified to state that *cin* (or *req*) is not modified until *req* (or *cin*) has the appropriate value.

8.3.5 Message communication

An asynchronous message-communicating system is a program where the components are the individual processes and the global variables represent the contents of the channels along which communications take place. For instance, let process F send messages to process G along a fifo channel C; the system is $F \parallel G$ and the global variable (declared in both F and G) is a sequence (of messages) C. Program F "sends" by appending messages to the end of C and program G "receives" by removing the message at the head of C, provided that C is non-null. Data pipelining can be regarded as message communication over a network of processes. In particular, let F_i send messages in fifo order to F_{i+1}, $0 \leq i < N$, and C_i be the channel directed from F_i to F_{i+1}. Then C_i is a sequence of messages and C_i is declared as a global variable in F_i and F_{i+1}. The system is $\langle \parallel i :: F_i \rangle$.

Channels between the processes need not be fifo. An unordered channel —see section 4.1.3— may be implemented by a bag of messages. Also, a bounded fifo channel C is implemented by a bounded sequence (see section 4.1.2); the sender can append a message to C only if the length of C is lower than the bound. Other forms of asynchronous communication —for instance, where each transmission is acknowledged— can be similarly modeled. Of particular interest is the transmission of a "signal"; a signal has no associated data and a new signal can be sent only after the previous one has been acknowledged. Signal transmission and acknowledgment can be modeled using a global boolean variable that is set to *true* to indicate signal transmission and *false* to indicate acknowledgment. We have used this scheme in the semaphore program —section 8.3.3— where variable rf, for instance, was set to *true* by program F to request an operation and set to *false* by program *semf* to indicate the completion of that operation.

Synchronous message communication, where the send and receive operations are executed simultaneously [92], cannot be easily modeled using the union operator.

A more elaborate example of message communication is treated next.

A client-server network

A set of *client* processes —$CL_0, \ldots, CL_j$— request service from a set of *server* processes —$SR_0, \ldots, SR_K$— during a computation. Any *server* can serve any *client*. A typical example is a set of user processes (*client*s) connected to a set of printers (*server*s) where the print request of any user can be satisfied by any printer.

We propose a communication network which connects *client*s and *server*s. Each channel in the network carries requests; further, a channel contains at most one request at any time. Therefore, a channel can be modeled by a (global) variable whose value is either a request or ϕ; ϕ denotes that the corresponding channel is empty.

The protocol for accessing such a variable is as follows: a "send" on the channel is simulated by writing into the corresponding variable provided that its current value is ϕ; a "receive" is simulated by reading the corresponding variable value and then setting its value to ϕ. A similar model of communication —called *word*— is proposed on page 60.

An interconnection network for the *clients* and the *servers* is shown in Fig. 8.4. The components of the system are the following.

*Client*s, $CL_j, 0 \leq j \leq J$: *Client* CL_j sends its requests using global variable r_j. As described above, CL_j may store a request in r_j provided that the value of r_j is ϕ.

Client-Manager, CM: This process merges the requests of the individual *Client*s and issues one request at a time to the *Server*s using variable r. It reads r_js (reading an r_j is accompanied by setting it to ϕ) in some order and writes into variable r.

Buffer, *BUFF*: This process implements a buffer of N requests, $N \geq 0$, in order to smooth out the speed variations between the clients and the servers. The process reads from r and writes into w.

Server-Manager, SM: This process receives a stream of requests through global variable w, and it distributes the requests among the w_ks. The policy for distribution need not be fair; it may depend on the number and types of the servers.

*Server*s, $SR_k, 0 \leq k \leq K$: *Server* SR_k receives requests from global variable w_k (as described above, SR_k sets w_k to ϕ after reading a request).

The entire system is

$$\langle [\!] \ j : \ 0 \leq j \leq J : \ CL_j \rangle \ [\!] \ CM \ [\!] \ BUFF \ [\!] \ SM \ [\!] \ \langle [\!] \ k : \ 0 \leq k \leq K : \ SR_k \rangle$$

and the variables —r_j and w_k, for all j and k, and r and w— may be declared local to this system.

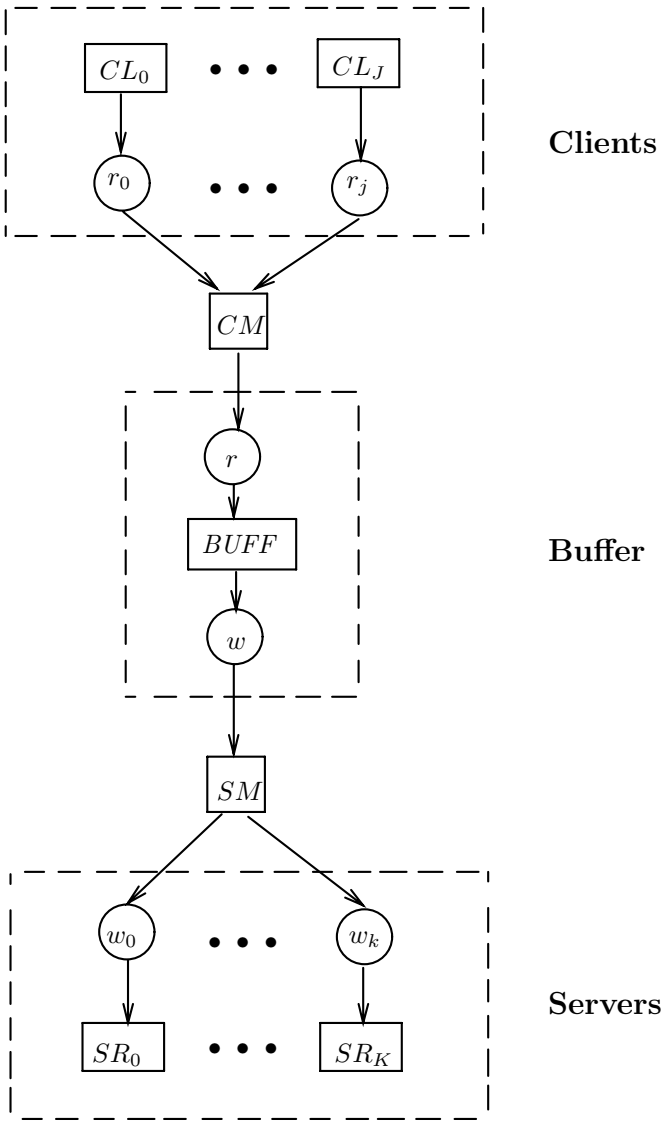


Figure 8.4: A *Client-Server* interconnection network: Process *CM* shown is *Client-Manager* and *SM* is *Server-Manager*.

8.4 Substitution Axiom under Union

Substitution axiom, described in section 5.4.3, allows us to replace an invariant by *true* and vice versa in any property of a program. This axiom has proved to be extremely useful in practice. However, care must be exercised when it is applied to unions of programs.

Any invariant of $F \parallel G$ can still be replaced by *true* (and vice versa) in any property of F, G, or $F \parallel G$. However, it is not permissible to use the substitution axiom with an invariant of F on a property of $F \parallel G$. To see this, consider programs F and G that share two global boolean variables, p and q. Initially, p and q are *true* in F, and the only action in F is $q := p$. Then, $p \wedge q$ is invariant in F, and using the substitution axiom in F, q is invariant in F.

Now suppose G does not modify q. Then q is stable in G. Therefore, q is invariant in $F \parallel G$, according to the corollary of the union theorem. This conclusion is invalid, as can be seen by considering program G that has the action $p := \neg p$. Then, in a state where p is *false*, an execution of $q := p$ in F falsifies q.

The invalid step in this argument is to apply the substitution axiom to a property of $F \parallel G$ using **invariant** q in F. This invariant is established using the substitution axiom with an invariant of F alone. Therefore, it cannot be used to establish a property of a larger program of which F is a component.

8.5 Theoretical Issues

We propose certain axioms that any asynchronous composition operator like union should satisfy. Also, we study what it means for one program to *refine* another. Two programs are taken to be *equivalent* if they refine each other. Our approach is based on identifying a program with its properties; a refined program has all the properties of the program it refines. We propose two different definitions of refinement. We show that only one of the notions of refinement (and equivalence) is preserved under union, i.e., if F and G are equivalent, so are $F \parallel H$ and $G \parallel H$ for any H. This demonstration shows why **transient** and **en** are important concepts, particularly in the study of asynchronous composition.

8.5.1 Axioms of union

It is clear from the description that union is commutative and associative; further, there is a program —which has no variable and whose only action is *skip*— that serves as the "identity" element of union. Formally, we pos-

tulate the following as axioms for the union operator. For programs F, G, and H:

- (Commutativity) $F \parallel G = G \parallel F$, if either side is defined.
- (Associativity) $(F \parallel G) \parallel H = F \parallel (G \parallel H)$, if either side is defined.
- (Existence of id) There is a program id such that $F = F \parallel id$.

We have not yet defined the meaning of equality of programs. We take up this question next and study its relationship to union. Henceforth, we write "$\lessRightarrow$" instead of "=" to denote equality of programs.

8.5.2 A definition of refinement

For a program F, let $F.\mathbf{co}$ be the set of pairs (p, q) such that

$$p \ \mathbf{co} \ q \ \text{ in } F$$

Let $F.\mathbf{transient}$ be the set of predicates p such that p is transient in F.

The following are restatements of the appropriate parts of the union theorem. For programs F and G,

$$(F \parallel G).\mathbf{co} = F.\mathbf{co} \ \cap \ G.\mathbf{co}$$
$$(F \parallel G).\mathbf{transient} \ = F.\mathbf{transient} \ \cup \ G.\mathbf{transient}$$

Next we define the refinement relation among programs. Program F is *refined* by G, written as $F \Rightarrow G$, if all of the following hold.

$$G.IC \ \Rightarrow \ F.IC$$
$$F.\mathbf{co} \ \subseteq \ G.\mathbf{co}$$
$$F.\mathbf{transient} \ \subseteq \ G.\mathbf{transient}$$

That is, G's initial condition is stronger than F's and G has all the **co**-properties and the transient predicates of F. Define $F \lessRightarrow G$ to mean that $F \Rightarrow G$ and $G \Rightarrow F$.

As a small exercise, given $F \Rightarrow G$, we show that any invariant of F is an invariant of G.

$$\begin{array}{ll} & \mathbf{invariant}\ p \ \text{ in } F \\ \equiv & \quad \{\text{definition of invariant}\} \\ & (F.IC \ \Rightarrow \ p),\ \mathbf{stable}\ p \ \text{ in } F \\ \Rightarrow & \quad \{\text{from } F \Rightarrow G,\ G.IC \ \Rightarrow \ F.IC \text{ and } \mathbf{stable}\ p \ \text{ in } G\} \\ & (G.IC \ \Rightarrow \ p),\ \mathbf{stable}\ p \ \text{ in } G \\ \equiv & \quad \{\text{definition of invariant}\} \\ & \mathbf{invariant}\ p \ \text{ in } G \end{array}$$

From its definition, $\Rightarrow$ is reflexive and transitive. Further, $\Rightarrow$ is preserved under union, as shown in the following theorem.

Theorem Given that $(F \Rrightarrow G)$ and $F \parallel H$ and $G \parallel H$ are defined, $(F \parallel H \Rrightarrow G \parallel H)$.
Proof:

Proof of $(G \parallel H).IC \Rightarrow (F \parallel H).IC$:

$$
\begin{array}{ll}
 & (G \parallel H).IC \\
\equiv & \quad \{\text{union theorem, condition (1)}\} \\
 & G.IC \wedge H.IC \\
\Rightarrow & \quad \{G.IC \Rightarrow F.IC, \text{ from } F \Rrightarrow G\} \\
 & F.IC \wedge H.IC \\
\equiv & \quad \{\text{union theorem, condition (1)}\} \\
 & (F \parallel H).IC
\end{array}
$$

Proof of $(F \parallel H).\mathbf{co} \subseteq (G \parallel H).\mathbf{co}$:

$$
\begin{array}{ll}
 & (F \parallel H).\mathbf{co} \\
= & \quad \{\text{union theorem, condition (2)}\} \\
 & F.\mathbf{co} \cap H.\mathbf{co} \\
\subseteq & \quad \{F.\mathbf{co} \subseteq G.\mathbf{co}, \text{ from } F \Rrightarrow G\} \\
 & G.\mathbf{co} \cap H.\mathbf{co} \\
= & \quad \{\text{union theorem, condition (2)}\} \\
 & (G \parallel H).\mathbf{co}
\end{array}
$$

Proof of $(F \parallel H).\mathbf{transient} \subseteq (G \parallel H).\mathbf{transient}$:

$$
\begin{array}{ll}
 & (F \parallel H).\mathbf{transient} \\
= & \quad \{\text{union theorem, condition (3)}\} \\
 & F.\mathbf{transient} \cup H.\mathbf{transient} \\
\subseteq & \quad \{F.\mathbf{transient} \subseteq G.\mathbf{transient}, \text{ from } F \Rrightarrow G\} \\
 & G.\mathbf{transient} \cup H.\mathbf{transient} \\
= & \quad \{\text{union theorem, condition (3)}\} \\
 & (G \parallel H).\mathbf{transient}
\end{array}
$$

□

Corollary If $F \Lleftrightarrow G$ then $F \parallel H \Lleftrightarrow G \parallel H$. □

Note The definition of refinement requires, implicitly, that a program name the same variables as the program it refines. A more general approach is to permit "data refinements" in which a group of variables is replaced by another group; the relation between the two groups of variables can be given by an invariant relation. See Knapp [108] for a discussion of refinement along these lines. □

$x =$	0	1	2
$\alpha.x =$	2	1	2
$\beta.x =$	1	2	2

Table 8.1: Actions of Program F

$x =$	0	1	2
$\alpha'.x =$	2	2	2
$\beta'.x =$	1	1	2

Table 8.2: Actions of Program G

8.5.3 Alternative definition of refinement

We propose an alternative definition of refinement that is weaker (more general) than the previous one. For programs F and G, define $F \Rrightarrow G$ by

$$G.IC \;\Rightarrow\; F.IC \;,\; F.\mathbf{co} \;\subseteq\; G.\mathbf{co}, \text{ and } F.lt \;\subseteq\; G.lt$$

where $F.lt$ is the set of pairs (p, q) where

$$p \;\mapsto\; q \text{ in } F$$

As before, $F \Lleftrightarrow G$ means that $F \Rrightarrow G$ and $G \Rrightarrow F$. Clearly, $\Rrightarrow$ is reflexive and transitive and $\Lleftrightarrow$ is an equivalence relation. However, it is no longer true that for any H,

$$\text{if } (F \Rrightarrow G) \text{ then } (F \,[\!]\, H \;\Rrightarrow\; G \,[\!]\, H)$$

or even

$$\text{if } (F \Lleftrightarrow G) \text{ then } (F \,[\!]\, H \;\Lleftrightarrow\; G \,[\!]\, H)$$

We give a counterexample to the last proposition by showing three programs F, G, and H where $F \Lleftrightarrow G$ but $F \,[\!]\, H \;\Lleftrightarrow\; G \,[\!]\, H$ does not hold. Let F, G, and H access a common integer variable x that takes on three possible values: 0, 1, or 2. Initially, $x = 0$ in F, G, and H. Program F has two actions α and β, where the values of x after executing α and β are shown in Table 8.1. Similarly, program G has two actions —α' and β'— whose effects are given in Table 8.2.

Programs F and G have the same set of **co**-properties because for any state (i.e., any particular value of x), the set of possible next states in F and G are identical; this can be seen by comparing Tables 8.1 and 8.2 column by column. Now, the fair executions of F and G yield identical sequences of states; in both cases, the possible sequences of states are given by $0\ 1^*\ 2^\infty$; that is, initially $x = 0$, x is 1 for a finite number of steps, x eventually becomes 2, and it remains 2 forever. Since the sequence of states in fair executions of F and G are identical, $F.lt = G.lt$. Therefore, $F \Lleftrightarrow G$, according to the proposed definition.

$x =$	0	1	2
$\gamma.x =$	0	0	2

Table 8.3: Actions of Program H

Consider a program H that has one action γ, as defined in Table 8.3. We show that $(F \parallel H).lt \neq (G \parallel H).lt$. Therefore, $(F \parallel H) \iff (G \parallel H)$ does not hold.

- Proof of $x = 0 \mapsto x = 2$ in $G \parallel H$:

  transient $x = 0 \vee x = 1$ in G , from Table 8.2
  transient $x = 0 \vee x = 1$ in $G \parallel H$, union theorem (part 3)
  $x = 0 \vee x = 1$ **en** $x = 2$ in $G \parallel H$, definition of **en**
  $x = 0 \vee x = 1 \mapsto x = 2$ in $G \parallel H$, basis rule of $\mapsto$
  $x = 0 \mapsto x = 2$ in $G \parallel H$, strengthen lhs □

- Proof that $x = 0 \mapsto x = 2$ in $F \parallel H$ does not hold:
  We display an execution of $F \parallel H$, starting in state $x = 0$, in which $x \neq 2$ holds at all times. Consider the fair execution sequence $(\beta\alpha\gamma)^\infty$ starting in $x = 0$; the sequence of states is $(0\ 1\ 1)^\infty$.

Discussion

The notion of transient predicate encapsulates "atomic actions". The definition in section 8.5.2 suggests that any refinement of a component that preserves atomicity also refines an asynchronous system. The weaker notion of refinement, as given in section 8.5.3, is not required to preserve atomicity, and the counterexample shows that the refinement of a component may not refine the system. Thus, the notion of atomicity —as embodied in **transient** and **en**— seems crucial in refinements of asynchronous systems.

There are occasions where we prefer to use the weaker notion of refinement (given in section 8.5.3); in such cases, we are careful to note that the entire system should be refined as a whole, not component by component.

8.6 Concluding Remarks

The most common operator for asynchronous compositions of programs is union. Many of the properties of $F \parallel G$ can be derived from the components, F and G, by applying the union theorem and its corollaries. The major exception are the *leads-to* properties. We have given a theorem —union theorem for progress, on page 246— for deductions of such properties, but it requires us to specify how the global variables are modified by each component.

In chapter 9, we develop theories that simplify deductions of properties from component specifications, especially for *leads-to*.

8.7 Bibliographic Notes

Program composition is a recurring theme in software engineering; Dahl, Dijkstra and Hoare [51] is an early classic on this subject. Some of the earliest formal treatment of structuring operators for concurrency appear in Hoare [92] and Milner [132].

The union operator of this chapter is from Chandy and Misra [32]; similar operators had been introduced earlier in CSP [92] and CCS [132]. The union theorem for progress is based on Singh [164] and Misra [134, note 17]. The vending machine example is inspired by Hoare [92]. Some of the difficulties regarding application of the substitution axiom (section 8.4) are noted in Sanders [158], Knapp [108] and Misra [134, note 14]. The example in section 8.5.3 is based on Misra [134, note 11]. Paulson [149] has mechanically verified several nontrivial action systems, developed by Chandy and Sanders [35] and Charpentier and Chandy [37], that are built through composition.

The compositional issues that arise in large-scale systems are significantly more complex than the ones discussed in this chapter; see Jackson and Zave [96] for an example from telephony that addresses feature modularity, structured feature composition, and analysis of feature interactions.

8.8 Exercises

1. A box F consists of the following actions.

$$\begin{array}{lll} b & \rightarrow & x := 0 \\ [\!] \ \neg b & \rightarrow & x := 1 \end{array}$$

   where x is local to F and b is global.

   Prove or disprove the following properties. Here G is any program other than F.

   (a) $(\textbf{stable}\ b \text{ in } G) \ \Rightarrow\ (\textbf{stable}\ b \ \wedge\ x = 0 \ \text{ in } F [\!] G)$

   (b) $(\textbf{stable}\ b \ \text{ in } G) \ \Rightarrow\ (\textbf{stable}\ x = 0 \text{ in } F [\!] G)$

   (c) $(\textbf{stable}\ b \text{ in } G) \ \Rightarrow\ (b \ \mapsto\ x = 0 \ \text{ in } F [\!] G)$

   (d) $(\mathit{true} \ \mapsto\ b \ \text{ in } G) \ \Rightarrow\ (\mathit{true} \ \mapsto\ x = 0 \ \text{ in } F [\!] G)$

2. Consider box F given below.

---

box F
  global integer y;
  local integer x;

    $x := x - 1$
  $[\!]$ $x \leq 0 \ \wedge \ y > 0 \ \rightarrow \ y := -y$
end $\{F\}$

Suppose **stable** $y \neq 0$ in G. Prove that $y \neq 0 \ \mapsto \ y < 0$ in $F \ [\!] \ G$.

3. (Modeling) The states of a *client* process C are either *computing* (sometimes called *thinking*), *waiting* for a resource (called *hungry*), or *using* the resource (called *eating*). The behavior of the *client* is given by (1) a computing process may transit only to waiting, (2) a waiting process remains waiting, (3) a process using the resource eventually transits to computing in one atomic step. A *server* process S can affect the states of a *client* as follows: it can only change a waiting *client* to one using the resource; it can cause no other state transition.

   State the given properties formally. Show that in $C \ [\!] \ S$ a process (1) remains computing unless it starts waiting, (2) remains waiting unless it starts using the resource, (3) continues using the resource unless it starts computing. Also, (4) the last transition eventually happens in one atomic step.

4. Let boxes F and G share only a single global variable x. Suppose that G can only read the value of x, not write into it. Which of the following holds? Here, p and q name only the variables of F (including x) and r and s the variables of G (including x) .

$$\frac{p \ \textbf{co} \ q \ \text{ in } F}{p \ \textbf{co} \ q \ \text{ in } F \ [\!] \ G} \quad \frac{p \ \textbf{en} \ q \ \text{ in } F}{p \ \textbf{en} \ q \ \text{ in } F \ [\!] \ G} \quad \frac{p \ \mapsto \ q \ \text{ in } F}{p \ \mapsto \ q \ \text{ in } F \ [\!] \ G}$$

$$\frac{r \ \textbf{co} \ s \ \text{ in } G}{r \ \textbf{co} \ s \ \text{ in } F \ [\!] \ G} \quad \frac{r \ \textbf{en} \ s \ \text{ in } G}{r \ \textbf{en} \ s \ \text{ in } F \ [\!] \ G} \quad \frac{r \ \mapsto \ s \ \text{ in } G}{r \ \mapsto \ s \ \text{ in } F \ [\!] \ G}$$

   Suppose further that F cannot modify x either. Which of the above hold?

5. Show the following (where $G.FP$ is the fixed point predicate of G). Use stability at fixed point rule (see page 100).

   (a) $$\frac{p \ \textbf{co} \ q \ \text{ in } F}{p \wedge G.FP \ \textbf{co} \ q \ \text{ in } F \ [\!] \ G}$$

(b)
$$\frac{p \ \textbf{en} \ q \ \text{in} \ F}{p \ \textbf{en} \ q \vee \neg G.FP \ \text{in} \ F \ [\!] \ G}$$

(c)
$$\frac{p \ \mapsto \ q \ \text{in} \ F}{p \ \mapsto \ q \vee \neg G.FP \ \text{in} \ F \ [\!] \ G}$$

(d) Using exercise (5c) show that

$$\frac{\begin{array}{c} p \ \mapsto \ q \ \text{in} \ F \\ r \ \Rightarrow \ G.FP \\ \textbf{stable} \ r \ \text{in} \ F \end{array}}{p \wedge r \ \mapsto \ q \wedge r \ \text{in} \ F \ [\!] \ G}$$

6. Give a counterexample to the following conjecture. Here the variables and relations have the same meaning as in the union theorem for progress on page 246.

$$\frac{\begin{array}{lcl} p \ \wedge \ x = m & \mapsto & (p \ \wedge \ x < m) \ \vee \ q \ \text{in} \ F \\ p \ \wedge \ x = m & \textbf{co} & (p \ \wedge \ x \leq m) \ \vee \ q \ \text{in} \ G \end{array}}{p \ \mapsto \ q \ \text{in} \ F \ [\!] \ G}$$

7. Consider boxes *semf1* and *semg1* of section 8.3.3. Show that

$$rf \vee rg \ \mapsto \ sh \neq N \ \text{in} \ semf1 \ [\!] \ semg1$$

8. (Semaphore) It is required to implement a strong semaphore to be shared among N processes. Sketch a generalized version of the program given on page 255.

9. We elaborate on the interface of box *BUFF* described as a part of the *Client-Server* network in section 8.3.5. The box in this exercise implements a one-place buffer as follows. *BUFF* reads data from variable r and writes the data into variable w, all in one step (it has no additional internal storage). Variable r (and w) has value ϕ when it holds no useful data. Box *BUFF* obeys the following protocol: whenever it reads any data from r it sets r to ϕ, and it writes into w only if $w = \phi$ (i.e., when w holds no data). It is expected that box *CM* (in the role of producer) writes into r only if $r = \phi$ and *SM* (in the role of consumer) sets w to ϕ after reading data from it.

   Propose a specification of *BUFF*. Using your specification and assuming that

$$\textbf{stable} \ r \neq \phi \ \text{in} \ CM \ [\!] \ SM$$
$$\textbf{stable} \ w = \phi \ \text{in} \ CM \ [\!] \ SM$$

   show that

$$w = \phi \ \mapsto \ r = \phi \ \text{in} \ BUFF \ [\!] \ CM \ [\!] \ SM$$
$$r \neq \phi \ \mapsto \ w \neq \phi \ \text{in} \ BUFF \ [\!] \ CM \ [\!] \ SM$$

10. Given programs F, G, H, and J where $F \parallel H$, $G \parallel J$, and $G \parallel H$ are defined, show that

$$\frac{F \mathrel{\Rrightarrow} G \ , \ H \mathrel{\Rrightarrow} J}{F \parallel H \ \mathrel{\Rrightarrow} \ G \parallel J}$$

Use the definition of $\mathrel{\Rrightarrow}$ given in section 8.5.2.

11. (Program refinement by strengthening a guard) This exercise gives a condition under which the guard of an action may be strengthened while preserving all the safety and *leads-to* properties. Let α be an action with guard q and β be the same action with the guard replaced by p. Let F be a program in which p *tracks* q (see exercise 18 of chapter 6 for definition of *tracks*). Show that every **co** and *leads-to* property of $F \parallel \alpha$ is also a property of $F \parallel \beta$. Note that **en** properties are not necessarily preserved.

8.9 Solutions to Exercises

1. (a) The proof is as follows.

stable b in G	, premise
stable $x = 0$ in G	, locality axiom
stable $b \ \wedge \ x = 0$ in G	, conjunction
stable $b \ \wedge \ x = 0$ in F	, text of F
stable $b \ \wedge \ x = 0$ in $F \parallel G$	, union theorem corollary

(b) This conjecture is false. In a state where $\neg b \ \wedge \ x = 0$ holds, executing the second action of F sets x to 1.

(c) The proof is as follows.

b **en** $x = 0$ in F	, text of F
stable b in G	, premise
b **en** $x = 0$ in $F \parallel G$	, union theorem corollary
$b \ \mapsto \ x = 0$ in $F \parallel G$	, definition of $\mapsto$

(d) This conjecture is false. Let there be two actions in G:

$b := \mathit{true}$
$\parallel \ b := \mathit{false}$

Clearly, $\mathit{true} \ \mapsto \ b$ in G. The following execution sequence of $F \parallel G$ is a counterexample to $\mathit{true} \ \mapsto \ x = 0$ in $F \parallel G$.

$\{x = 1\}$
loop
$\{x = 1\}$
$b := \mathit{true}$ (in G) $\{x = 1\}$;
$b := \mathit{false}$ (in G) $\{\neg b \ \wedge \ x = 1\}$;
$b \ \rightarrow \ x := 0$ (in F) $\{\neg b \ \wedge \ x = 1\}$;
$\neg b \ \rightarrow \ x := 1$ (in F) $\{\neg b \ \wedge \ x = 1\}$
forever

2. Proof of $y \neq 0 \ \mapsto \ y < 0$ in $F \parallel G$:

$y \neq 0 \ \mapsto \ x \leq 0 \ \wedge \ y \neq 0$ in $F \parallel G$, proved below (1)
$x \leq 0 \ \wedge \ y \neq 0 \ \mapsto \ y < 0$ in $F \parallel G$, proved below (2)
$y \neq 0 \ \mapsto \ y < 0$ in $F \parallel G$, transitivity on (1,2)

Proof of (1) $y \neq 0 \ \mapsto \ x \leq 0 \ \wedge \ y \neq 0$ in $F \parallel G$:

stable $y \neq 0$ in F , text of F
stable $y \neq 0$ in G , given
stable $y \neq 0$ in $F \parallel G$, union theorem
$\mathit{true} \ \mapsto \ x \leq 0$ in $F \parallel G$, proved below (1.1)
$y \neq 0 \ \mapsto \ x \leq 0 \ \wedge \ y \neq 0$ in $F \parallel G$, stable conjunction on the above two

Proof of (2) $x \leq 0 \ \wedge \ y \neq 0 \ \mapsto \ y < 0$ in $F \parallel G$:

stable $y \neq 0$ in G , given
stable $x \leq 0$ in G , locality axiom
stable $x \leq 0 \ \wedge \ y \neq 0$ in G , stable conjunction
$x \leq 0 \ \wedge \ y \neq 0$ **en** $y < 0$ in F , text of F
$x \leq 0 \ \wedge \ y \neq 0$ **en** $y < 0$ in $F \parallel G$, union theorem corollary
$x \leq 0 \ \wedge \ y \neq 0 \ \mapsto \ y < 0$ in $F \parallel G$, definition of $\mapsto$

Proof of (1.1) $\mathit{true} \ \mapsto \ x \leq 0$ in $F \parallel G$:

For any integer k,
$x = k$ **en** $x < k$ in F , text of F
stable $x = k$ in G , locality axiom
$x = k$ **en** $x < k$ in $F \parallel G$, union theorem corollary
$x = k \ \mapsto \ x < k$ in $F \parallel G$, definition of $\mapsto$
$\mathit{true} \ \mapsto \ x \leq 0$ in $F \parallel G$, induction on integers

3. Let t, h, e denote that the *client* is computing, waiting or using. Clearly,

$t \vee h \vee e \ \equiv \ \mathit{true}$ (ST1)
$\neg(t \wedge h) \ , \ \neg(h \wedge e) \ , \ \neg(t \wedge e)$ (ST2)

The specification of the *client* is

t **co** $t \vee h$ in C (C1)
stable h in C (C2)

e **co** $e \vee t$ in C (C3)
transient e in C (C4)

The specification of the *server* is

constant t in S (S1)
stable e in S (S2)

Proof of t **co** $t \vee h$ in $C \parallel S$:
stable t in S , from S1
t **co** $t \vee h$ in $C \parallel S$, union theorem corollary on (C1, above)

Proof of h **co** $h \vee e$ in $C \parallel S$:
stable $\neg t$ in S , from S1
stable $h \vee e$ in S , $\neg t \equiv h \vee e$ from (ST1, ST2)
h **co** $h \vee e$ in S , strengthen lhs
h **co** $h \vee e$ in $C \parallel S$, union theorem corollary on (C2, above)

Proof of e **co** $e \vee t$ in $C \parallel S$: union theorem corollary on (C3, S2).

Proof of **transient** e in $C \parallel S$: union theorem on (C4).

4. Since G can only read from x, predicate p is stable in G. Using the union theorem corollary,

$$\frac{p \textbf{ co } q \text{ in } F}{p \textbf{ co } q \text{ in } F \parallel G} \quad \text{and} \quad \frac{p \textbf{ en } q \text{ in } F}{p \textbf{ en } q \text{ in } F \parallel G}$$

Also, applying induction on the structure of $p \mapsto q$ in F, we can show

$$\frac{p \mapsto q \text{ in } F}{p \mapsto q \text{ in } F \parallel G}$$

Thus, G interferes not at all with F's execution.

However, none of the properties of G are necessarily inherited by $F \parallel G$. To see this, let x be a boolean variable and G have the action $y := x$, where y is a local boolean variable. From the text of G,

stable x in G
x **en** y in G
$x \mapsto y$ in G

However, none of the above is a property of $F \parallel G$ if F has the action $x := \neg x$.

If neither F nor G modifies x, applying the argument given above, every property of a component is a property of the system.

5. (a)
p **co** q in F , premise
$p \wedge G.FP$ **co** q in F , strengthen lhs
stable $p \wedge G.FP$ in G , stability at fixed point in G
$p \wedge G.FP$ **co** q in $F \parallel G$, union theorem corollary

(b)
p **en** q in F , premise
p **en** $q \vee \neg G.FP$ in F , exercise (4b) of chapter 6
$p \wedge G.FP$ **en** $q \vee \neg G.FP$ in F , exercise (4c) of chapter 6
stable $p \wedge G.FP$ in G , stability at fixed point
$p \wedge G.FP$ **en** $q \vee \neg G.FP$ in $F \parallel G$
, union theorem corollary
p **en** $q \vee \neg G.FP$ in $F \parallel G$, exercise (4c) of chapter 6

(c) Proof is by induction on the structure of $p \mapsto q$ in F.

- p **en** q in F: see solution to exercise (5b)

- $p \mapsto r$ in F, $r \mapsto q$ in F:
$p \mapsto r \vee \neg G.FP$ in $F \parallel G$, induction hypothesis
$r \mapsto q \vee \neg G.FP$ in $F \parallel G$, induction hypothesis
$p \mapsto q \vee \neg G.FP$ in $F \parallel G$, cancellation on the above two

- $p = \langle \exists\ i ::\ p_i \rangle$ where $\langle \forall\ i ::\ p_i \mapsto q$ in $F \rangle$:
$\langle \forall\ i ::\ p_i \mapsto q \vee \neg G.FP \rangle$ in $F \parallel G$
, induction hypothesis
$p \mapsto q \vee \neg G.FP$ in $F \parallel G$, disjunction on above: use $p = \langle \exists\ i ::\ p_i \rangle$

(d)
stable $r \wedge G.FP$ in G , stability at fixed point
stable r in G , from above and $r \Rightarrow G.FP$
stable r in $F \parallel G$, from above and **stable** r in F
$p \mapsto q \vee \neg G.FP$ in $F \parallel G$, using (5c) on $p \mapsto q$ in F
$p \wedge r \mapsto q \wedge r$ in $F \parallel G$, PSP on above two: use $r \Rightarrow G.FP$

6. Let F and G share a global variable x that assumes values from {0,1,2}; let the well-founded order be defined by $0 < 1 < 2$. Program F consists of a single action, $x := (x + 1)\ mod\ 3$, and G consists of $x := 1$. Let p be $x = 1$ and q be $x = 0$. Then, for all m, $m \in \{0, 1, 2\}$,

$p \wedge x = m \mapsto (p \wedge x < m) \vee q$ in F
(i.e., $x = 1 \mapsto x = 0$ in F) and
$p \wedge x = m$ **co** $(p \wedge x \leq m) \vee q$ in G
(i.e., $x = 1$ **co** $x = 1 \vee x = 0$ in G) .

However, $p \mapsto q$ in $F \,[\!]\, G$
(i.e., $x = 1 \mapsto x = 0$ in $F \,[\!]\, G$)

does not hold: consider alternate executions of the actions of F and G starting in state $x = 1$.

7. Show that

$$rf \ \textbf{en} \ sh \neq \mathrm{N} \text{ in } semf1 \,[\!]\, semg1$$

by applying the union theorem to

$$rf \wedge sh = \mathrm{N} \ \textbf{co} \ rf \vee sh \neq \mathrm{N} \text{ in } semf1$$
$$rf \wedge sh = \mathrm{N} \ \textbf{co} \ rf \vee sh \neq \mathrm{N} \text{ in } semf2$$
$$\textbf{transient} \ rf \wedge sh = \mathrm{N} \text{ in } semf1$$

Similarly,
$rg \ \textbf{en} \ sh \neq \mathrm{N}$ in $semf1 \,[\!]\, semg1$

The desired result follows by converting the two **ensures** properties to *leads-to*s and taking their disjunction.

8. Let the processes be numbered 1 through N. Variable sh assumes a value between 0 and N; $sh = 0$ denotes that the semaphore has not been granted to any process; for $i \neq 0$, $sh = i$ denotes that process i holds the semaphore.

Let $pr.i$, an integer, be the priority of process i. The strategy we adopt is that a process that is granted the semaphore is assigned the lowest priority. (The fair task scheduler in section 11.6 uses a similar strategy.)

Let low be an integer smaller than any $pr.i$. Boolean variable $r.i$, for process i, has the same role as rf or rg. The box that interacts with process i, $1 \leq i \leq N$, is shown below.

---

box *sem.i*
  global boolean $r.i$;
  global integer $pr.i = i$;
  global integer $low, sh = 0, 0$;

    P:: $r.i \wedge sh = 0 \wedge \langle \forall\, j :: (pr.j \leq pr.i) \vee \neg r.j \rangle \rightarrow$
        $pr.i, low, r.i, sh := low, low - 1, false, i$
  [] V:: $r.i \wedge sh = i \rightarrow r.i, sh := false, 0$
end $\{sem.i\}$

---

9. We abbreviate $BUFF$ to B and $CM \parallel SM$ to M. The safety property of B is that the only allowable change in r and w is to transfer data from r to w setting r to ϕ; this change is allowed provided that $w = \phi$ prior to the change. If $r = \phi$ or $w \neq \phi$, both r and w retain their values. Thus, for any X and Y ranging over data values and ϕ,

$$r, w = X, Y \ \ \textbf{co} \ \ r, w = X, Y \ \vee \ r, w, Y = \phi, X, \phi \ \text{ in } B \qquad \text{(B1)}$$

The progress property is that the state $r \neq \phi \ \wedge \ w = \phi$ does not persist because the data will be moved from r to w.

$$\textbf{transient } r \neq \phi \ \wedge \ w = \phi \ \text{ in } B \qquad \text{(B2)}$$

Assuming

$$\textbf{stable } r \neq \phi \ \text{ in } M \qquad \text{(M1)}$$
$$\textbf{stable } w = \phi \ \text{ in } M \qquad \text{(M2)}$$

we first prove that

$$r \neq \phi \ \wedge \ w = \phi \ \ \textbf{co} \ \ r \neq \phi \ \equiv \ w = \phi \ \text{ in } B \parallel M \qquad \text{(BM1)}$$
$$\textbf{transient } r \neq \phi \ \wedge \ w = \phi \ \text{ in } B \parallel M \qquad \text{(BM2)}$$

• Proof of (BM1) $r \neq \phi \ \wedge \ w = \phi \ \ \textbf{co} \ \ r \neq \phi \ \equiv \ w = \phi \ \text{ in } B \parallel M$:
In B

$$\begin{array}{ll} & r \neq \phi \ \wedge \ w = \phi \\ \textbf{co} & \{\text{elimination theorem on B1}\} \\ & \langle \exists\, X, Y :: (X \neq \phi \ \wedge \ Y = \phi \ \wedge \ r, w = X, Y) \\ & \vee \ (X \neq \phi \ \wedge \ r, w, Y = \phi, X, \phi)\rangle \\ \Rightarrow & \{\text{predicate calculus}\} \\ & (r \neq \phi \ \wedge \ w = \phi) \ \vee \ (r = \phi \ \wedge \ w \neq \phi) \ \text{ in } B \\ \equiv & \{\text{predicate calculus}\} \\ & r \neq \phi \ \equiv \ w = \phi \end{array}$$

Thus,

$$\begin{array}{ll} r \neq \phi \ \wedge \ w = \phi \ \ \textbf{co} \ \ r \neq \phi \ \equiv \ w = \phi \ \text{ in } B & \\ & \text{, from above} \\ \textbf{stable } r \neq \phi \ \wedge \ w = \phi \ \text{ in } M & \text{, conjunction of M1, M2} \\ r \neq \phi \ \wedge \ w = \phi \ \ \textbf{co} \ \ r \neq \phi \ \equiv \ w = \phi \ \text{ in } B \parallel M & \\ & \text{, union theorem corollary} \quad \square \end{array}$$

• Proof of (BM2) $\textbf{transient } r \neq \phi \ \wedge \ w = \phi \ \text{ in } B \parallel M$:
Apply union theorem using (B2). $\square$

Now we prove the desired progress properties using (BM1, BM2).

- Proof of $w = \phi \ \mapsto \ r = \phi$ in $B \parallel M$:
  All properties in the following proof are in $B \parallel M$.
  $r \neq \phi \ \wedge \ w = \phi$ **co** $r = \phi \ \vee \ w = \phi$, weaken rhs of BM1
  transient $r \neq \phi \ \wedge \ w = \phi$, BM2
  $w = \phi$ **en** $r = \phi$, definition of **en**
  $w = \phi \ \mapsto \ r = \phi$, definition of $\mapsto$ □

- Proof of $r \neq \phi \ \mapsto \ w \neq \phi$ in $B \parallel M$:
  All properties in the following proof are in $B \parallel M$.
  $r \neq \phi \ \wedge \ w = \phi$ **co** $r \neq \phi \ \vee \ w \neq \phi$, weaken rhs of BM1
  transient $r \neq \phi \ \wedge \ w = \phi$, BM2
  $r \neq \phi$ **en** $w \neq \phi$, definition of **en**
  $r \neq \phi \ \mapsto \ w \neq \phi$, definition of $\mapsto$ □

10.
$F \parallel H$
$\Rightarrow$ $\{F \equiv\!\!> G$. Apply the theorem in section 8.5.2$\}$
$G \parallel H$
$\Rightarrow$ $\{H \equiv\!\!> J$. Apply the theorem in section 8.5.2$\}$
$G \parallel J$

11. Preservation of **co** properties are easy to establish. For progress properties show that if

    r **en** s in $F \parallel \alpha$, then
    $r \ \mapsto \ s$ in $F \parallel \beta$.

    Then apply the result of exercise 11 of chapter 6 to conclude that all *leads-to* properties are preserved. See Misra [134, note 19] for details.

9
Conditional and Closure Properties

9.1 Introduction

The union theorem, introduced in section 8.2.3, is the main tool for the study of asynchronous compositions of programs. The major virtue of this theorem is that it provides a simple rule for deducing the **co**-properties and transient predicates of a system from those of its component boxes. The major shortcoming is that it does not provide a simple rule for deducing the *leads-to* properties of a system from those of its components. The only way we can use a progress property of a component, $p \mapsto q$ in F, to deduce a similar property of the system is to either (1) apply the union theorem for progress (see section 8.2.6), which requires introduction of a well-founded order over the values of the shared variables, or (2) exhibit the proof of $p \mapsto q$ in F and show that the other components do not falsify the proof steps. The first strategy is cumbersome, and the second defeats the whole purpose of modular program construction.

The difficulty we have encountered is not merely due to the formalism we have adopted. The problem of designing components of a system so that only limited knowledge of the other components is required during the design process is fundamental and fundamentally difficult. However, in practice, application programs are routinely developed for execution under operating systems given only the specification of the latter in the form: "messages are eventually delivered", or "each resource request is eventually met". We propose a theory in this chapter that may explain how modular

design of software is possible in practice. The theory builds on the logical operators introduced in the previous chapters.

Until now, we have specified a program F by a set of properties that hold when F is executing alone; call such properties *unconditional*. The union theorem (section 8.2.3) provides some help in combining the unconditional properties of the components to deduce system properties. Limitations of the union theorem illustrate that unconditional properties are too weak for specifications when the goal is to construct a system from components.

Overview of the chapter

In the next section, we introduce *conditional* properties; conditional property of a component F states what is assumed in F about its environment G and what may then be guaranteed about the system, $F \parallel G$. If box H meets the assumptions of F's environment, properties of $F \parallel H$ can be derived from the guarantee part of the conditional property. Progress properties of a system are derived quite easily in this fashion from component properties. Several examples are given to illustrate the methodology.

In section 9.3, we introduce an even stronger class of properties, called *closure*. We postulate that each component F is typically designed for composition with specific environments; therefore, we restrict $F \parallel G$ to be defined only when G satisfies the properties for F's environments, and F for G's. The properties of the environments are encoded into the text of a component, and they can be checked by a linker during composition with another component. The specification of F is given by its properties not when it is executing alone, but when it is executing with its proposed environment. Closure properties are generalizations of the usual **co**, **en**, **transient**, $\mapsto$, etc. Several examples illustrate that use of such properties can reduce proof lengths significantly.

9.2 Conditional Properties

9.2.1 Specification using conditional properties

The kinds of properties used so far —**co**, **en**, **transient**, $\mapsto$, etc.— are called *unconditional*. Given that Π and Π' are unconditional properties,

$$\langle \forall\, G :\ \Pi' \ \text{ in } G :\ \Pi \ \text{ in } F \parallel G \rangle$$

is called a *conditional* property of program F. Property Π' is what F assumes about its environment G, and Π is what it guarantees about the system, $F \parallel G$, provided that the assumption is met. Any program G that satisfies Π' is a possible environment of F. Such a conditional property can be proved from the code/specification of F by assuming that Π' in G holds for an arbitrary G, and proving Π in $F \parallel G$. The main tool in such

a proof is the union theorem. We have seen such proofs in section 8.3.2 for the handshake protocol and in section 8.3.3 for a semaphore program.

Example (handshake protocol) Consider the handshake protocol of section 8.3.2. The text of a program *send* was shown in that section, and its environment F was postulated to have the following properties. For arbitrary natural numbers m and n,

- (F1) $nr, ns = m, n$ **co** $ns = n \ \wedge \ (nr = m \ \vee \ m \leq nr \leq n)$ in F
- (F2) $nr < ns \ \mapsto \ nr = ns$ in F

Then the following properties of $F \parallel send$ were deduced.

- (FG1) **invariant** $nr \leq ns \leq nr + 1$ in $F \parallel send$
- (FG2) $true \ \mapsto \ ns > n$ in $F \parallel send$

Therefore, a conditional property of *send* is

$\langle \forall F :$
$\quad nr, ns = m, n$ **co** $ns = n \ \wedge \ (nr = m \ \vee \ m \leq nr \leq n)$ in F
$\quad nr < ns \ \mapsto \ nr = ns$ in F
$\quad :$
$\quad$ **invariant** $nr \leq ns \leq nr + 1$ in $F \parallel send$,
$\quad true \ \mapsto \ ns > n$ in $F \parallel send$
$\rangle$ □

Conditional properties of multiple components can be combined to deduce a system property as follows. Let f, g, and h be programs. Suppose

$\langle \forall \ G : \ \Pi'$ in $G : \ \Pi$ in $f \parallel G \rangle$ (1)
$\langle \forall \ H : \ \Pi''$ in $H : \ \Pi'$ in $g \parallel H \rangle$ (2)
Π'' in h (3)

Then

Π' in $g \parallel h$, combining the properties (2,3) using h for H
Π in $f \parallel g \parallel h$, combining above and (1) using $g \parallel h$ for G

We propose a general rule in the next section to simplify such deductions.

9.2.2 Linear network

Properties of a regular network —a systolic array [31, 110], for instance— can often be established easily if each component of the network is specified by a suitable conditional property. The following rule applies to a network consisting of programs f_i, $0 \leq i \leq N$. Program f_{i+1} expects the property Π_i of its environment and it guarantees Π_{i+1} when composed with such an environment.

Linear network rule

$$\frac{\begin{array}{l}\Pi_0 \ \text{ in } f_0 \text{ and} \\ \langle \forall\ i:\ 0 \le i < N: \\ \qquad \langle \forall\ g:\ \Pi_i \ \text{ in } g:\ \Pi_{i+1} \ \text{ in } f_{i+1} \ [\!]\ g \rangle \\ \rangle\end{array}}{\Pi_N \ \text{ in } \langle [\!]\ i:\ 0 \le i \le N:\ f_i \rangle}$$

Proof is by induction on N. For $\langle [\!]\ i:\ 0 \le i \le j:\ f_i \rangle$ use the abbreviation F_j. The conclusion is, then, $\Pi_N \ \text{ in } F_N$.

Case $N = 0$**:** $\Pi_0 \ \text{ in } F_0$	, premise, $F_0 = f_0$
Case $N + 1,\ N \ge 0$**:**	
$\langle \forall\ g:\ \Pi_N \ \text{ in } g:\ \Pi_{N+1} \ \text{ in } f_{N+1} \ [\!]\ g \rangle$	, premise with i set to N
$\Pi_N \ \text{ in } F_N$	, induction hypothesis
$\Pi_{N+1} \ \text{ in } f_{N+1} \ [\!]\ F_N$	, above two: use F_N for g
$\Pi_{N+1} \ \text{ in } F_{N+1}$	, $F_{N+1} = f_{N+1} \ [\!]\ F_N$

The rule given here applies to any network configuration. There is no constraint on the direction of "data flow"; any program f_i may read/write variables changed by any other program. We call the network "linear" because it is possible to order the components based on an ordering of the properties to be established. We consider an example in section 9.2.4 where program f_i communicates with both f_{i-1} and f_{i+1}, $0 < i < N$, yet the apparent circularity is resolved by judicious use of conditional properties.

9.2.3 Example: producer, consumer

We treat a producer–consumer problem in this section, deriving a progress property of the system solely from the specifications of its components. A producer writes successive values from a sequence *in* into a shared variable x; the consumer removes the value from x and appends it to a sequence *out*. A protocol for accessing x is imposed on both components so that no data is overwritten before it is read, nor is any data read twice. Specification of each component includes conditional properties: assuming that its environment obeys the protocol for accessing x, it guarantees certain properties of the system. We show how the conditional properties facilitate derivation of the following progress property of the system: *out* eventually equals any finite prefix of *in*.

Box *con* (consumer) has two variables: global x and local *out*. The job of *con* is to take the value of x and append it to sequence *out*; variable x has a special value ϕ to denote that x holds no useful value; i.e., its most recent value has been appended to *out*. It is expected that some other program writes into x. However, it is crucial that the other program not change x

as long as $x \neq \phi$; otherwise, a value in x will be overwritten without being copied into *out*.

In the following, m and n are of the same type as values written into x (including ϕ); B is a free variable of type sequence. Let $B +\!\!+ m$ denote the sequence obtained by appending m to B; note that $B +\!\!+ \phi = B$. Property (c1) specifies how x and *out* are modified. Property (c2) assures us that every item from x is appended to *out* provided that the environment of *con* obeys the desired protocol for writing into x.

Specification of con

$$x, out = m, B \;\; \textbf{co} \;\; x, out = m, B \;\vee\; x, out = \phi, B +\!\!+ m \;\; \text{in } con \tag{c1}$$

$$\begin{array}{l} \langle \forall\, G :\; \langle \forall\, n :\; n \neq \phi :\; \textbf{stable}\; x = n \;\; \text{in } G \rangle : \\ \qquad x, out = m, B \;\mapsto\; x,\; out = \phi,\; B +\!\!+ m \;\; \text{in } con \parallel G \\ \rangle \end{array} \tag{c2}$$

Next, consider program *prd* (for producer) that writes successive values of its local sequence *in* into x. This program obeys the expected protocol of writing into x: it stores the next item into x only if $x = \phi$ (see property p1). The program has a local variable p that is the number of items from *in* that have been written into x. The conditional property (p2) assumes that the environment of *prd* does not write any non-ϕ value into x, i.e., *prd* is the only producer for x. In the following, $in[i]$ is item i of *in*, $i \geq 0$. Initially, $p = 0$ and $x = \phi$.

Specification of prd

$$p, x = i, m \;\; \textbf{co} \;\; p, x = i, m \;\vee\; (m = \phi \;\wedge\; p, x = i + 1, in[i]) \;\; \text{in } prd \tag{p1}$$

$$\begin{array}{l} \langle \forall\, F :\; \textbf{stable}\; x = \phi \;\; \text{in } F :\; p, x = i, \phi \;\mapsto\; p, x = i + 1, in[i] \\ \qquad\qquad \text{in } prd \parallel F \\ \rangle \end{array} \tag{p2}$$

Specification of con $\parallel$ *prd*

We show that eventually *out* equals any finite prefix of *in*. That is, for any natural number j,

$$p, x, out = 0, \phi, \langle\,\rangle \;\mapsto\; out = in^j \;\; \text{in } con \parallel prd \tag{cp1}$$

where $\langle\,\rangle$ denotes the empty sequence and in^j is the sequence of items $in[i]$, $0 \leq i < j$, in increasing order of i.

As would be expected, the proof of (cp1) has to establish that (1) the next item from *in* is stored in x if $x = \phi$, from the specification of *prd*, and (2) the value in x is appended to *out* and x set to ϕ, from the specification of *con*. Below we deduce several properties (c3, p3, cp2, cp3) merely as exercises in derivation; they are not required for the proof of (cp1). The

progress properties of $con \parallel prd$ are established by instantiating F by con in (p2) (see cp2, below) and G by prd (see cp3).

• Proof of (c3), **stable** $x = \phi$ in con:

$$\begin{array}{ll} \textbf{stable}\ x, out = \phi, B \ \text{ in } con & \text{, let } m = \phi \text{ in (c1)} \\ \textbf{stable}\ x = \phi \ \text{ in } con & \text{, disjunction over all } B \end{array}$$ □

• Proof of (cp2), $p, x = i, \phi \ \mapsto \ p, x = i + 1, in[i]$ in $con \parallel prd$:
From (p2 and c3) using con for F. □

• Proof of (p3), $\langle \forall\ n :\ n \neq \phi :\ \textbf{stable}\ x = n \ \text{ in } prd \rangle$:

$$\begin{array}{ll} \langle \forall\ i, n :\ n \neq \phi :\ \textbf{stable}\ p, x = i, n \ \text{ in } prd \rangle & \\ & \text{, setting } m \text{ to } n \text{ and } n \neq \phi \text{ in (p1)} \\ \langle \forall\ n :\ n \neq \phi :\ \textbf{stable}\ x = n \ \text{ in } prd \rangle & \\ & \text{, disjunction over } i \end{array}$$ □

• Proof of (cp3), $x, out = m, B \ \mapsto \ x, out = \phi, B +\!\!+ m$ in $con \parallel prd$:
From (c2 and p3) using prd for G. □

• Proof of (cp4), $p, x, out = i, m, B$ **co**
$p, x, out = i, m, B \ \vee \ (m = \phi \ \wedge p, x, out = i + 1, in[i], B) \ \vee$
$p, x, out = i, \phi, B +\!\!+ m$ in $con \parallel prd$:

From the union theorem, the above **co**-property has to be proved in each component, con and prd. We leave the proof as an exercise; note that p is constant in con and out in prd. □

• Proof of (cp5),
$p, x, out = i, \phi, B \ \mapsto \ p, x, out = i + 1, in[i], B$ in $con \parallel prd$:

In $con \parallel prd$ ::
$p, x, out = i, \phi, B$ **co** $p, x, out = i, \phi, B \ \vee \ p, x, out = i + 1, in[i], B$
, set m to ϕ in (cp4)

The result follows by applying PSP to above and (cp2). □

• Proof of (cp6), For $m \neq \phi$,
$p, x, out = i, m, B \ \mapsto \ p, x, out = i, \phi, B +\!\!+ m$ in $con \parallel prd$:

$p, x, out = i, m, B$ **co** $p, x, out = i, m, B \ \vee \ p, x, out = i, \phi, B +\!\!+ m$
, from (cp4) using $m \neq \phi$

The result follows by applying PSP to above and (cp3). □

We are now ready to prove (cp1). We show that out will eventually equal the prefix of in up to position j, for any natural number j,

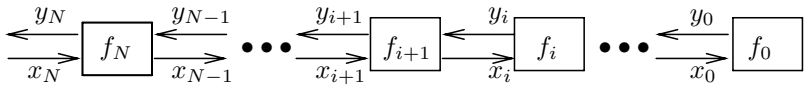


Figure 9.1: The structure of a network to compute factorials

- Proof of (cp1), $p, x, out = 0, \phi, \langle\,\rangle \;\mapsto\; out = in^j$ in $con \,[\!]\, prd$:
  The following proof is over $con \,[\!]\, prd$; i is any natural number:

  $p, x, out = i, \phi, in^i \;\mapsto\; p, x, out = i+1, in[i], in^i$
  , set B to in^i in (cp5)
  $p, x, out = i+1, in[i], in^i \;\mapsto\; p, x, out = i+1, \phi, in^{i+1}$
  , set i, m, B to $i+1, in[i], in^i$ in (cp6)
  $p, x, out = i, \phi, in^i \;\mapsto\; p, x, out = i+1, \phi, in^{i+1}$
  , transitivity on above two
  $p, x, out = 0, \phi, \langle\,\rangle \;\mapsto\; p, x, out = j, \phi, in^j$
  , induction over i □

9.2.4 Example: factorial network

This example, from Misra and Chandy [142], illustrates the application of the linear network rule of section 9.2.2. The proof given here is significantly shorter than the original.

A network to compute factorials of a sequence of natural numbers is shown in Fig. 9.1. Here, each program f_i receives a sequence of natural numbers in global variable x_i. The program produces the sequence of their factorials (maintaining the proper order) in y_i. The computation strategy is as follows. If f_{i+1}, $0 \leq i < N$, receives a positive number n, it sends $n-1$ to f_i, by appending it to x_i. Assuming that f_i delivers $(n-1)!$ eventually in y_i, f_{i+1} then computes $n! = n \times (n-1)!$ and appends $n!$ to y_{i+1}. To maintain the proper order of outputs, f_{i+1} uses the following strategy: it stores the sequence of all received numbers whose factorials are yet to be computed; if the head of the sequence is 0, a 1 is appended to y_{i+1} (and the 0 discarded); if the head of the sequence is nonzero, it is multiplied with the next received number from y_i and the result is appended to y_{i+1} (and the head item of the stored sequence is discarded). The last program, f_0, computes the factorial of each received number and appends it to y_0. If the numbers sent to f_N are at most N, the numbers sent to f_0 are zeroes, so their factorials are easily computed.

Data flow in the network is in both directions, from f_{i+1} to f_i and back. Yet the network can be regarded as linear because the correctness of f_i depends only on the correctness of f_j, $j < i$, as shown below.

Notation For a sequence of natural numbers u, let $u!$ be the sequence of their factorials. Let $u \sqsubseteq v$ denote that u is a prefix of v. □

Consider the following specification in which each predicate is an invariant of the appropriate program. Variable x_i is the sequence of all numbers sent along the corresponding channel; similarly y_i. The conditional property, below, states that f_{i+1} works appropriately: if it receives a prefix of $x_i!$ in y_i, it produces a prefix of $x_{i+1}!$ in y_{i+1}.

$$
\begin{array}{l}
y_0 \ \sqsubseteq \ x_0! \ \text{ in } f_0 \\
\langle \forall\ i:\ 0 \le i < N: \\
\quad \langle \forall\ G:\ y_i \ \sqsubseteq \ x_i! \ \text{ in } G:\ y_{i+1} \ \sqsubseteq \ x_{i+1}! \ \text{ in } f_{i+1} \ [\!] \ G \rangle \\
\rangle
\end{array}
$$

Then applying the linear network rule we get

$$y_N \ \sqsubseteq \ x_N! \ \text{ in } \langle [\!]\ i:\ 0 \le i \le N:\ f_i \rangle$$

Next, assume the following progress properties: (1) f_0 responds to each input along x_0 by sending an output along y_0 and (2) if eventually there is an output in y_i for each input in x_i, $0 \le i < N$, eventually there is an output in y_{i+1} for each input in x_{i+1}. Here, m and n are arbitrary natural numbers, and $|x_i|$ is the length of sequence x_i.

$$
\begin{array}{l}
|x_0| = m \ \mapsto \ |y_0| = m \ \text{ in } f_0 \\
\langle \forall\ i:\ 0 \le i < N: \\
\quad \langle \forall\ G: |x_i| = m \ \mapsto \ |y_i| = m \ \text{ in } G: \\
\qquad\qquad |x_{i+1}| = n \ \mapsto \ |y_{i+1}| = n \ \text{ in } f_{i+1} \ [\!] \ G \rangle \\
\rangle
\end{array}
$$

By applying the linear network rule, we derive for any m, $m \ge 0$,

$$|x_N| = m \ \mapsto \ |y_N| = m \ \text{ in } \langle [\!]\ i:\ 0 \le i \le N:\ f_i \rangle$$

9.2.5 Example: concurrent bag

An object, as in object-oriented programming or Seuss, may be regarded as a global variable that is shared among a number of programs. A queue, for instance, is shared among producers and consumers; a producer adds to the queue and a consumer removes from it. The accesses to the object are sequential if each operation on the object terminates before the next one can begin. For the queue problem, a consumer, for instance, has to receive a response to each of its requests: either an item from the queue or an indication that the queue is empty. It is not possible to defer the request until the queue becomes nonempty.

A major advantage of sequential access is that a data object can be specified succinctly by writing a set of equations that relate the effects of the various operations on this object, see Guttag [81]. For instance, let $\langle\rangle$ denote an empty queue, $add(q, x)$ is the queue obtained by appending item x to q and $remove(q)$ is the queue obtained by removing the first item of q if q is nonempty. Then we have

$remove(add(\langle\rangle, x)) = \langle\rangle$ and,
$remove(add(q, x)) = add(remove(q), x)$, for $q \neq \langle\rangle$

The implicit assumption in such a specification is that each operation represents a total function. Therefore, the operations have to be deterministic; the unordered channel (section 4.1.3) where the message delivery order is arbitrary cannot be expressed in this formalism. Further, each operation has to terminate; the fact that a producer has to wait for the queue to become nonfull (so it may wait forever) is difficult to state. Finally, this formalism assumes that the operations are performed in a strict order, i.e., the durations of the two operations, *add* and *remove*, may not overlap. This eliminates the possibility of the operations being concurrent.

Now consider the shared variable x in the producer-consumer example of section 9.2.3. This is a one-place queue (also called a *word* in section 4.1.2). A producer can add only if $x = \phi$ and consumer can remove only if $x \neq \phi$. Therefore, it is possible for either program to wait forever. Additionally, for a queue that can hold more than one item, it is possible for *add* and *remove* to be executed concurrently (i.e., an operation may start while the other one is deferred).

The purpose of this section is to specify a bounded *bag* on which the operations are nondeterministic, possibly concurrent, and potentially nonterminating. We view this object as a program. Its specification includes conditional properties that specify the protocol for access and also its guarantees about the system. The major portion of this example is devoted to proving that finite concatenations of bags is a bag.

Specification of concurrent bag

A concurrent bag is shared among a group of producers and consumers. The schematic of the interaction between the programs is given in Fig. 9.2. Program B represents the bag. The producers and consumers, collectively, constitute program F, which is the environment of B. Producers add items to the bag by successively storing them in r; consumers remove successive items from w. There may be several producers and consumers, or even a single program that is both the producer and the consumer; the exact number is irrelevant for the specification. We have seen such an interaction topology for client servers in section 8.3.5; here B is analogous to the program $BUFF$.

We postulate a special value ϕ for a variable to denote that it contains no useful data. The protocol for reading and writing is as follows. Program F writes into r only if $r = \phi$; program B reads from r only if $r \neq \phi$ and sets r to ϕ after reading the value. Program B stores a value in w only if $w = \phi$; program F reads from w and then sets w to ϕ, signaling that it is ready to consume the next data item. Thus, the accesses of F and B are symmetric with respect to r and w.

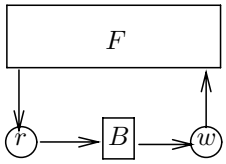


Figure 9.2: B is a concurrent bag that reads from r and writes into w. The environment F of B writes into r and reads from w.

Convention All bags are finite. We treat r and w also as bags that contain at most one item; ϕ is an empty bag. Let $u + v$ denote the union u and v. Write $|u|$ for the size of u. Note that $|u + v| = |u| + |v|$. □

Introduce an order relation $\prec$ over all data values and ϕ, as follows:

$$x \prec y \quad \equiv \quad x = \phi \ \wedge \ y \neq \phi$$

The following specification is of a bag B that can internally hold at most N items, $N \geq 0$. (A bag of size 0 merely moves data from r to w.) Let $\top$ denote the types of the elements of bag; free variables R and W are of type $\top$.

Specification B
  local bag($\top$**)** b;
  global $(\top \ \cup \ \{\phi\})$ r, w;
  initially $b + w = \phi$ in B $(B0)$
  stable $r \preceq R$ in B $(B1)$
  stable $W \preceq w$ in B $(B2)$
  invariant $|b| \leq N$ in B $(B3)$
  constant $r + b + w$ in B $(B4)$
  $\langle \forall \ F :$ **stable** $R \preceq r$ in F, **stable** $w \preceq W$ in F:
    $|b + w| \leq N \ \mapsto \ r = \phi$ in $B \ [\!] \ F$, $(B5)$
    $|b + r| > 0 \ \mapsto \ w \neq \phi$ in $B \ [\!] \ F$ $(B6)$
  $\rangle$
end $\{B\}$

Variable b is the bag of data items internally stored by B. Property $(B0)$ says that both b and w are initially empty. Property $(B1)$ states that B can set r only to ϕ; similarly, $(B2)$ says that B can write only non-ϕ values into w provided that $w = \phi$. Observe that $(B2)$ prevents B from overwriting a non-ϕ value in w by either ϕ or another non-ϕ value. Property $(B3)$ specifies the size constraint on b. $(B4)$ is a conservation law: program B can neither create nor destroy any data in $r + b + w$. This allows B to freely move data between r, b, and w; in particular, the order in which items are received in r may be different from the order in which they are sent via w.

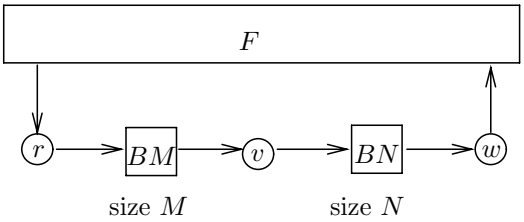


Figure 9.3: Concatenation of bags BM, BN of sizes M, N

Note that $(B4)$ is symmetric in r, b, and w, allowing data movement from w to b or r, for instance. However, $(B1, B2)$ constrain the directions of data movement. The conditional progress properties $(B5, B6)$ assume that the environment obeys the access protocols for r and w; property $(B5)$ states that if there is room in b or if w is empty, the data in r, if any, will be eventually removed; property $(B6)$ is the counterpart $(B5)$ for writing into w.

From the locality of b, environment F has the property

$$\textbf{stable } |b| \leq N \text{ in } F$$

so using $(B3)$ and the union theorem

$$\langle \forall\ F ::\ \textbf{invariant } |b| \leq N \text{ in } B \parallel F \rangle \qquad (B7)$$

Bag concatenation

We show that concatenation of two bags of sizes M and N and variable v, as shown in Fig. 9.3, implements a bag of size $M + N + 1$. This result can be used to implement a bag of any size k, $k > 0$, by concatenating $(k + 1)$ bags of size 0.

Let BM and BN be programs that implement bags of size M and N, respectively. These two programs can be "concatenated" provided that the output global variable of one is the input of the other. Fig. 9.3 shows the concatenation where r and v are the input/output variables of BM and v and w are the corresponding variables of BN. Therefore, variable v is local to $BM \parallel BN$. Henceforth, we write BMN as an abbreviation for $BM \parallel BN$. Our proof obligation is that BMN satisfies the specification B, with N replaced by $M + N + 1$.

The proof strategy is as follows. First, construct the specifications of BM and BN. These are obtained from specification B by substituting the appropriate input, output variables and size constraints: for BM we substitute bm, BM, v, V, and M for b, B, w, W, and N, respectively. For BN we substitute bn, BN, v, and V for b, B, r, and R, respectively. Next, we combine these properties, using the union theorem, to establish the properties of BMN.

Specification BM
local bag($\top$) bm;
global ($\top \cup \{\phi\}$) r, v;
initially $bm + v = \phi$ in BM ($BM0$)
stable $r \preceq R$ in BM ($BM1$)
stable $V \preceq v$ in BM ($BM2$)
invariant $|bm| \leq M$ in BM ($BM3$)
constant $r + bm + v$ in BM ($BM4$)
$\langle \forall\, G :$ **stable** $R \preceq r$ in G, **stable** $v \preceq V$ in G:
$|bm + v| \leq M \;\mapsto\; r = \phi$ in $BM \,[\!]\, G$ ($BM5$)
$|bm + r| > 0 \;\mapsto\; v \neq \phi$ in $BM \,[\!]\, G$ ($BM6$)
$\rangle$
end $\{BM\}$

Specification BN
local bag($\top$) bn;
global ($\top \cup \{\phi\}$) v, w;
initially $bn + w = \phi$ in BN ($BN0$)
stable $v \preceq V$ in BN ($BN1$)
stable $W \preceq w$ in BN ($BN2$)
invariant $|bn| \leq N$ in BN ($BN3$)
constant $v + bn + w$ in BN ($BN4$)
$\langle \forall\, H :$ **stable** $V \preceq v$ in H, **stable** $w \preceq W$ in H:
$|bn + w| \leq N \;\mapsto\; v = \phi$ in $BN \,[\!]\, H$ ($BN5$)
$|bn + v| > 0 \;\mapsto\; w \neq \phi$ in $BN \,[\!]\, H$ ($BN6$)
$\rangle$
end $\{BN\}$

Similar to ($B7$), we have the following invariants.

$\langle \forall\, G ::$ **invariant** $|bm| \leq M$ in $BM \,[\!]\, G\rangle$ ($BM7$)
$\langle \forall\, H ::$ **invariant** $|bn| \leq N$ in $BN \,[\!]\, H\rangle$ ($BN7$)

We have to show that BMN satisfies specification B, with N replaced by $M + N + 1$ (in $B3$ and $B6$). The local variable b of BMN is defined by

$$b \;=\; bm + v + bn \qquad \text{(D)}$$

• Proof of ($B0$) **initially** $b + w = \phi$ in BMN:
initially $bm + v = \phi$ in BM , from $BM0$
initially $bn + w = \phi$ in BN , from $BN0$
initially $bm + v + bn + w = \phi$ in BMN
, union theorem on above two
initially $b + w = \phi$ in BMN , definition of b from (D) □

• Proof of $(B1)$ **stable** $r \preceq R$ in BMN:

constant r in BN	, locality
stable $r \preceq R$ in BN	, from above
stable $r \preceq R$ in BM	, $BM1$
stable $r \preceq R$ in BMN	, union theorem (above two) □

• Proof of $(B2)$ **stable** $W \preceq w$ in BMN: similar to the proof of $B1$. □

• Proof of $(B3)$ **invariant** $|b| \leq M + N + 1$ in BMN:

invariant $\lvert bm\rvert \leq M$ in $BM \parallel BN$,	from $BM7$ with BN for G
invariant $\lvert bn\rvert \leq N$ in $BM \parallel BN$,	from $BN7$ with BM for H
invariant $\lvert b\rvert \leq M + N + 1$ in B	, using (D) □

• Proof of $(B4)$ **constant** $r + b + w$ in BMN:

constant $r + bm + v$ in BM	, $BM4$
constant $bn + w$ in BM	, locality
constant $r + b + w$ in BM	, from above two using (D)
constant $r + b + w$ in BN	, similarly
constant $r + b + w$ in BMN	, union theorem on above two □

• Proof of $(B5)$ $\langle \forall\, F :$ **stable** $R \preceq r$ in F, **stable** $w \preceq W$ in F:
$|b + w| \leq M + N + 1 \;\mapsto\; r = \phi$ in $BMN \parallel F \rangle$:

Consider any F that satisfies the assumptions of $(B5)$.

stable $R \preceq r$ in F	, assumption
stable $R \preceq r$ in BN	, locality
stable $R \preceq r$ in $BN \parallel F$	, union theorem
stable $v \preceq V$ in $BN \parallel F$	, $(BN1)$ and the locality of v
stable $V \preceq v$ in $BM \parallel F$	, similarly
stable $w \preceq W$ in $BM \parallel F$	, similarly

Thus,

$BN \parallel F$ meets the assumptions for G in $(BM5, BM6)$
$BM \parallel F$ meets the assumptions for H in $(BN5, BN6)$

In the following proof all properties are in $BM \parallel BN \parallel F$.

$|b + w| \leq M + N + 1$
$\Rightarrow$ {from (D), $b = bm + v + bn$}
$|bm + v + bn + w| \leq M + N + 1$
$\Rightarrow$ {arithmetic}
$|bm + v| \leq M \;\vee\; |bn + w| \leq N$
$\mapsto$ {from $BN5$, using $BM \parallel F$ for H, $|bn + w| \leq N \;\mapsto\; v = \phi$}
$|bm + v| \leq M \;\vee\; v = \phi$
$\Rightarrow$ {$v = \phi \;\Rightarrow\;$ {from $BM7$} $(|bm + v| = |bM| \leq M)$}

$|bm + v| \leq M$
$\mapsto$ {from $BM5$, using $BN \parallel F$ for G}
$r = \phi$ □

• Proof of $(B6)$: similar to that of $(B5)$. □

Data refinement

Here, we propose a refinement of Specification B. We prove the correctness of the refinement by showing that the original specification can be derived from the refined specification. We restrict attention to the case where the bag size N is zero. (As we have noted, any finite bag can be implemented by concatenating several zero-size bags.) The refined specification, given below, permits movement of data from r to w provided that $w = \phi$; further, $r \neq \phi \ \wedge \ w = \phi$ cannot persist forever. (Such a specification for a buffer is given in the solution to exercise 9 of chapter 8.) In this specification, X and Y are free variables of the same type as r and w.

Specification *Ref*
global $(\top \ \cup \ \{\phi\})\ r, w$;
initially $w = \phi$ in B $(R0)$
$r, w = X, Y$ **co** $r, w = X, Y \ \vee \ (Y = \phi \ \wedge \ r, w = \phi, X)$ in B $(R1)$
transient $r \neq \phi \ \wedge \ w = \phi$ in B $(R2)$
end {*Ref*}

We have to prove $(B0$–$B6)$ given $N = 0$ (i.e., $b = \phi$) and $(R0$–$R2)$. Proof of $(B0)$ from $(R0)$ and $b = \phi$ is trivial. Proofs of $(B1, B2, B4)$ follow from $(R1)$ by employing the elimination theorem. We show one proof.

• Proof of $(B4)$ **constant** $r + w$: {note: $b = \phi$}
For any z, a bag of type $\top$,

$r + w = z$
co {elimination theorem on $R1$}
$\langle \exists\ X, Y :\ X + Y = z :\ r, w = X, Y \ \vee \ (Y = \phi \ \wedge \ r, w = \phi, X) \rangle$
$\Rightarrow$ {predicate calculus}
$\langle \exists\ X, Y ::\ X + Y = z \ \wedge \ r, w = X, Y \rangle$
$\vee \ \langle \exists\ X, Y ::\ X + Y = z \ \wedge \ Y = \phi \ \wedge \ r, w = \phi, X \rangle$
$\Rightarrow$ {predicate calculus}
$r + w = z$

Therefore, **constant** $r + w$. □

Proofs of $(B5,\ B6)$ appear as solutions to exercise 9 of chapter 8.

Discussion

Equational notation, as in Guttag [81], is attractive for specifying data objects because it separates the concern of implementation from deductions of properties of the object. Concurrently accessed objects do not seem amenable to equational specification. We specify such objects using traditional safety and progress properties. The specification of a bag, ($B0$–$B6$), is concise, and it supports concurrent access. We have used this specification to deduce properties of the object and carry out refinements.

The specification of a bag includes the declarations of the global variables —r and w— through which this program communicates with its environment. If multiple producers/consumers interact with the bag, they have to interleave their accesses to r and w. As we showed in section 8.3.5, a client manager, CM, and a service manager, SM, can be introduced to merge the various requests from producers into r and distribute the output from w to the consumers. Programs CM and SM can be specified independently.

9.3 Closure Properties

Conditional properties are rich in expressive power. The examples given in sections 9.2.3, 9.2.4 and 9.2.5 show the effectiveness of conditional properties in specifying programs that are composed with environments of known characteristics. However, manipulations of conditional properties are more difficult than their unconditional counterparts; for instance, we had to instantiate program G by $BN \parallel F$ and H by $BM \parallel F$ in the proof of (B5) for bag concatenation (see section 9.2.5), and we had to reason about the properties of an ensemble of programs. Specifications and proofs can be considerably simplified if we can encode the assumptions about an environment in a program's text (or specification). Then, we can deduce properties of *any* system of which this program is a component from the text (or specification) of the program alone.

It is possible to include arbitrary safety and progress properties of the environment as part of a program specification. However, we impose a constraint: given programs F and G, it should be decidable if F is a possible environment of G and G of F, i.e., if $F \parallel G$ is defined. This constraint permits an automatic linker to form the union of component programs (or reject them when a union cannot be formed because the appropriate assumptions are not met). Moreover, such a constraint simplifies reasoning about compositions of programs, as we show in this section.

We constrain the environment of a program by attaching appropriate type information to the global variables. Specifically, we declare a global variable in a program with an *internal type* and an *external type*; the internal type describes the permissible values and operations on this variable inside the given program, whereas the external type describes this informa-

tion for the environment. It is then possible to deduce properties of $F \parallel G$ from the text of F alone, where G is *any* program that satisfies the external type constraints specified for the global variables in F; i.e., G is a possible environment of F.

Traditionally, type declarations have simplified specification and verification by delegating their decidable aspects to a compiler. We find that type checking across programs (by a linker) can play an equally crucial role in the study of program composition.

9.3.1 Types of global variables

So far in the treatment of program composition, we have associated a single type with every variable including every global variable. This form of type declaration may be interpreted as follows: any operation on a global variable that is permissible in one box of a program is also permissible in any other box (where this variable is declared). Thus, the boxes are *symmetric* in their accesses to global variables. (The only possible source of asymmetry is that a global variable is accessible in one box but not in another, because it is declared only in the former box.)

Usually, boxes in a system access and manipulate global variables asymmetrically. For instance, senders and receivers that communicate over unidirectional channels have asymmetric accesses to the channels. Similarly, for the bag example of section 9.2.5, variables r and w are accessed asymmetrically by B and F. As an extreme case, a local variable of box F may be regarded as a global variable that can be accessed by F alone.

Motivated by these observations, we propose to declare a global variable in a program with a base type (integer, for instance) and internal and external types that specify the access rights, i.e., which operations are permissible internally and externally on the variable. For instance, a global integer variable x may have permissible operations $\{read, write\}$ inside F and operation $\{read\}$ outside F. It follows that x can be written only in F; i.e., x is constant in every program G other than F.

Types and subtypes

We refrain from constructing an elaborate theory of types; we merely require that the types form a lattice under a subtype ($\subseteq$) relation. Henceforth, we regard a type as consisting of a base type —such as integer, boolean, string, lists of pairs of reals, functions from strings to integers, etc.— and a set of operations on that base type. We say

$$\text{integer} : \{read\}$$

for instance, to denote a base-type integer on which *read* is the only permissible operation. Let $S \subseteq T$ denote that S is a *subtype* of T, given that

S and T have the same base type and the set of permissible operations in S is a subset of the corresponding operations in T. Thus,

$$(\text{integer} : \emptyset) \subseteq (\text{integer} : \{read\}) \subseteq \text{integer}$$

Here, the last occurrence of "integer" denotes the full integer type on which all available operations on integers are allowed.

The relation $\subseteq$ is a partial order and the set of types formed from a given base type is a lattice with respect to $\subseteq$. For S and T having the same base type, we write $S \cup T$, $S \cap T$ for the least upper bound and the greatest lower bound of S and T; these are obtained from S and T by taking union and intersection, respectively, of the associated operation sets. In a base type, say integer,

integer : $\emptyset$	is the bottom element of the corresponding lattice
integer	is the top element.

Syntactic conventions for type declarations

The following example illustrates typical declarations of global variables.

global integer	x : (**intl**: $read, increment$; **extl**: $read$)
global integer	y : (**intl**: $read, increment$)
global integer	z : (**intl**: $read$; **extl** : $\emptyset$)
global seq(integer)	c : (**intl**: $receive$; **extl**: $send$)

In this example, variable x can be "read" or "incremented" in F, but it can only be "read" outside F. Variable y is accessed similarly to x within F; however, it is not restricted in any way in the external programs (an unspecified field denotes that all operations are permissible). Variable z has no permissible external operations, so it is a local variable of F. Variable c can be used to represent a channel for which F is the unique receiver.

Let $F.x.\textbf{intl}$, $F.x.\textbf{extl}$ denote the internal and external type of x in F, respectively. If x is not declared in F and the base type of x is T, then take

$F.x.\textbf{intl}$	to be $T : \emptyset$
$F.x.\textbf{extl}$	to be T

Note We often omit the base type, T, for $F.x.\textbf{intl}$ and $F.x.\textbf{extl}$, when it is clear from the context. □

Link constraint

"Type check" usually refers to a compiler determining the consistency of the internal type declarations. We propose that the linker of the boxes of a program also perform a type check, as follows. For every global variable x declared in $\langle [\!] \; i :: F_i \rangle$, program composition has to satisfy

Link Constraint $\quad \langle \forall\ i, j :\ i \neq j :\ F_i.x.\mathbf{intl} \subseteq F_j.x.\mathbf{extl} \rangle$

That is, for any j, the external type declaration of x in F_j is satisfied by internal declaration in every F_i. If x is not declared in F_i, then the link constraint is trivially satisfied by F_i; see syntactic conventions above.

In chapter 8, union of programs F and G was defined provided that the initial conditions and the declarations in F and G were consistent. Now we have generalized the notion of variable declaration; consistency of declarations means satisfaction of the link constraint.

Type declaration in a composite program

The internal and external types of a global variable x in $\langle [\!] \ i ::\ F_i \rangle$ are given by

$$\langle [\!] \ i ::\ F_i \rangle.x.\mathbf{intl} = \langle \cup\ i ::\ F_i.x.\mathbf{intl} \rangle$$
$$\langle [\!] \ i ::\ F_i \rangle.x.\mathbf{extl} = \langle \cap\ i ::\ F_i.x.\mathbf{extl} \rangle$$

That is, the internal operations on x in $\langle [\!] \ i ::\ F_i \rangle$ are the ones that are internal to some F_i and an operation is externally allowable in $\langle [\!] \ i ::\ F_i \rangle$ iff it is allowable in every F_i.

Let H be an abstraction of the program $\langle [\!] \ i ::\ F_i \rangle$ (see section 8.2.2). The type of a global variable x in H may be redefined as long as

$$H.x.\mathbf{intl} \supseteq \langle [\!] \ i ::\ F_i \rangle.x.\mathbf{intl}$$
$$H.x.\mathbf{extl} \subseteq \langle [\!] \ i ::\ F_i \rangle.x.\mathbf{extl}$$

That is, the internal type of x in H supports the operations in all the components, and the external type meets the restrictions imposed by all the components.

Localizing a variable

A variable may be declared local to components F and G by constructing a program that includes F and G as components in which the variable is declared to be local, as in section 8.2.2. An alternative is to use the following implicit mechanism that we illustrate with an example.

Suppose that only the operations $\{\alpha, \beta, \gamma\}$ can be applied to variable x. If we declare

$$F.x.\mathbf{extl} = \{\beta, \gamma\}, \quad G.x.\mathbf{extl} = \{\alpha, \gamma\}, \quad H.x.\mathbf{extl} = \{\alpha, \beta\}$$

then

$$\begin{aligned} & (F \ [\!] \ G \ [\!] \ H).x.\mathbf{extl} \\ = \ & F.x.\mathbf{extl} \cap G.x.\mathbf{extl} \cap H.x.\mathbf{extl} \\ = \ & \emptyset \end{aligned}$$

Thus, no program other than F, G, and H can access x.

This mechanism can be used to declare that a channel c between a sender and a receiver is local to these two programs, as shown below.

$$\begin{array}{ll} \text{sender}.c.\mathbf{extl} = \{receive\} & \text{, declare} \\ \text{receiver}.c.\mathbf{extl} = \{send\} & \text{, declare} \\ (\text{sender} \parallel \text{receiver}).c.\mathbf{extl} = \emptyset & \text{, definition of } \mathbf{extl} \end{array}$$

Axioms of union

We proposed three axioms for union in section 8.5.1. The definition of union has become more restrictive with the adoption of link constraint. Yet we have the same set of axioms, and it can be shown that the union theorem and its corollaries from section 8.2.3 are still valid.

9.3.2 Definitions of closure properties

For a property Π —where Π is a **co**, **en**, $\mapsto$, **stable**, **invariant**, **transient**, or **constant**— define its closure, **c**Π, as follows. In the following, G is quantified over all programs such that $F \parallel G$ is defined.

$$\mathbf{c}\Pi \;\text{ in } F \;\equiv\; \langle \forall\, G ::\; \Pi \;\text{ in } F \parallel G \rangle$$

Thus, for example

$$\begin{array}{l} \mathbf{cstable}\; p \;\text{ in } F \;\equiv\; \langle \forall\, G ::\; \mathbf{stable}\; p \;\text{ in } F \parallel G \rangle \\ p \;\; \mathbf{c}\!\mapsto \;\; q \;\text{ in } F \;\equiv\; \langle \forall\, G ::\; p \;\mapsto\; q \;\text{ in } F \parallel G \rangle \end{array}$$

9.3.3 Closure theorem

Closure theorem In the following, G is quantified over all programs where $F \parallel G$ is defined.

1. $p \;\; \mathbf{cco} \;\; q \;\text{ in } F \;\equiv\; p \;\; \mathbf{co} \;\; q \;\text{ in } F \;\wedge\; \langle \forall\, G ::\; p \;\; \mathbf{co} \;\; q \;\text{ in } G \rangle$

2. $\mathbf{ctransient}\; p \;\text{ in } F \;\equiv\; \mathbf{transient}\; p \;\text{ in } F$

3. $p \;\; \mathbf{cen} \;\; q \;\text{ in } F \;\equiv\; p \;\; \mathbf{en} \;\; q \;\text{ in } F \wedge \langle \forall\, G ::\; p \wedge \neg q \;\; \mathbf{co} \;\; p \vee q \;\text{ in } G \rangle$

4. $\mathbf{cinvariant}\; p \;\text{ in } F \;\equiv\; \mathbf{invariant}\; p \;\text{ in } F \;\wedge\; \langle \forall\, G ::\; \mathbf{stable}\; p \;\text{ in } G \rangle$

5. $\mathbf{cstable}\; p \;\text{ in } F \;\equiv\; \mathbf{stable}\; p \;\text{ in } F \;\wedge\; \langle \forall\, G ::\; \mathbf{stable}\; p \;\text{ in } G \rangle$

6. $\mathbf{cconstant}\; e \;\text{ in } F \;\equiv\; \mathbf{constant}\; e \;\text{ in } F \wedge \langle \forall\, G ::\; \mathbf{constant}\; e \;\text{ in } G \rangle$

Proof:

1. p **cco** q in F
   $\equiv$ {definition of **cco**}
   $\langle \forall\, G ::\ p$ **co** q in $F \parallel G \rangle$
   $\equiv$ {union theorem}
   $\langle \forall\, G ::\ p$ **co** q in $F \ \wedge\ p$ **co** q in $G \rangle$
   $\equiv$ {range of quantification is nonempty because of id}
   p **co** q in $F \ \wedge\ \langle \forall\, G ::\ p$ **co** q in $G \rangle$

2. **ctransient** p in F
   $\equiv$ {definition of **ctransient**}
   $\langle \forall\, G ::$ **transient** p in $F \parallel G \rangle$
   $\equiv$ {instantiating G by id and using $F \parallel id = F$}
   $\langle \forall\, G ::$ **transient** p in $F \ \wedge$ **transient** p in $F \parallel G \rangle$
   $\equiv$ {union theorem}
   $\langle \forall\, G ::$ **transient** p in $F \wedge ($**transient** p in $F \ \vee$ **transient** p in $G) \rangle$
   $\equiv$ {simplify}
   transient p in F

3. p **cen** q in F
   $\equiv$ {definition of **cen**}
   $\langle \forall\, G ::\ p$ **en** q in $F \parallel G \rangle$
   $\equiv$ {instantiating G by id and using $F \parallel id = F$}
   $\langle \forall\, G ::\ p$ **en** q in $F \ \wedge\ p$ **en** q in $F \parallel\ G \rangle$
   $\equiv$ {expanding p **en** q in $F \parallel G$ using the union theorem corollary}
   $\langle \forall\, G ::\ (p$ **en** q in $F) \ \wedge\ (p \wedge \neg q$ **co** $p \vee q$ in $F) \ \wedge$
   $(p \wedge \neg q$ **co** $p \vee q$ in $G) \ \wedge\ (p$ **en** q in $F \ \vee\ p$ **en** q in $G) \rangle$
   $\equiv$ {simplify using: p **en** q in $F \ \Rightarrow\ p \wedge \neg q$ **co** $p \vee q$ in F}
   $\langle \forall\, G ::\ (p$ **en** q in $F) \ \wedge\ (p \wedge \neg q$ **co** $p \vee q$ in $G) \rangle$
   $\equiv$ {Rewrite. The range of quantification is nonempty.}
   p **en** q in $F \ \wedge\ \langle \forall\, G ::\ p \wedge \neg q$ **co** $p \vee q$ in $G \rangle$

4. **cinvariant** p in F
   $\equiv$ {definition of **cinvariant**}
   $\langle \forall\, G ::$ **invariant** p in $F \parallel G \rangle$
   $\equiv$ {instantiating G by id and using $F \parallel id = F$}
   $\langle \forall\, G ::$ **invariant** p in F, **invariant** p in $F \parallel G \rangle$
   $\equiv$ {**invariant** p in $F \ \Rightarrow\ (F.ic \ \Rightarrow\ p)$,
   invariant p in $F \parallel G \equiv ((F \parallel G).ic \Rightarrow p) \wedge ($**stable** p in $F \parallel G))$}
   $\langle \forall\, G ::\ F.ic \ \Rightarrow\ p$, **invariant** p in F,
   $(F \parallel G).ic \ \Rightarrow\ p$, **stable** p in $F \parallel G \rangle$
   $\equiv$ {union theorem: $(F \parallel G).ic = F.ic \ \wedge\ G.ic$.
   Hence, $(F.ic \ \Rightarrow\ p) \ \Rightarrow\ ((F \parallel G).ic \ \Rightarrow\ p)$}
   $\langle \forall\, G ::\ F.ic \ \Rightarrow\ p$, **invariant** p in F, **stable** p in $F \parallel G \rangle$

$\equiv$ $\{(\textbf{invariant}\ p\ \text{ in } F)\ \Rightarrow\ (F.ic\ \Rightarrow\ p)\}$
$\langle \forall\ G ::\ \textbf{invariant}\ p\ \text{ in } F,\ \textbf{stable}\ p\ \text{ in } F \parallel G \rangle$
$\equiv$ {union theorem}
$\langle \forall\ G ::\ \textbf{invariant}\ p\ \text{ in } F,\ \textbf{stable}\ p\ \text{ in } F,\ \textbf{stable}\ p\ \text{ in } G \rangle$
$\equiv$ $\{(\textbf{invariant}\ p\ \text{ in } F)\ \Rightarrow\ (\textbf{stable}\ p \text{ in } F)\}$
$\langle \forall\ G ::\ \textbf{invariant}\ p\ \text{ in } F,\ \textbf{stable}\ p\ \text{ in } G \rangle$
$\equiv$ {Rewrite. Range of quantification is nonempty because of id }
$\textbf{invariant}\ p\ \text{ in } F\ \wedge\ \langle \forall\ G ::\ \textbf{stable}\ p\ \text{ in } G \rangle$

We leave the proofs of (5) and (6) as exercises. □

Corollaries

It is easy to deduce the following corollaries from the proofs of (3,4) in the closure theorem. As before, G is quantified over all programs where $F \parallel G$ is defined.

(3′) $p\ \textbf{cen}\ q\ \text{ in } F\ \equiv$
$(\textbf{transient}\ p \wedge \neg q \text{ in } F)\ \wedge\ (p \wedge \neg q\ \textbf{cco}\ p \vee q\ \text{ in } F)$

(4′) $\textbf{cinvariant}\ p\ \text{ in } F\ \equiv\ \textbf{initially}\ p \text{ in } F\ \wedge\ \textbf{cstable}\ p\ \text{ in } F$

The closure theorem is the basis for establishing closure properties, analogous to the union theorem for compositional properties. Most cases in the closure theorem require us to prove a certain property in G whenever $F \parallel G$ is defined. This is where type declarations play a role. From the external declarations of a global variable x in F, we can often establish that the variable value is not changed externally; then x is constant in G. Similarly, if an integer variable y can be only read or incremented externally, $\textbf{stable}\ y \geq m$ in G, for any m. The following corollary exploits these possibilities.

Corollary For a program F, let $\textbf{stable}\ p$ in G and $\textbf{constant}\ e$ in G hold for all G whenever $F \parallel G$ is defined.

$p\ \textbf{cco}\ q\ \text{ in } F\ \equiv\ p\ \textbf{co}\ q\ \text{ in } F$ (1)
$p\ \textbf{cen}\ q\ \text{ in } F\ \equiv\ p\ \textbf{en}\ q\ \text{ in } F$ (2)
$\textbf{cinvariant}\ p\ \text{ in } F\ \equiv\ \textbf{invariant}\ p\ \text{ in } F$ (3)
$\textbf{cstable}\ p\ \text{ in } F\ \equiv\ \textbf{stable}\ p\ \text{ in } F$ (4)
$\textbf{cconstant}\ e\ \text{ in } F\ \equiv\ \textbf{constant}\ e\ \text{ in } F$ (5)

Proof: Note that

$(\textbf{stable}\ p\ \text{ in } G\ \wedge\ p\ \Rightarrow\ q)\ \Rightarrow\ (p\ \textbf{co}\ q\ \text{ in } G)$

Also, by strengthening the lhs and weakening the rhs,

$(\textbf{stable}\ p\ \text{ in } G)\ \Rightarrow\ (p \wedge \neg q\ \textbf{co}\ p \vee q\ \text{ in } G)$

The corollaries then follow from the closure theorem. □

9.3.4 Derived rules

There is a closure operator corresponding to every safety and progress operator; this suggests a large number of derived rules using different combinations of these operators. Fortunately, the following three meta-rules suffice. As before, Π is any property and $\mathbf{c}\Pi$ its closure.

- (inflation) $\dfrac{\mathbf{c}\Pi \text{ in } F}{\Pi \text{ in } F}$

- (lifting) $\dfrac{\mathbf{c}\Pi \text{ in } F}{\mathbf{c}\Pi \text{ in } F \parallel G}$ provided that $F \parallel G$ is defined.

- (coercion) Any inference rule remains valid if each operator is replaced by its closure.

The inflation rule allows us to prove/implement a property Π by proving/implementing $\mathbf{c}\Pi$. The lifting rule says that a closure property of a component is inherited by the system; this allows modular construction of a system by partitioning the implementations of the closure properties among the components. The coercion rule allows us to manipulate the closure operators in exactly the same manner as the original safety and progress operators.

Proofs of the derived rules

Proof of inflation

$\mathbf{c}\Pi$ in F
$\equiv$ {definition of closure}
$\langle \forall\, G :: \Pi \text{ in } F \parallel G \rangle$
$\Rightarrow$ {setting G to id}
Π in $F \parallel id$
$\equiv$ {$F \parallel id = F$}
Π in F □

Proof of lifting

$\mathbf{c}\Pi$ in F
$\equiv$ {definition of closure}
$\langle \forall\, F' :: \Pi \text{ in } F \parallel F' \rangle$
$\Rightarrow$ {instantiate F' by $G \parallel H$, for the given G and any H such that $F \parallel (G \parallel H)$ is defined}
$\langle \forall\, H :: \Pi \text{ in } F \parallel (G \parallel H) \rangle$
$\equiv$ {associativity of $\parallel$}
$\langle \forall\, H :: \Pi \text{ in } (F \parallel G) \parallel H \rangle$
$\equiv$ {definition of closure}
$\mathbf{c}\Pi$ in $F \parallel G$ □

Proof of coercion

Consider an inference rule of the form $\frac{\Pi}{\Pi'}$. We show $\frac{\mathbf{c}\Pi}{\mathbf{c}\Pi'}$.

$$
\begin{array}{ll}
 & \mathbf{c}\Pi \ \text{ in } F \\
\equiv & \quad \{\text{definition of closure}\} \\
 & \langle \forall\, G :: \ \Pi \ \text{ in } F \parallel G \rangle \\
\Rightarrow & \quad \{\text{from the given inference rule, } \Pi \ \Rightarrow \ \Pi'\} \\
 & \langle \forall\, G :: \ \Pi' \ \text{ in } F \parallel G \rangle \\
\equiv & \quad \{\text{definition of closure}\} \\
 & \mathbf{c}\Pi' \ \text{ in } F
\end{array}
$$

□

As an application of coercion, consider the transitivity rule for *leads-to*:

$$\frac{p \ \mapsto \ q, \ q \ \mapsto \ r}{p \ \mapsto \ r}$$

Coercion permits us to assert

$$\frac{p \ \mathbf{c}\mapsto \ q, \ q \ \mathbf{c}\mapsto \ r}{p \ \mathbf{c}\mapsto \ r}$$

Substitution axiom

In chapters 5 and 6, we freely used the substitution axiom to replace any invariant by *true* and vice versa in any property of a program. The substitution axiom does not apply to union of two programs if properties are derived in the components using the invariants of those components; see section 8.4. Fortunately, we can now formulate a substitution axiom that applies to program union, and it reduces to the simpler version (of chapters 5 and 6).

Statement of the substitution axiom:

Predicate I can be replaced by *true* and vice versa in any property of a component F_i provided that

$$\textbf{cinvariant } I \ \text{ in } \langle [\!] \ i :: \ F_i \rangle$$

Discussion

Closure permits us to ignore the identity of the program for which a property has been derived because, using lifting, the closure property is a property of any system of which the given program is a component. Therefore, much of the reasoning can be done for a single program, in a manner similar to what we did in chapters 5 and 6. As an added bonus, the coercion rule permits us to apply the same inference rules. We illustrate the power of this approach by redoing some of the examples of the earlier chapters.

9.3.5 Example: handshake protocol

In this section we consider the handshake-protocol from section 8.3.2, which is also treated in section 9.2.1. Box *send* sends messages; a subsequent message is sent only on receiving an acknowledgment to the last message. Let ns and nr denote the number of messages sent and acknowledged. We write the code for *send* with the appropriate declarations for the global variables ns and nr. The operation *increment* on an integer increases it by 1.

---

box *send*
  global integer $ns = 0$ (**intl**: $read, increment$; **extl**: $read$);
  global integer $nr = 0$ (**intl**: $read$; **extl**: $read, increment$);

    $nr \geq ns \;\rightarrow\; increment.ns$
end $\{send\}$

We prove below the following closure properties of *send*. Henceforth, m and n are arbitrary integers.

cstable $m \leq nr$ in *send*, and
cstable $n \leq ns$ in *send* (S1)
cinvariant $0 \leq nr$ in *send*, and
cinvariant $ns \leq nr + 1$ in *send* (S2)
$nr \geq ns \;\wedge\; ns = n \;\; \mathbf{c}\!\!\mapsto \;\; ns = n + 1$ in *send* (S3)

Note The informal description makes it clear that the environment of *send* increases nr only once after receiving a message, i.e., if $nr < ns$. However, our type declaration mechanism is not strong enough to express this fact. The external type declaration of nr in *send* permits *send*'s environment to increase nr arbitrarily. Hence, we cannot establish $nr \leq ns$ for the system from *send*'s text alone. □

Proofs of (S1–S3)

From the type declarations in *send*, we deduce the following properties for *send* and for any G, where *send* $[\!]$ G is defined.

constant ns in G
stable $m \leq nr$ in G
constant nr in *send*
stable $n \leq ns$ in *send*

We leave the proof of property (S1) to the reader; it is easily established from the above properties using the corollary of the union theorem.

• Proof of (S2), **cinvariant** $ns \leq nr + 1$ in *send*:

stable $m \leq nr$ in G	, from type declaration in *send*
stable $ns - 1 \leq nr$ in G	, **constant** $(ns - 1)$ in G; replace m by $(ns - 1)$ above
invariant $ns - 1 \leq nr$ in *send*	, text of *send*
cinvariant $ns - 1 \leq nr$ in *send*	, closure theorem on above two
cinvariant $ns \leq nr + 1$ in *send*	, rewrite

The remaining part of (S2) can be proved similarly. □

• Proof of (S3), $nr \geq ns \ \wedge \ ns = n$ **c↦** $ns = n + 1$ in *send*:
For any G where *send* $[\!]$ G is defined

stable $nr \geq n$ in G	, type declaration in *send*
stable $ns = n$ in G	, type declaration in *send*
stable $nr \geq n \ \wedge \ ns = n$ in G	, stable conjunction
$nr \geq n \ \wedge \ ns = n$ **en** $ns = n + 1$ in *send*	, text of *send*
$nr \geq ns \ \wedge \ ns = n$ **cen** $ns = n + 1$ in *send*	, closure theorem corollary on above two, rewrite lhs
$nr \geq ns \ \wedge \ ns = n$ **c↦** $ns = n + 1$ in *send*	, apply coercion on above

□

The receive program

---

box *receive*
  global integer $ns = 0$ (**intl**: *read*; **extl**: *read*, *increment*);
  global integer $nr = 0$ (**intl**: *read*, *increment*; **extl**: *read*);

  $nr < ns \ \rightarrow$ *increment*.*nr*
end {*receive*}

It is easy to see that *send* $[\!]$ *receive* is defined, because the following link constraints are satisfied.

send.*ns*.**intl** $\subseteq$ *receive*.*ns*.**extl**
send.*nr*.**intl** $\subseteq$ *receive*.*nr*.**extl**
receive.*ns*.**intl** $\subseteq$ *send*.*ns*.**extl**
receive.*nr*.**intl** $\subseteq$ *send*.*nr*.**extl**

We have the following properties for *receive*, analogous to those for *send*. We leave their proofs to the reader.

cinvariant $nr \leq ns$ in *receive* (R1)
$nr < ns \ \wedge \ nr = m$ **c↦** $nr = m + 1$ in *receive* (R2)

Deriving properties of send ‖ receive

Applying lifting, each of the properties (S1–S3) and (R1–R2) is a property in *send* ‖ *receive*. Therefore,

cstable $m \leq nr$, **cstable** $n \leq ns$	, from (S1)	(SR1)
cinvariant $0 \leq nr \leq ns \leq nr + 1$	, from (S2, R1)	(SR2)
$nr \geq ns \ \wedge \ ns = n \ \ \mathbf{c}\!\!\mapsto \ ns = n + 1$	, from (S3)	(SR3)
$nr < ns \ \wedge \ nr = m \ \ \mathbf{c}\!\!\mapsto \ nr = m + 1$	, from (R2)	(SR4)

Note that only parts of (SR2) is proved in each of *send* and *receive*.

• *true* $\mathbf{c}\!\!\mapsto \ nr > m$ in *send* ‖ *receive*: (SR5)
Proof: In the following proof, all properties are in *send* ‖ *receive*.

$nr = n$
$\Rightarrow$ {use SR2}
$nr, ns = n, n \ \vee \ nr, ns = n, n + 1$
$\mathbf{c}\!\!\mapsto$ {use SR3 on the first disjunct}
$ns = n + 1 \ \vee \ nr, ns = n, n + 1$
$\Rightarrow$ {use SR2 to expand the first disjunct}
$nr, ns = n + 1, n + 1 \ \vee \ nr, ns = n, n + 1$
$\mathbf{c}\!\!\mapsto$ {use SR4 on the second disjunct}
$nr, ns = n + 1, n + 1 \ \vee \ nr = n + 1$
$\Rightarrow$ {simplify}
$nr > n$

From above, conclude *true* $\mathbf{c}\!\!\mapsto \ nr > m$ using induction. □

9.3.6 Example: concurrent bag

We consider the example of concurrent bag from section 9.2.5. The following is a specification of a bag B of size $N, N \geq 0$, using closure operators.

Specification B
local bag($\top$**)** b;
global $(\top \ \cup \ \{\phi\})$ r (**intl**: *read*; **extl**: *write*);
global $(\top \ \cup \ \{\phi\})$ w (**intl**: *write*; **extl**: *read*);

{The following properties are in B}

initially $b + w = \phi$	(B0)
cinvariant $\lvert b \rvert \leq N$	(B1)
constant $r + b + w$	(B2)
$\lvert b + w \rvert \leq N \ \ \mathbf{c}\!\!\mapsto \ r = \phi$	(B3)
$\lvert b + r \rvert > 0 \ \ \mathbf{c}\!\!\mapsto \ w \neq \phi$	(B4)

end $\{B\}$

The semantics of *read* and *write* are as follows. The operation *read* on x has the post-condition $x = \phi$, and *write* on x has the pre-condition $x = \phi$ and post-condition $x \neq \phi$.

Note the following differences from the specification of section 9.2.5. The specification does not include

stable $r \preceq R$, **stable** $W \preceq\ w$ in B

because these can be deduced from the internal type declaration of r and w. Similarly, the assumptions about the environment have been eliminated and all the properties, except (B2), are replaced by closures.

Bag concatenation

We redo the proof of bag concatenation to illustrate the advantages of working with closure properties. First, we rewrite the specifications of BM and BN (see page 292) using closure properties. For BM, we have a bag of size M, with input and output variables r and v, respectively. Its specification is same as that of B with bm, v, and M replacing b, w, and N.

Observe from the specifications that $BM \parallel BN$ is defined because BM and BN satisfy the link constraints, and their initial conditions are compatible. It is interesting to note that

$$\begin{aligned} & (BM \parallel BN).v.\mathbf{extl} \\ = \ & BM.v.\mathbf{extl} \cap BN.v.\mathbf{extl} \\ = \ & \emptyset \end{aligned}$$

Therefore, v cannot be accessed outside $BM \parallel BN$; effectively, it is a local variable of $BM \parallel BN$.

Specification BM
local bag($\top$**)** bm;
global $(\top \cup \{\phi\})$ r (**intl**: *read*; **extl**: *write*);
global $(\top \cup \{\phi\})$ v (**intl**: *write*; **extl**: *read*);

{The following properties are in BM}
initially $bm + v = \phi$ (BM0)
cinvariant $|bm| \leq M$ (BM1)
constant $r + bm + v$ (BM2)
$|bm + v| \leq M$ **c↦** $r = \phi$ (BM3)
$|bm + r| > 0$ **c↦** $v \neq \phi$ (BM4)
end $\{BM\}$

Similarly, BN is a bag of size N, with input and output variables v and w. Its specification may be obtained from that of B by replacing b and r by bn and v, respectively.

Specification BN
local bag($\top$**)** bn;
global $(\top \cup \{\phi\})$ v (**intl**: *read*; **extl**: *write*);
global $(\top \cup \{\phi\})$ w (**intl**: *write*; **extl**: *read*);

{The following properties are in BN}
initially $bn + w = \phi$ (BN0)
cinvariant $|bn| \leq N$ (BN1)
constant $v + bn + w$ (BN2)
$|bn + w| \leq N \;\; \mathbf{c}\!\!\mapsto \;\; v = \phi$ (BN3)
$|bn + v| > 0 \;\; \mathbf{c}\!\!\mapsto \;\; w \neq \phi$ (BN4)
end $\{BN\}$

Now we show that $BM \parallel BN$ satisfies the specification B (with N replaced by $M + N + 1$). As before, we define b by

$$b = bm + v + bn \tag{D}$$

All properties in these proofs are in $BM \parallel BN$, except where shown.

• Proof of (B0) **initially** $b + w = \phi$:
initially $bm + v = \phi$, (BM0) by lifting
initially $bn + w = \phi$, (BN0) by lifting
initially $b + w = \phi$, above two and (D) □

• Proof of (B1) $|b| \leq M + N + 1$:
$|b|$
$=$ {from (D), $b = bm + v + bn$}
$|bm| + |v| + |bn|$
$\leq$ {from BM1 by lifting, and $|v| = 1$}
$M + 1 + |bn|$
$\leq$ {from BN1 by lifting}
$M + N + 1$ □

• Proof of (B2) **constant** $r + b + w$:
The proof is similar to the corresponding proof in section 9.2.5.
constant $r + bm + v$ in BM , BM2
constant $bn + w$ in BM , locality
constant $r + b + w$ in BM , above two and (D)
constant $r + b + w$ in BN , similarly
constant $r + b + w$ in $BM \parallel BN$, union theorem corollary

Note This proof cannot exploit the closure theorem because BM2 and BN2 are not closure properties. □

• Proof of (B3) $|b+w| \leq M+N+1 \;\; \mathbf{c}\!\!\mapsto \;\; r=\phi$:

$|b+w| \leq M+N+1$
$\Rightarrow$ {from (D), $b = bm+v+bn$}
$|bm+v| \leq M \;\vee\; |bn+w| \leq N$
$\mathbf{c}\!\!\mapsto$ {(BN3) by lifting on the second disjunct}
$|bm+v| \leq M \;\vee\; v=\phi$
$\Rightarrow$ {from (BM1) by lifting $|bm| \leq M$. So, $v=\phi \;\Rightarrow\; |bm+v| \leq M$}
$|bm+v| \leq M$
$\mathbf{c}\!\!\mapsto$ {(BM3) by lifting}
$r=\phi$ □

• Proof of (B4), $|b+r|>0 \;\; \mathbf{c}\!\!\mapsto \;\; w \neq \phi$:
Proof is similar to that of (B3). □

It is instructive to compare the length of the proof of (B3) with the analogous result, (B5), proved on page 293 with conditional properties. Use of coercion and lifting simplifies the proof considerably.

9.3.7 Example: token ring

We have considered this example in sections 5.5.3 and 6.5.3. Previously, we started by postulating certain properties for the entire token ring, shown as (TR0–TR7) below, from which we derived mutual exclusion and absence of starvation. Now, we start with simpler properties of the individual components in the ring and show that (TR0–TR7) are satisfied. The properties of the components are simple enough that they are easily implemented; see Refinement of the Specification on page 312.

Specification of the token ring

Let R be a finite ring of N processes of which box i is R_i, $0 \leq i < N$. The entire ring R is $\langle [\!] \; i : \; 0 \leq i < N : \; R_i \rangle$, for some N, $N \geq 0$. Each box R_i has a local state that takes on three possible values: h, e, and t, for *hungry*, *eating*, and *thinking*. Let the predicates h_i, e_i, and t_i mean that the state of R_i is *hungry*, *eating*, *thinking*, respectively. Clearly, these predicates are pairwise disjoint and $h_i \vee e_i \vee t_i$ holds.

There is a single global variable p whose value is the index of the token holder. The properties of R that we had postulated in the earlier chapters are given below. Here, i ranges over $0 \leq i < N$, and i' is the index of the right neighbor of process i in the ring. All the properties (TR0–TR7) are in R.

$$\textbf{initially} \; \langle \forall \; j :: \; e_j \;\Rightarrow\; p=j \rangle \qquad \text{(TR0)}$$
$$e_i \;\; \textbf{co} \;\; e_i \;\vee\; t_i \qquad \text{(TR1)}$$
$$t_i \;\; \textbf{co} \;\; t_i \;\vee\; h_i \qquad \text{(TR2)}$$

h_i **co** $h_i \vee e_i$ (TR3)
$h_i \wedge p \neq i$ **co** h_i (TR4)
$p = i$ **co** $p = i \vee \neg e_i$ (TR5)
$h_i \wedge p = i \mapsto e_i$ (TR6)
$p = i \mapsto p = i'$ (TR7)

We had proved earlier that for all i and j, $0 \leq i < N$ and $0 \leq j < N$,

(mutual exclusion) $e_i \wedge e_j \Rightarrow i = j$ in R (in section 5.5.3)
(absence of starvation) $h_j \mapsto e_j$ in R (in section 6.5.3)

Specifications of processes

We propose specifications for boxes R_i and prove all the properties of R, (TR0–TR7), from these specifications. Since only a token holder can modify the token's position, p can be modified in R_i only if $p = i$. Thus, the only operations that modify p are, for all i, $0 \leq i < N$,

$increment.i :: \quad p = i \rightarrow p := (p + 1) \ mod \ N$

It follows that **stable** $p = i$ in R_j, $j \neq i$.

Specification R_i $(0 \leq i < N)$ {All properties below are in R_i}
global enum $(0..N-1)$ p
(**intl**: $read, increment.i$; **extl**: $read, increment.j$, $j \neq i$);
local boolean h_i, e_i, t_i;

initially $\langle \forall j :: e_j \Rightarrow p = j \rangle$ (R0)
e_i **co** $e_i \vee t_i$ (R1)
t_i **co** $t_i \vee h_i$ (R2)
h_i **co** $h_i \vee e_i$ (R3)
stable $h_i \wedge p \neq i$ (R4)
$p = i$ **co** $p = i \vee \neg e_i$ (R5)
$h_i \wedge p = i$ **c**$\mapsto$ e_i (R6)
$t_i \wedge p = i$ **c**$\mapsto$ $(h_i \wedge p = i) \vee p = i'$ (R7)
e_i **c**$\mapsto$ $p = i'$ (R8)
end $\{R_i\}$

Observe that $R = \langle [\!] \ i : \ 0 \leq i < N : \ R_i \rangle$ is well defined because

$R_i.p.\mathbf{intl} \subseteq R_j.p.\mathbf{extl}$

for all i and j, $i \neq j$, and the initial conditions of the R_i's are consistent.

Proofs of (TR0–TR7)

We prove that properties (TR0–TR7) hold in R given that R_i, $0 \leq i < N$, has the properties (R0–R8). The initial condition, (R0), and the safety properties, (R1–R5), are the counterparts of (TR0–TR5); we establish each of the latter from the corresponding property of R_i using the type declaration of p. The progress property, (R6), appears similar to (TR6), but the former is a property of a component whereas the latter is a property of the entire system; we prove (TR6) from (R6) using lifting and inflation. Progress properties (R6, R7, R8) are used to prove (TR7).

- Proof of (TR0), **initially** $\langle \forall\ i ::\ e_i\ \Rightarrow\ p = i \rangle$:

  From (R0) and the union theorem. □

Let G be a program such that $R_i \ [\!] \ G$ is defined. We have, from locality,

$$\textbf{stable}\ h_i\ \text{ in } G,\ \textbf{stable}\ e_i\ \text{ in } G,\ \textbf{stable}\ t_i\ \text{ in } G \qquad \text{(R9)}$$

From the external type declaration of p

$$\textbf{stable}\ p = i\ \text{ in } G \qquad \text{(R10)}$$

Forming stable conjunction of (R10) with the individual properties in (R9), we get in G:

$$\textbf{stable}\ (h_i \wedge p = i)\ ,\ \textbf{stable}\ (e_i \wedge p = i)\ ,\ \textbf{stable}\ (t_i \wedge p = i) \qquad \text{(R11)}$$

- Proof of (TR1), $e_i\ \textbf{co}\ e_i\ \vee\ t_i\ \text{ in } R$:

$e_i\ \textbf{co}\ e_i \vee t_i\ \text{ in } R_i$	, from (R1)
$\textbf{stable}\ e_i\ \text{ in } G$	, from (R9)
$e_i\ \textbf{cco}\ e_i \vee t_i\ \text{ in } R_i$	, closure theorem corollary
$e_i\ \textbf{cco}\ e_i \vee t_i\ \text{ in } R$	, lifting
$e_i\ \textbf{co}\ e_i \vee t_i\ \text{ in } R$	, inflation □

- Proofs of (TR2, TR3):
  Similar to (TR1); use (R2, R9) for (TR2) and (R3, R9) for (TR3). □

- Proof of (TR4), $h_i\ \wedge\ p \neq i\ \textbf{co}\ h_i\ \text{ in } R$:

$\textbf{stable}\ h_i\ \text{ in } G$	, from (R9)
$h_i\ \wedge\ p \neq i\ \textbf{co}\ h_i\ \text{ in } G$	, strengthen lhs
$\textbf{stable}\ h_i\ \wedge\ p \neq i\ \text{ in } R_i$	, from (R4)
$h_i\ \wedge\ p \neq i\ \textbf{co}\ h_i\ \text{ in } R_i \ [\!] \ G$	, union theorem corollary
$h_i\ \wedge\ p \neq i\ \textbf{cco}\ h_i\ \text{ in } R_i$	, definition of **cco**
$h_i\ \wedge\ p \neq i\ \textbf{co}\ h_i\ \text{ in } R$	, lifting; inflation □

• Proof of (TR5): similar to that of (TR1) using (R5, R10). □

• Proof of (TR6) $h_i \ \wedge \ p = i \ \mapsto \ e_i$ in R:
From (R6), using lifting and inflation. □

• Proof of (TR7), $p = i \ \mapsto \ p = i'$ in R:

In R_i
$p = i$
$\equiv \quad \{t_i \ \vee \ h_i \ \vee \ e_i \ \equiv \ true\}$
$(t_i \ \wedge \ p = i) \ \vee \ (h_i \ \wedge \ p = i) \ \vee \ (e_i \ \wedge \ p = i)$
$\mathbf{c}\mapsto \{$(R7): replace $(t_i \ \wedge \ p = i)$ above by $(h_i \ \wedge \ p = i) \ \vee \ p = i'\}$
$(h_i \ \wedge \ p = i) \ \vee \ p = i' \ \vee \ (e_i \ \wedge \ p = i)$
$\mathbf{c}\mapsto \{$(R6): replace $(h_i \ \wedge \ p = i)$ above by $e_i\}$
$e_i \ \vee \ p = i'$
$\mathbf{c}\mapsto \{$(R8): replace e_i above by $p = i'\}$
$p = i'$

From $p = i \ \ \mathbf{c}\mapsto \ \ p = i'$ in R_i, we can derive (TR7) by lifting and inflation. □

Refinement of the specification

The safety properties (R1–R5) in the specification of R_i are easily implemented. But how do we implement the closure properties (R6–R8)? We propose that an implementation be derived through successive refinements of the specifications. A simple refinement is to replace $\mathbf{c}\mapsto$ in (R6–R8) by **cen** to obtain (R6′–R8′), given below.

In R_i ::
$h_i \ \wedge \ p = i \ \ \mathbf{cen} \ \ e_i$ (R6′)
$t_i \ \wedge \ p = i \ \ \mathbf{cen} \ \ (h_i \ \wedge \ p = i) \ \vee \ p = i'$ (R7′)
$e_i \ \ \mathbf{cen} \ \ p = i'$ (R8′)

The correctness of this refinement is straightforward: (R6–R8) follow from (R6′–R8′) by coercion. Next, we claim that each of (R6′–R8′) can be refined to an **ensures** property. We show the proof for R6′; proofs for R7′ and R8′ are similar.

• Proof of (R6′), $h_i \ \wedge \ p = i \ \ \mathbf{cen} \ \ e_i$:

$h_i \ \wedge \ p = i \ \ \mathbf{en} \ \ e_i$	, postulate
stable $h_i \ \wedge \ p = i$ in G	, from R11
$h_i \ \wedge \ p = i \ \ \mathbf{cen} \ \ e_i$	, closure theorem corollary

9.4 Combining Closure and Conditional Properties

Closure properties may appear in the assumption or guarantee part of a conditional property. This is useful when the type definition is not strong enough to specify all possible changes in a shared variable. For example, in the program *send* of the handshake protocol (see page 304), type declarations specify that nr can only be incremented by the environment of *send*. It cannot be specified that nr can increase only up to ns in the environment; our type mechanism is not strong enough to permit such assertions. That is why it is not possible to prove that $|nr - ns| \leq 1$ is an invariant — actually, a **cinvariant**— of the composite program from the text of *send*. However, we can prove the following conditional property of *send*:

$$\begin{array}{l}\langle \forall F : \\ \quad \textbf{cinvariant } nr \leq ns \ \text{ in } F \\ \quad : \\ \quad \textbf{cinvariant } |nr - ns| \leq 1 \text{ in } F \ [\!] \ send \\ \rangle\end{array}$$

This kind of specification is particularly useful for stating facts about shared variables, such as nr and ns, that are local to a subsystem —$send \ [\!] \ receive$, in this case— of a larger system.

The linear network rule of section 9.2.2 has a nice counterpart with closure properties. If each property in that rule is a closure property, then *all* properties hold for the entire network; previously, only Π_N was a property of the network.

Linear network rule

$$\frac{\begin{array}{l}\mathbf{c}\Pi_0 \ \text{ in } f_0 \ \text{ and} \\ \langle \forall\ i :\ 0 \leq i < N : \\ \quad \langle \forall\ g :\ \mathbf{c}\Pi_i \ \text{ in } g :\ \mathbf{c}\Pi_{i+1} \ \text{ in } f_{i+1} \ [\!] \ g \rangle \\ \rangle\end{array}}{\langle \forall\ j :\ 0 \leq j \leq N :\ \mathbf{c}\Pi_j \ \text{ in } \langle [\!]\ i :\ 0 \leq i \leq N :\ f_i \rangle \rangle}$$

The proof is simple. In the linear network rule of section 9.2.2, replace Π_i by $\mathbf{c}\Pi_i$ and N by j, $0 \leq j \leq N$, to conclude that

$$\begin{array}{ll}\mathbf{c}\Pi_j \ \text{ in } \langle [\!]\ i :\ 0 \leq i \leq j :\ f_i \rangle & \text{, above argument} \\ \mathbf{c}\Pi_j \ \text{ in } \langle [\!]\ i :\ 0 \leq i \leq N :\ f_i \rangle & \text{, lifting}\end{array}$$

9.5 Concluding Remarks

Conditional and closure properties simplify specifications and verifications of composite programs. We believe that programmers intuitively employ

some such reasoning for constructing programs in practice. Whenever a component of a program is designed to provide a service —exception handling or message delivery, for instance— it is implicitly assumed that other components of the system do not interfere, and the components provide specific services no matter what system they are embedded in. If accesses to shared variables are not disciplined, program design will have to rely on the general and expensive methods of chapter 8 to make such guarantees.

Closure properties can often replace conditional properties, as we have seen in the example of the handshake protocol (compare the specifications on page 283 and section 9.3.5) and concurrent bag (see page 290 and section 9.3.6). Whenever such a replacement is possible, the accompanying proofs become much shorter. Therefore, specifications using closure properties are always to be preferred. The shortcoming of closure, though, is that assumed properties of the environment have to be expressed within a type system. We could have permitted program texts to include arbitrary assumptions about the environment, but that is no better than writing a conditional property of the program. The restrictions we have imposed allow an automatic linker to decide if two programs can be composed through union. The type mechanism also imposes a discipline on programming.

The type mechanism motivates an object-oriented style of programming like Seuss. We can imagine that *nr* and *ns* in the handshake protocol are objects on which *read* and *increment* are the only applicable methods. The type declaration specifies the access rights to a shared object by a component *and* its environment. Though we do not pursue the topic of access rights any further in this book, we believe that it has the potential to simplify the programming and possibly security aspects of system design.

9.6 Bibliographic Notes

The notion of conditional property is inherent in Misra and Chandy [142] and in the rely-guarantee specifications of Jones [97]. The approach advocated in this chapter is based on [32, section 7.2.2]. Collette [45, 46] and Collette and Knapp [47] contain treatment of composition of rely-guarantee style specifications. Also see de Roever et al. [53] for a comprehensive survey of both compositional and noncompositional verifications, in particular for the verification of network protocols. Results similar to the linear network rule have appeared elsewhere; see, for instance, Lam and Shankar [112]. The factorial network is from [142], and the concurrent bag example is from Misra [136].

10
Reduction Theorem

10.1 Introduction

The operational semantics of the programming model —action systems in chapter 2 and object-oriented systems in chapter 3— is based on tight executions, where each action execution is completed before another one is started. This is a convenient model for understanding a program and reasoning about its properties, because an action represents an indivisible unit whose execution cannot be preempted by another. We applied induction on the number of actions in a tight execution to deduce invariant properties in chapter 5, for instance. We developed a logic to reason about tight executions in chapters 5, 6, 8, and 9, and we extend the logic for the general programming model in chapter 12.

In actual implementations, we would expect the boxes to be partitioned for executions on multiple processors, which may be distributed over a wide-area network. The processors execute the associated actions concurrently; we call such executions *loose*. A typical loose execution of a program will interleave the steps of the individual actions of its components.

It is well known that the net effect of an interleaved execution may be quite different from the sequential executions of its component actions. To see this, consider two boxes α and β, each of which has one action. The actions in α and β (we also call these actions α and β) consist of a sequence of elementary steps α_i and β_i, for $1 \leq i \leq 3$, as shown below.

α:: α_1; α_2; α_3 and
β:: β_1; β_2; β_3 .

The loose execution $\alpha_1\ \beta_1\ \alpha_2\ \beta_2\ \alpha_3\ \beta_3$ interleaves the executions of actions α and β. In this example, suppose α_1 and β_1 are calls on a procedure to read the value of variable x, α_2 and β_2 are local steps to increment the value read and α_3 and β_3 are calls on a procedure to store the incremented value in x. Then the given loose execution increases the value of x by 1, whereas any tight execution of these two actions increases x by 2.

As this example shows, unrestricted concurrent executions may produce results that cannot be produced by tight executions. In this chapter, we develop a set of conditions under which loose executions mimic tight executions. We require that the actions that are concurrently executed satisfy certain *compatibility* conditions, described in section 10.3. Obviously, independent actions that affect states of different boxes may be executed concurrently. Our definition of compatibility is more general, allowing greater potential for concurrency. Roughly, two procedures are compatible if their interleaved execution may be simulated by executing them one after the other in some order. We give an exact definition and show how compatibility of procedures may be proved. We prove a reduction theorem that establishes that if a loose execution of some finite set of actions starting in state u terminates in state v, there is a tight execution that has the same effect.

For a fifo channel on which there is a single sender and a receiver, the sender and the receiver are compatible; see page 325. Thus, all actions in which communication is point-to-point can be executed concurrently, without the need for a scheduler. We discuss this in more detail in section 10.7.

Compatibility information cannot be deduced automatically. Yet it is unrealistic to expect the user to provide this information for all pairs of actions; in most cases, different boxes will be coded by different users, and no user may even know which other actions will be executing. Therefore, we have developed a theory whereby compatibility of procedures in different boxes may be deduced automatically from the compatibility information about procedures that belong to the same box: users simply specify which procedures in a box are compatible and an algorithm then determines which pairs of actions are compatible, so that they may be executed concurrently. The user need not provide complete compatibility information for procedures in a box. If no two procedures are declared compatible, the actions of the program can still be executed concurrently —by having independent actions be executed simultaneously, for instance— but some safe concurrent executions may not be permitted.

Overview of the chapter

In the next section, an abstract model of Seuss programs is given and the model is justified. The model is based on relational calculus; each elementary step of an action is viewed as a binary relation over the program states and an execution of an action is their relational product. In section 10.2.3

we rewrite the restrictions on programs described in section 3.2.6 in terms of the model of this chapter. We show in section 10.2.4 that each procedure can be regarded as a binary relation over the program states. The definition of compatibility appears in section 10.3. Loose executions are defined in section 10.4. A statement of the reduction theorem and its proof are given in section 10.5.

We use some elementary results from relational calculus in this chapter; see appendix A.5 for a brief introduction to this topic.

10.2 A Model of Seuss Programs

In this section, we formalize the notion of box, procedure and executions of procedures (program execution is treated in section 10.5). The *cats* of Seuss are not modeled because they have no relevance at run time. Also, we do not distinguish between action and method because this distinction is unnecessary for the proof of the theorem.

10.2.1 Basic concepts

- A *box* is a pair (S, P) where

  S is a set of *states*
  P is a set of *procedures*

  Each procedure has a unique name and is designated either *partial* or *total*.

- A *procedure* is a tuple (T, N, E) where:

  T is a set of *terminal* symbols;
  each symbol in T is a binary relation over its box states.
  N is a set of *nonterminal* symbols;
  each symbol in N is the name of a procedure of another box.
  E is a nonempty set of *executions*;
  each execution is a finite string over $T \cup N$.

  An execution of a total procedure is a sequence where each element of the sequence is either a terminal or a total procedure of another box. An execution of a partial procedure is of the form $b\ h\ e$, where b is a terminal, h —which is optional— is a nonterminal that names a partial procedure of another box, and e is a sequence in which each element is either a terminal or a total procedure of another box.

- A *program* is a finite set of boxes. Program state is given by the box states. (Therefore, each terminal symbol may be viewed as a binary relation over the program states.)

- An expanded execution of procedure p is a string of terminals, obtained from an execution of p by replacing each nonterminal q by an expanded execution of q.

  We show in section 10.4.2 that this definition is well-grounded.

Conventions:

1. Terminal symbols of different procedures are distinct.

2. Each execution of procedure p begins with a $begin_p$ symbol and ends with a end_p symbol. Both of these are terminal symbols of procedure p.

3. For terminal s, $s.box$ is the box of which s is a symbol. Similarly, $p.box$ is defined for procedure p.

Note We have not specified the initial states of the boxes, because we do not need the initial states to prove the main theorem. □

10.2.2 Justification of the model

A terminal symbol of a procedure denotes a local step within the procedure. A local step can affect only the state of the corresponding box, and we allow a step to have a nondeterministic outcome. Hence, each terminal is modeled as a binary relation over box states.

In the formal model, procedures are parameter-less. Although this would be an absurd assumption in practice, it simplifies mathematical modeling considerably. We justify this assumption as follows. First, we can remove a value parameter from a procedure by creating a set of procedures, one for each possible value of the parameter, and the caller can decide which procedure to call based on the parameter value. Thus, all value parameters may be removed at the expense of increasing the set of procedures. Next, consider a procedure with result parameters; to be specific, let $read(w)$ return a boolean value in w. The caller of *read* cannot decide a priori what the returned value will be. However, we can remove parameter w as follows. First, model *read* by two different procedures, *readt* and *readf*, which return the values *true* and *false*, respectively. Now we have two different execution fragments modeling the call on $read(w)$:

readt; $w :=$ *true*
readf; $w :=$ *false*

An execution that calls $read(w)$ is represented by two executions in our model, one for each possible value returned by *read* for w. Thus, we can remove all parameters from procedures.

Next, we justify the model of procedure execution. An execution is a sequence of steps taken by a procedure and the procedures it calls. For the moment, assume that each procedure has a single alternative; the general case is treated below, though we do not consider negative alternatives at all in this chapter. To motivate further discussion, consider a procedure P that calls $read(w)$, described above, twice in succession. The terminal symbols of P are α and β where

α denotes $w := \mathit{true}$, and β denotes $w := \mathit{false}$

The nonterminals of P are *readt* and *readf*, as described above.

An execution of P does the following steps twice: call *read* and then assign the value returned to w. If P is executed alone, the possible executions are

$begin_P\ readt\ \alpha\ readt\ \alpha\ end_P$
$begin_P\ readf\ \beta\ readf\ \beta\ end_P$

However, if other procedures are executed concurrently with P, the value being read can change in between the two read operations (being written by other concurrently executing procedures). Therefore, the loose executions of P are

$begin_P\ readt\ \alpha\ readt\ \alpha\ end_P$
$begin_P\ readf\ \beta\ readf\ \beta\ end_P$
$begin_P\ readt\ \alpha\ readf\ \beta\ end_P$
$begin_P\ readf\ \beta\ readt\ \alpha\ end_P$

In particular, the execution $begin_P\ readt\ \alpha\ readf\ \beta\ end_P$ denotes that the boolean value is changed from *true* to *false* by another procedure during the two calls to *read* by P. Our goal is to model concurrent executions; therefore, we admit all four executions shown above as possible executions of P.

Treatment of alternatives

Each alternative of a procedure is akin to a procedure. Associated with an alternative is a nonempty set of executions, and the executions of a procedure —assuming that all its alternatives are positive— is the union of the executions of its alternatives.

The treatment of negative alternatives is more involved and we have not developed the necessary theory. The following example illustrates how such a theory may be developed. Let procedure f have two alternatives:

$f::\ \ p;\ g\ \rightarrow\ s$
$\quad \not| \ q;\ h\ \rightarrow\ t$

Procedure f is replaced by two procedures, f^+ and f^-, corresponding to its accepting and rejecting executions. Procedure f^+ has the following execution (disregarding the *begin* and *end* symbols): $p\ g^+\ s$. Note that g^+ has to be used instead of g in this execution. An execution of f rejects because (1) both p and q are *false*, or (2) p holds but g rejects, or (3) q holds but h rejects, or (4) q holds and h accepts. The corresponding executions are $\neg p\ \wedge\ \neg q$, $p\ g^-$, $q\ h^-$, and $q\ h^+\ t$.

10.2.3 Partial order on boxes

We restate the following restriction on programs described in section 3.2.6. For each procedure, there is a partial order over the boxes of the program such that during execution of that procedure, a procedure may call another only if the former belongs to a higher box than the latter. Different procedures may impose different partial orders on the boxes. It was shown in section 3.4.4 why a static partial order —i.e., one that is the same for all procedures— is inadequate in practice.

Definition For procedures p and q, we write $p\ calls\ q$ to mean that p has q as a nonterminal. Let $calls^+$ be the transitive closure of $calls$, and $calls^*$ the reflexive transitive closure of $calls$. Define a relation $calls_p$ over procedures where

$$(x\ calls_p\ y)\ \equiv\ (p\ calls^*\ x) \wedge (x\ calls\ y) \qquad \square$$

In operational terms, $x\ calls_p\ y$ means procedure x may call procedure y in some execution of procedure p.

Each program is required to satisfy the following condition (PB). For every procedure p, there is a partial order $\geq_p$ over the boxes such that

$$x\ calls_p\ y\ \Rightarrow\ x.box >_p y.box \qquad \text{(PB)}$$

Note Let $b >_p c$ be $b \geq_p c\ \wedge\ b \neq c$. Relation $\geq_p$ is reflexive and $>_p$ is irreflexive. $\square$

Observation 1

$$p\ calls^*\ x\ \Rightarrow\ p.box \geq_p x.box$$
$$p\ calls^+\ x\ \Rightarrow\ p.box >_p x.box$$

Proof: Define $calls^i$, for $i \geq 0$, as follows.

$$p\ calls^0\ p\ ,$$
$$p\ calls^{i+1}\ q\ \equiv\ \langle \exists r ::\ p\ calls^i\ r\ \wedge\ r\ calls\ q \rangle$$

We prove the following results using induction over i,

$$p\ calls^0\ x\ \Rightarrow\ p.box \geq_p x.box$$
$$p\ calls^{i+1}\ x\ \Rightarrow\ p.box >_p x.box\ ,\ \text{for all } i,\ i \geq 0$$

For $i = 0$:

$$
\begin{array}{ll}
 & p\ calls^0\ x \\
\equiv & \{\text{definition of } calls^0\} \\
 & p = x \\
\Rightarrow & \{\text{predicate calculus}\} \\
 & p.box \geq_p x.box
\end{array}
$$

For $i + 1$, $i \geq 0$:

$$
\begin{array}{ll}
 & p\ calls^{i+1}\ x \\
\equiv & \{\text{definition of } calls^{i+1}\} \\
 & \langle \exists r ::\ p\ calls^i\ r\ \wedge\ r\ calls\ x \rangle \\
\equiv & \{\text{from the definition of } calls^*\text{: } p\ calls^i\ r\ \Rightarrow\ p\ calls^*\ r\} \\
 & \langle \exists r ::\ p\ calls^i\ r\ \wedge\ p\ calls^*\ r\ \wedge\ r\ calls\ x \rangle \\
\Rightarrow & \{\text{induction on } p\ calls^i\ r\} \\
 & \langle \exists r ::\ (p.box \geq_p r.box)\ \wedge\ (p\ calls^*\ r\ \wedge\ r\ calls\ x) \rangle \\
\equiv & \{\text{definition of } r\ calls_p\ x\} \\
 & \langle \exists r ::\ (p.box \geq_p r.box)\ \wedge\ r\ calls_p\ x \rangle \\
\Rightarrow & \{\text{partial order condition, (PB), on the second conjunct}\} \\
 & \langle \exists r ::\ (p.box \geq_p r.box)\ \wedge\ (r.box >_p x.box) \rangle \\
\Rightarrow & \{>_p \text{ is a partial order}\} \\
 & p.box >_p x.box
\end{array}
$$

The claims in observation 1 follow from

$$
\begin{array}{l}
p\ calls^*\ x\ \equiv\ \langle \exists i :\ i \geq 0 :\ p\ calls^i\ x \rangle \\
p\ calls^+\ x\ \equiv\ \langle \exists i :\ i > 0 :\ p\ calls^i\ x \rangle
\end{array}
$$

□

From observation 1,

$$p\ calls^+\ q\ \Rightarrow\ (p.box >_p q.box)$$

Therefore, p and q are in different boxes. It follows that in a tight execution no call is made on a box when one of its procedures has started but not completed its execution.

Observation 2 $calls^+$ is an acyclic (i.e., irreflexive, asymmetric, and transitive) relation over the procedures.
Proof: From its definition $calls^+$ is transitive. Also,

$$
\begin{array}{ll}
 & p\ calls^+\ p \\
\Rightarrow & \{\text{from observation 1}\} \\
 & p.box >_p p.box
\end{array}
$$

a contradiction. Therefore, $calls^+$ is irreflexive. Asymmetry of $calls^+$ follows similarly. □

10.2.4 Procedures as relations

With each terminal symbol we have associated a binary relation over program states. Next, we associate such a relation with each procedure and each execution of a procedure; to simplify notation, we use the same symbol for an execution (or a procedure) and its associated relation. For execution e, $(u, v) \in e$ means that if e is started in state u, it is possible for it to end in state v. For procedure p, $(u, v) \in p$ means that there is an execution e of p such that $(u, v) \in e$. Formally,

- The relation for a procedure is the union of relations of all its executions.
- The relation for an execution $x_0, \ldots, x_n$ is the relational product of the sequence of relations corresponding to the x_i's.

Observe that symbol x_i in an execution may be a terminal, so the relation for it has already been defined or a nonterminal for which the relation has to be computed recursively using this definition. We show in lemma 1 that the rules given above define unique relations for each execution and procedure; the key to the proof is the acyclicity of $calls^+$.

Definition The *height* of a procedure is a natural number; height is 0 if the procedure has no nonterminal. Otherwise,

$$p \; calls \; q \;\Rightarrow\; p.height > q.height$$

This definition of height is well-grounded because $calls^+$ induces an acyclic relation on the procedures. □

Lemma 1 There is a unique relation for each procedure and for each execution.

Proof: We prove the result by induction on n, the height of a procedure.

Case $n = 0$**:** The procedure has only terminals in all its executions. The relation associated with any execution of the procedure is the relational product of its terminals. The relation associated with the procedure is the union of all its executions; therefore, these relations are uniquely determined.

Case $n > 0$**:** The relations for the terminals in an execution are given. The nonterminals have heights smaller than n; therefore, from the induction hypothesis, relations for these nonterminals can be computed. Hence, the relation for an execution —which is the relational product of the sequence of relations of its terminals and nonterminals— can also be computed. The relation for a procedure is the union of the relations of its executions; therefore, it is uniquely determined. □

Note that an execution may have the empty relation associated with it, which denotes that the steps of the execution never appear contiguously in a program execution. Consider the following execution from the example on page 319: $begin_P\ readt\ \alpha\ readf\ \beta\ end_P$. Here two successive reads of the same variable yield different values. Such a sequence may appear only as a noncontiguous subsequence in a program execution where interleaved steps of another procedure's execution alter the value of the variable in between the two read operations.

Notation Henceforth, each symbol —terminal and nonterminal— has an associated binary relation over program states. Concatenation of symbols corresponds to their relational product. For strings x and y, we write $x \subseteq y$ to denote that the relation corresponding to x is a subset of the relation for y. □

Observation 3 For terminal symbols s and t of different boxes, $st = ts$ (i.e., the executions st and ts have the same effect on the program state). Also, if s and t are sequences of terminals, and all symbols in s and t are from different boxes, then $st = ts$. □

10.3 Compatibility

Procedures p and q are *compatible*, denoted by $p \sim q$, iff all of the following conditions hold. Observe that $\sim$ is a symmetric relation.

C0. $p\ calls\ p' \ \Rightarrow\ p' \sim q$ and $q\ calls\ q' \ \Rightarrow\ p \sim q'$.

C1. If p and q are in the same box,

$$(p \text{ is total } \Rightarrow\ qp \subseteq pq)$$
$$(q \text{ is total } \Rightarrow\ pq \subseteq qp)$$

C2. If p and q are in different boxes, the transitive closure of the relation $(\geq_p \cup \geq_q)$ is a partial order over the boxes.

Condition (C0) says that procedures that are called by compatible procedures are compatible. Condition (C1) says that for p and q in the same box, the effect of executing a partial procedure and then a total procedure can be simulated by executing them in the reverse order. Condition (C2) says that compatible procedures impose similar (i.e., nonconflicting) partial orders on boxes.

Notes

1. If partial procedures p and q of the same box call no other procedure, they are compatible because the given conditions hold vacuously.

2. Total procedures p and q of the same box are compatible only if $pq = qp$, applying (C1) twice.

3. Condition (C0) is well-grounded because if $p\ calls\ p'$, the height of p exceeds that of p'.

4. For procedures with parameters (as is the case in Seuss) compatibility has to be established by checking the given conditions, (C0, C1, C2), with all possible values of parameters; see the channels example in section 10.3.1. □

Automatic derivation of compatibility relation

Compatibility condition (C1) cannot be checked by a traditional compiler, because a proof is required to establish this condition (see examples in section 10.3.1, below, for such proofs). Therefore we expect a programmer to declare that certain pairs of procedures in each box are compatible, by proving the corresponding (C1) condition. Once the declarations are available, a compiler can generate the compatibility relation among all procedures (across different boxes), employing conditions (C0) and (C2), as follows.

First, the *calls* relation of the program ($p\ calls\ q$) is generated from the program text. Then $calls^*$ and $calls_p$ relations are computed (see section 10.2.3 for definitions of these relations). These determine the partial order over boxes, according to (PB) in that section. Hence, for each pair of procedures p and q, condition (C2) can be checked.

Next, (C0) can be checked as follows. For all pairs of procedures p and q, computation proceeds in the order of increasing sum of their heights. Since the compatibility information is available through declarations of procedures in the same box, assume that p and q belong to different boxes. If their heights are both 0, they call no other procedure; hence, (C0) is satisfied and they are compatible. Otherwise, let p and q have a combined height n, $n > 0$. The compatibility of p', q and p, q' —where $p\ calls\ p'$ and $q\ calls\ q'$— have already been determined because each pair has a combined height lower than n; hence, $p \sim q$ can be computed using (C0).

10.3.1 Examples of compatibility

Semaphore

Consider a general semaphore as given by the program *Semaphore* of section 3.2.3. We show that $V \sim V$ and $P \sim V$, i.e., from (C1),

$$\begin{aligned} VV &= VV \\ PV &\subseteq VP \end{aligned}$$

The first identity is trivial. For the second identity, we compute the relations corresponding to P and V as follows. Below, $\circ$ denotes relational product.

$$
\begin{array}{ll}
 & P \\
= & \{\text{from the program text}\} \\
 & \quad (n > 0) \ \circ \ (n := n - 1) \\
= & \{\text{definitions of predicate and assignment; see appendix A.5}\} \\
 & \quad \{(x, x) |\ x > 0\} \ \circ \ \{(x, x - 1) |\ x > 0\} \\
= & \{\text{simplify}\} \\
 & \quad \{(x, x - 1) |\ x > 0\}
\end{array}
$$

Similarly,

$$V = \{(x, x + 1) |\ x \geq 0\}$$

Taking their relational product,

$$PV = \{(x, x) |\ x > 0\}$$

and

$$VP = \{(x, x) |\ x \geq 0\}$$

Therefore, $PV \ \subseteq \ VP$.

For a binary semaphore (see section 4.9.1) P, V are not compatible. Here, the V operation fails if it is executed when the semaphore value is 1. We define a failed state ! from which all further executions remain in the same state. Then

$$P = \{(1, 0), (!, !)\}$$
$$V = \{(0, 1), (1, !), (!, !)\}$$

Taking their relational products

$$PV = \{(1, 1), (!, !)\}$$
$$VP = \{(0, 0), (1, !), (!, !)\}$$

This establishes $PV \ \not\subseteq \ VP$.

Channels

Consider the unbounded fifo channel of section 4.1.1. We show $get \sim put$; i.e., for any x and y,

$$get(x) \ put(y) \ \subseteq \ put(y) \ get(x)$$

Hence, any state reachable by executing $get(x) \ put(y)$ is also reachable by executing $put(y) \ get(x)$ starting from the same initial state.

Given $(u, v) \in get(x) \ put(y)$, we show that $(u, v) \in put(y) \ get(x)$. From $(u, v) \in get(x) \ put(y)$ and the definition of relational product, conclude

that there is a state w such that $(u, w) \in get(x)$ and $(w, v) \in put(y)$. Since $(u, w) \in get(x)$, from the implementation of *get*, the channel is nonempty in state u; i.e., the channel state s is of the form $a +\!\!+ S$ for some item a and a sequence of items S. Then, we have

$$\{s = a +\!\!+ S\}\ put(y)\ \{s = a +\!\!+ S +\!\!+ y\}\ get(x)\ \{x +\!\!+ s = a +\!\!+ S +\!\!+ y\}$$
$$\{s = a +\!\!+ S\}\ get(x)\ \{x +\!\!+ s = a +\!\!+ S\}\ put(y)\ \{x +\!\!+ s = a +\!\!+ S +\!\!+ y\}$$

The final states, given by the values of x and s, are identical. This completes the proof.

The preceding argument shows that two procedures from different boxes that call *put* and *get* (i.e., a sender and a receiver) may be executed concurrently. Further, since $get \sim get$ by definition, multiple receivers may also be executed concurrently. However, $put \sim put$ does not hold; for arbitrary x and y,

$$put(x)\ put(y) \neq put(y)\ put(x)$$

because a fifo channel is a sequence, and appending a pair of items in different orders results in different sequences. Since *put* is not compatible with *put*, multiple senders should not be executed concurrently.

In a client–server type of interaction, it is often required that multiple senders (clients) and receivers (servers) be executed concurrently, sending and receiving over a single channel. As shown above, a fifo channel is inadequate for this purpose. Therefore, we use the unordered channel of section 4.1.3. We show that $put \sim put$ and $put \sim get$ for the unordered channel; i.e., for all x and y

$$put(x)\ put(y)\ =\ put(y)\ put(x)$$
$$get(x)\ put(y)\ \subseteq\ put(y)\ get(x)$$

The proof of the first identity is trivial because *put* is implemented as a bag union. The proof of the second result is similar to that for the fifo channel. We need consider the initial states where the bag b is nonempty. In the following, $x \cup b$ is an abbreviation for $\{x\} \cup b$.

$$\{b = B, B \neq \emptyset\}\ get(x)\ \{x \cup b\ =\ B\}\ put(y)\ \{x \in B,\ x \cup b = B \cup y\} \quad (1)$$
$$\{b = B, B \neq \emptyset\}\ put(y)\ \{b = B \cup y\}\ get(x)\ \{x \in (B \cup y),\ x \cup b = B \cup y\} (2)$$

If the post-condition of (1) holds, the post-condition of (2) also holds because $x \in B \Rightarrow x \in (B \cup y)$. Hence, any final state of $get(x)\ put(y)$ is also a final state of $put(y)\ get(x)$.

10.3.2 Semicommutativity of compatible procedures

In lemma 2, below, we prove a result analogous to condition (C1) on page 323 for compatible procedures. This result applies to any pair of compatible procedures, not necessarily those in the same box.

Lemma 2 Let p be a total procedure and $p \sim q$. Then, $qp \subseteq pq$.

Proof: We apply induction on n, the sum of the heights of p and q. The result holds from the definition of $\sim$ if p and q are in the same box. Assume, therefore, that p and q are in different boxes.

Case $n = 0$**:** Both p and q are at height 0; hence, p and q have only terminals in all their executions. Since, p, q are from different boxes, the result follows by application of observation 3 (see page 323).

Case $n > 0$**:** From (C2), the transitive closure of $(\geq_p \cup \geq_q)$ is a partial order over the boxes; we abbreviate this relation by $\geq$. We prove the result for the case where $\neg(q.box > p.box)$. By symmetry, a similar argument applies for the remaining case, $\neg(p.box > q.box)$.

The proof is based on the following sublemmas. Sublemma 3 completes the proof.

> **sublemma 1:** For symbol x in any execution of p, $qx \subseteq xq$.
> **sublemma 2:** For any execution e of p, $qe \subseteq eq$.
> **sublemma 3:** $qp \subseteq pq$.

• Proof of sublemma 1: We consider two cases: x is a terminal, and x is a nonterminal.

- x is a terminal: Consider any expanded execution of q. A terminal t in this expanded execution is a symbol of procedure r where $q\ calls^*\ r$.

$$
\begin{array}{ll}
 & x.box = t.box \\
\Rightarrow & \{x \text{ and } t \text{ are terminals of } p \text{ and } r, \text{ respectively}\} \\
 & x.box = t.box \ \wedge\ x.box = p.box \ \wedge\ t.box = r.box \\
\Rightarrow & \{\text{predicate calculus}\} \\
 & p.box = r.box \\
\Rightarrow & \{q\ calls^*\ r; \text{ observation 1 on page 320}\} \\
 & p.box = r.box \ \wedge\ q.box \geq_q r.box \\
\Rightarrow & \{\text{predicate calculus}\} \\
 & q.box \geq_q p.box \\
\Rightarrow & \{\geq \text{ is the transitive closure of } (\geq_p \cup \geq_q)\} \\
 & q.box \geq p.box \\
\Rightarrow & \{\text{by assumption, } p \text{ and } q \text{ are from different boxes}\} \\
 & q.box > p.box \\
\Rightarrow & \{\text{assumption: } \neg(q.box > p.box)\} \\
 & false
\end{array}
$$

Thus, x and t belong to different boxes, so applying observation 3, $xt = tx$. Applying this argument for all terminals t in every expanded execution of q, we have $qx = xq$.

- x is a nonterminal: Since $p \sim q$ and $p\ calls\ x$, from (C0), $x \sim q$. The combined heights of x and q is less than n. Also, x is total, since it is a nonterminal of p and p is total. From the induction hypothesis, $qx \subseteq xq$. (End of sublemma 1 proof) □

• Proof of sublemma 2: We have to show that for any execution e of p, $qe \subseteq eq$. Proof is by induction on the length of e. If the length of e is 1, the result follows from $qx \subseteq xq$ (sublemma 1). For e of the form fx, where f is a string and x is a symbol:

$$\begin{array}{ll} & qfx \\ \subseteq & \{\text{Induction: } qf \subseteq fq\text{; monotonicity of relational product}\} \\ & fqx \\ \subseteq & \{\text{from sublemma 1, } qx \subseteq xq\text{; monotonicity of product}\} \\ & fxq \end{array}$$

(End of sublemma 2 proof) □

• Proof of sublemma 3, $qp \subseteq pq$:

$$\begin{array}{ll} & qp \\ = & \{\text{definition of } p\} \\ & q(\cup_{e \in p}\ e) \\ = & \{\text{distributivity of relational product over union}\} \\ & (\cup_{e \in p}\ qe) \\ \subseteq & \{\text{from sublemma 2, } qe \subseteq eq\text{; monotonicity of } \cup\} \\ & (\cup_{e \in p}\ eq) \\ = & \{\text{distributivity of relational product over union}\} \\ & (\cup_{e \in p}\ e)q \\ = & \{\text{definition of } p\} \\ & pq \end{array}$$

(End of sublemma 3 proof) □

Lemma 3 $(p \sim q\ \wedge\ p\ calls^*\ p'\ \wedge\ q\ calls^*\ q') \Rightarrow (p' \sim q')$.

Proof: First, prove

$$(p \sim q\ \wedge\ p\ calls^i\ p'\ \wedge\ q\ calls^j\ q') \Rightarrow (p' \sim q')$$

by induction on $i + j$, $i, j \geq 0$. Lemma 3 follows from this proposition. □

10.4 Loose Execution

A loose execution of a program allows concurrent executions of its procedures (i.e., in an interleaved fashion). Henceforth, we consider executions in which only a finite number of procedures are executed to completion; that is, the loose executions are finite. Concurrency in a loose execution is restricted as follows: (1) only compatible procedures may be executed

concurrently, and (2) two procedures from the same box are never executed concurrently. An implementation that obeys (1) and (2) is given in chapter 11.

Now we show how to eliminate condition (2) above, by imposing a simple requirement on the terminals of a box.

10.4.1 Box condition

The condition that at most one procedure from a box is executed at any time can be encoded in our model by making it impossible for procedure q to start if procedure p of the same box has started and not yet completed.

Definition Let σ be a sequence of terminals and nonterminals. Procedure p is *incomplete* after σ if σ contains fewer end_p's than $begin_p$'s. □

Box condition Let p and q be procedures of the same box, and p be incomplete after σ. Then $\sigma\, begin_q = \epsilon$, where ϵ denotes the empty relation. □

The following lemma shows that under certain conditions a terminal symbol can be transposed with an adjacent nonterminal.

Lemma 4 Let p and q be procedures, t a terminal of p, and σ any sequence of symbols.
1. If p is incomplete after σ, then $\sigma qt \subseteq \sigma tq$.
2. If p is incomplete after σt, then $\sigma tq \subseteq \sigma qt$.

Proof: We prove the first part; the other part is left to the reader.

$$
\begin{array}{ll}
 & \sigma qt \\
= & \{q \text{ is the union of all its expanded executions, } g\} \\
 & \langle \cup_g (\sigma g t) \rangle \\
= & \{\text{partition } g \text{ into } e \text{ and } f\text{: each execution in } e \\
 & \quad \text{has a terminal from } p.box, \text{ and } f \text{ does not}\} \\
 & \langle \cup_e (\sigma e t) \rangle \ \cup \ \langle \cup_f (\sigma f t) \rangle \\
= & \{e \text{ is of the form } \sigma'\, begin_r\, \sigma'', \text{ where:} \\
 & \quad \sigma' \text{ has no terminal from } p.box; \\
 & \quad r \text{ is some procedure from } p.box\} \\
 & \langle \cup (\sigma\sigma'\, begin_r\, \sigma''\, t) \rangle \ \cup \ \langle \cup_f (\sigma f t) \rangle \\
= & \{\sigma\sigma'\, begin_r = \epsilon, \text{ from Box condition, because} \\
 & \quad p \text{ is incomplete after } \sigma\text{; therefore, after } \sigma\sigma' \text{ too.} \\
 & \quad \text{Also, } r.box = p.box\} \\
 & \langle \cup_f (\sigma f t) \rangle \\
= & \{f \text{ has no terminal from } p.box,\ t \text{ is a terminal of } p.box; \\
 & \quad \text{apply observation 3 on page 323}\} \\
 & \langle \cup_f (\sigma t f) \rangle \\
\subseteq & \{f \text{ is a subset of the (expanded) executions of } q\} \\
 & \sigma tq
\end{array}
$$

10.4.2 Execution tree

Definition An *execution tree* of procedure p is an ordered tree such that (1) the root is labeled p, (2) every non-leaf node is labeled with a nonterminal symbol, and (3) the sequence of labels of the children of a non-leaf node q is an execution of q. A *full execution tree* is an execution tree in which each leaf node is labeled with a terminal symbol. □

An execution tree of procedure p is constructed by taking an execution of p and possibly expanding some of the nonterminals in a similar fashion; see the example given later in this section. Any execution tree is finite. This is because if procedure q is a parent of procedure r in a tree for p, then $q\ calls_p\ r$; from partial order on boxes (PB), $q.box >_p r.box$. Therefore, the boxes along a path are distinct. Since the program has a finite number of boxes, each path in the tree is finite. Also, the degree of each node is finite because each execution is finite in length. From Koenig's lemma, the tree is finite.

Definition The *frontier* of an execution tree is the ordered sequence of symbols in the leaf nodes of the tree. An *expanded execution* of procedure p is the frontier of some full execution tree of p. Hence, an expanded execution consists of terminals only. □

A loose execution is given by (1) a finite set of full execution trees (of some of the procedures), and (2) a finite sequence of terminals called a *run*. The run describes the steps (i.e., the terminals) of the interleaved execution. Each execution tree shows the entire history (of all procedures called) during the execution of a procedure in that run. The trees and the run satisfy conditions (M0, M1), given below.

Condition (M0) states that each symbol of the run can be uniquely identified with a leaf node of some tree and vice versa. Additionally, the run contains the procedure executions (the frontiers of the corresponding trees) as subsequences. Since each symbol of the run belongs to a unique tree we write $x.root$ for the root of the tree that symbol x belongs to.

Condition (M1) states that if two procedures are incomplete at any point in the run, they either belong to the same tree (i.e., they are both part of the execution of the same procedure) or they are compatible.

- (**M0**) There is a 1–1 correspondence between the symbols in the run and the leaf nodes of the trees. The subsequence of the run corresponding to symbols from a tree T is the frontier of T.

- (**M1**) Suppose procedure p is incomplete before symbol s in the run. Then either $(p.root = s.root)$ or $(p.root \sim s.root)$.

A *tight execution* is a loose execution in which (M1) is modified as follows: suppose procedure p is incomplete before symbol s in the run; then $p.root = s.root$. Therefore, there is no interleaving of procedure executions.

Example

We show a loose execution that is an interleaving of two procedure executions. The two execution trees are shown in Fig. 10.1. The nonterminals are given in uppercase, terminals in lowercase. We use $[P$ as an abbreviation for $begin_P$ and $P]$ for end_P. The following run prescribes the interleaving: $[P\ p\ [Q\ [T\ q\ r\ Q]\ [U\ [R\ [S\ s\ u\ S]\ U]\ R]\ P]\ T]$. Clearly, $P \sim T$ for this to be an acceptable loose execution. We can derive this result using (M1) at the point preceding s in the run: since P is incomplete there, either $(P.root = s.root)$ or $(P.root \sim s.root)$. From the execution trees, $P.root = P$ and $s.root = T$, so $(P.root \neq s.root)$. Therefore, $(P.root \sim s.root)$ or $P \sim T$.

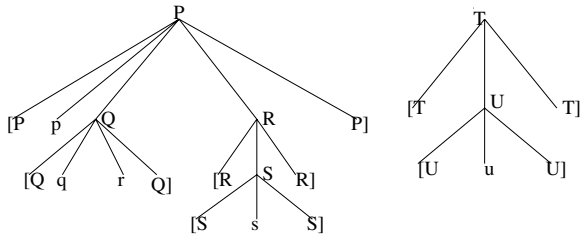


Figure 10.1: Execution trees in a loose execution

10.5 Reduction Theorem and Its Proof

Reduction theorem For any loose execution E there is a finite sequence of procedures F such that $E \subseteq F$ (here, F denotes the relational product of the corresponding procedures).

10.5.1 Proof of the reduction theorem

Suppose R is the run of some loose execution. We transform run R and the execution trees in stages; let R' denote the transformed run. The transformed run may consist of terminals as well as nonterminals, and its execution trees need not be full (i.e., leaf nodes may have nonterminal labels). We show how to transform the execution trees and the run so that the following invariants are maintained. Note the similarity of (N0, N1) with (M0, M1).

- (**N0**) There is a 1–1 correspondence between the symbols in the run and the leaf nodes of the trees. The subsequence of the run corresponding to symbols from a tree T is the frontier of T.

- (**N1**) Suppose procedure p is incomplete before symbol s in the run. Then either ($p.root = s.root$) or ($p.root \sim s.root$).

- (**N2**) $R \subseteq R'$.

Conditions (N0, N1, N2) are satisfied initially by the given run and the execution trees: (N0, N1) follow from (M0, M1), and (N2) holds because $R = R'$.

The reduction process terminates when there are no *end* symbols in the run; then all symbols are the roots of the trees. Hence, this run corresponds to a sequence of nonterminals, and according to (N2), it establishes the reduction theorem. The resulting sequence of nonterminals can simulate the original loose execution: if the original execution starting in state u can end in state v, there exists a tight execution of the sequence of nonterminals that transforms the program state from u to v.

For a run that contains an *end* symbol, we apply either a *replacement* step or a *transposition* step. Let the first *end* symbol appearing in the run belong to procedure q.

Replacement step If a contiguous subsequence of the run corresponds to the frontier of a subtree rooted at q (then the subsequence is an execution of q), replace the subsequence by symbol q and delete the subtree rooted at q (retaining q as a leaf node). □

This step preserves (N0). (N1) also holds because, for any symbol x in the execution that is replaced by q, $p.root \sim x.root$ holds prior to replacement, and $x.root = q.root$. Hence, $p.root \sim q.root$ after the replacement. The relation for a procedure is weaker than for any of its executions; therefore, the replacement step preserves (N2).

Transposition step If a run has an *end* symbol, and a replacement step is not applicable, then execution of some procedure q is noncontiguous. We then apply a transposition step to transpose two adjacent symbols in the run (leaving the execution trees unchanged) that makes the symbols of q more contiguous. Continued transposition make it possible to apply a replacement step eventually. □

Suppose q is a partial procedure (similar arguments apply to partial procedures that have no pre-procedures and to total procedures). An execution of q is of the form ($begin_q\ b\ h\ \ldots x \ldots\ end_q$), where h is the pre-procedure of q and x is either a terminal symbol or a nonterminal, designating a total procedure of q. All procedures that complete before q have already been

replaced by nonterminals, because the first *end* symbol appearing in the run belongs to q. Note that h is a procedure that completes before q.

Suppose x is preceded immediately by y in this run. Then, y is not part of the execution of q. We show that x and y can be transposed, bringing x closer to h. Transposing x and y preserves (N0, N1). We show below that transposition preserves (N2) as well.

Case 0 (both x and y are terminals): Let y be a terminal of procedure p. Procedure q is incomplete before y because its end_q symbol comes later. For p and q in the same box, the relation corresponding to prefix σ of the run up to (and including) y is ϵ, from the box condition. Hence, $\sigma yx = \epsilon \subseteq \sigma xy$. If p and q belong to different boxes, from observation 3, symbols x and y can be transposed.

Case 1 (both x and y are nonterminals): Symbol x is part of q's execution; therefore, $q.root\ calls^*\ x$. It is given that symbol y is not part of q's execution. Also, y cannot be part of the execution of any procedure that calls q because q is incomplete before y; therefore, $q.root \neq y.root$.

$$
\begin{array}{ll}
 & q \text{ is incomplete just before } y \\
\Rightarrow & \{\text{(N1)}\} \\
 & q.root = y.root \ \vee\ q.root \sim y.root \\
\Rightarrow & \{q.root \neq y.root \text{ (see explanation above)}\} \\
 & q.root \sim y.root \\
\Rightarrow & \{q.root\ calls^*\ x \ \wedge\ y.root\ calls^*\ y\text{; lemma 3}\} \\
 & x \sim y \\
\Rightarrow & \{x \text{ is total; lemma 2}\} \\
 & yx \subseteq xy
\end{array}
$$

Case 2 (x is a terminal, y a nonterminal): q is incomplete just before y. Applying lemma 4 (part 1), x and y may be transposed.

Case 3 (x is a nonterminal, y is a terminal): Let p be the procedure of which y is a terminal. Since the first *end* symbol in the run belongs to q, end_p comes after x. Therefore, p is incomplete before x. Applying lemma 4 (part 2) with p as the incomplete procedure, x and y may be transposed.

Thus, x and y may be transposed in all cases, preserving N3. Hence, all symbols in the execution of q to the right of h can be brought next to h.

Next, we bring the $begin_q$ symbol and the predicate b next to h using an argument similar to case 3. Thus, all of q's symbols to the left and right of h can be made contiguous around h, and a replacement step can then be applied.

For a total procedure q the reduction is done similarly; $begin_q$ serves the role of h in the above argument. For a procedure q that has no pre-procedure, the reduction process is similar with b serving the role of h.

Proof of termination of the reduction scheme:

We show that only a finite number of replacement and transposition steps can be applied to any loose execution. For a given run, consider the procedure q whose *end* symbol, end_q, is the first *end* symbol in the run. Define two quantities, n and c, for each run as follows.

n = the number of end symbols in the run
$c = \Sigma\ c_j$

where c_j is the number of symbols in the run not belonging to q between the pre-procedure h of q and symbol j of q, and the sum in c is over all symbols of q. Value of c is arbitrary if the run has no *end* symbol.

The pair (n, c) decreases lexicographically with each transposition and replacement step. This is because a replacement step removes one end symbol from the run, thus decreasing n. A transposition step decreases c while keeping n unchanged. Therefore, n will become 0; then the run has no *end* symbol, so, from (N0), the symbols are the roots of the execution trees.

10.6 A Variation of the Reduction Theorem

The following variation of the reduction theorem may be useful for applications in a wide-area network, such as the World Wide Web, because it requires no check for compatibility and no scheduler of any kind (see section 11.2 for the role of scheduler in the general case). Consider a Seuss program in which every procedure calls at most one other procedure. Then any pair of actions may be taken to be compatible, and any concurrent execution is a loose execution.

The proof of the reduction theorem in this case is similar to the earlier proof. If procedures p and q call p' and q', respectively, then the concurrent executions of p and q are serialized in the order in which p' and q' are called; the entire execution of p is made contiguous around p' and similarly for q. As before, we reduce the procedure whose end symbol is the first end symbol in the run. If this procedure calls no other procedure, all its symbols are terminals and, by applying case (0) and case (2) of the transposition step, we can bring all its symbols together next to its first symbol. If the procedure calls another procedure, according to the reduction scheme, the called procedure has already been reduced, and we bring all the symbols next to the called procedure symbol in a similar fashion.

The major simplification in the reduction scheme for this special case is due to the fact that it is never necessary to transpose two nonterminals. Therefore, case (1) of the transposition step never arises. Consequently, the condition for compatibility of two procedures (page 323) is irrelevant in this case.

10.7 Concluding Remarks

Concurrent programming is possible because most components do not interfere with each other. Large systems are easier to build out of non-interfering components, because each component can be designed independently. Whenever there is interference, a scheduler or some such controlling mechanism is required for arbitration. Typically, non-interference is taken to mean that the components work on different parts of the data space. In this chapter, we have given a more general definition using compatibility.

How general is the notion of compatibility? We have shown that senders and receivers that communicate over point-to-point fifo channels are compatible; see page 325. This means that if all actions in a program employ only point-to-point channels for communication, then the actions can be executed concurrently; all pairs of actions are compatible and there is no need for a scheduler. If there are multiple senders on a fifo channel, the senders are *not* compatible. In this case, we have to use unordered channels to regain compatibility.

We surmise that message-based communication is popular in practice because it permits unrestrained concurrent execution. Contrast this with a shared variable program (reads and writes on shared variables are not compatible) where the actions have to be scheduled carefully to achieve the appropriate loose executions. The definition of compatibility proposed in this chapter may be used to define a "generalized channel", a shared variable for which its appropriate methods are compatible. Actions that access only these generalized channels are compatible and may be executed concurrently without intervention by a scheduler.

One of the distinctions between partial and total procedures is motivated by the requirements of the reduction theorem. It is possible to treat a partial procedure as a total procedure where a return code —"accept" or "reject"— notifies the caller of the status of procedure execution. If all procedures are total, compatibility reduces to commutativity: two procedures from the same box are compatible if they commute, and procedures from different boxes are compatible if the procedures they call are compatible. Unfortunately, commutativity is a strong requirement that is rarely met in practice; P and V on a semaphore do not commute, nor do *put* and *get* on a channel. Having partial procedures permits a richer class of concurrent executions, because a pair of partial procedures that call no other procedures are always compatible, and a partial procedure need only semicommute with a total procedure.

What the reduction theorem does not say

The reduction theorem merely says that a finite loose execution is equivalent to sequential executions of some of the actions. The theorem does not imply a similar result for an *infinite* loose execution (which can be de-

fined analogously to its finite counterpart). Consequently, no assertions can be made about the progress properties of loose executions; Adams [5] has derived a number of results for progress properties under more restrictive definitions of compatibility.

A reasonable additional restriction on loose executions ensures that invariant and stable properties and some progress properties of tight executions hold in loose executions as well. Valiant [166] has introduced the concept of *bulk synchrony*. Applied to our model, it means that every loose execution eventually reaches a point where there are no incomplete actions. This requirement can be enforced by a scheduler —see chapter 11— that periodically stops scheduling any action until all executing actions complete. Let a *milestone* be a point in a loose execution where there are no incomplete actions. Every infinite loose execution has an infinite number of milestones; henceforth, consider only infinite loose executions. It can be shown that every invariant of a tight execution holds at each milestone, and if p is stable in tight executions and p holds at some milestone in a loose execution, then p holds at all subsequent milestones. A progress property $p \mapsto q$ of tight executions, where q is stable, also holds in a loose execution in the following sense: if p holds at a milestone, q holds at that or some subsequent milestone.

10.8 Bibliographic Notes

The notion of reduction can be traced back to Church and Rosser (see [38]) in connection with reduction of terms in λ-calculus. Reduction of concurrent executions to sequential ones was pioneered by Lipton [123]. Lipton develops certain conditions under which the steps of a component may be considered indivisible (i.e., occurring sequentially) in a concurrent execution. A step f in a component is a *right mover* if for any step h of another component, whenever fh is defined, then so is hf and they yield the same result (i.e., their executions result in the same final state). Similarly, g is a *left mover* if for any h of another component hg is defined implies gh is defined, and $hg = gh$. Lipton shows that a sequence of steps of a component, $r_0\ r_1\ \ldots\ r_n\ c\ l_0\ l_1\ \ldots\ l_m$, may be considered indivisible if each r_i is a right mover, l_j a left mover and c is unconstrained. This characterization permits proofs of absence of deadlock. Doeppner [65], Back [13], and Lamport and Schneider [120] generalize this work to allow proofs of general safety properties. Cohen [43] gives a beautiful treatment of several reduction theorems, using Kleene algebra to compare their relative merits.

A particularly important application of reduction is in two-phase locking in database transactions [67]. Each transaction acquires all its locks for the items that it accesses before it unlocks any item. It can be shown that interleaved executions of such transactions is equivalent to executing them

sequentially in some order. In this case, lock acquisition is a right mover and unlock is a left mover.

For reductions that deal with progress properties, see Misra [137] and Cohen and Lamport [44]. Cohen [42] has introduced the notion of "decoupling" of program components, which permits them to be executed concurrently. Cohen and later Rao [155] establish a number of important results, including the connection between commutativity and decoupling. Joshi [99] proposes a concept called "immediacy", based on the earlier work of Cohen [42]. It combines the underlying assumptions of fairness and commutativity, and permits certain sequence of actions (such as sending and receiving on a channel) to be considered simultaneous; hence, a simple rendezvous can replace asynchronous communication. Immediacy properties obey many algebraic laws, and may be used as a basis for developing a Hoare-like logic for progress proofs. Cohen [43] has shown how similar proofs can be carried out within Kleene algebra.

The work in this chapter is developed to a far greater depth in the Ph.D. thesis of Adams [5]. He defines weak and strong compatibility, which are appropriate for studying safety and progress properties, respectively. He also identifies intermediate classes of compatibility suitable for studying specific progress properties. The notion of bulk synchrony is due to Valiant [166]. The original model of bulk synchrony was applicable to systems in which message-passing was the primary means of communication; see Ramachandran, Grayson, and Dahlin [154] for a comparative study of three bulk synchrony models, one of which deals with shared-memory interactions.

11
Distributed Implementation

11.1 Introduction

In this chapter, we develop algorithms for implementing Seuss programs on multiprocessor architectures in which the processors communicate using messages. The implementation strategy is to partition the boxes over a set of processors and have a scheduler that instructs each processor which action to execute next. The scheduler can be centralized or distributed among the processors. In the next section, we describe the scheduler in abstract terms that permits either type of implementation; specific implementations are described in section 11.5.

The role of a scheduler is to orchestrate concurrent executions of actions. Crucial to concurrent execution is the notion of *compatibility*, defined in section 10.3: if a set of actions are pairwise compatible, their executions are non-interfering, and their concurrent execution is equivalent to some serial executions of these actions. The precise definition of compatibility and the central theorem that establishes the correspondence between concurrent and serial executions are given in chapter 10. In this chapter, we assume that the compatibility relation among the actions is given, and we devise a scheduling strategy that allows concurrent executions of compatible actions. Additionally, we guarantee fairness in execution: each action of each box is eventually executed.

The scheduler is completely unnecessary if all pairs of actions in a program are compatible. Such is the case, for instance, if communication is through fifo channels, and each channel has a single sender (see page 325).

Also, if each action calls at most one procedure (see section 10.6), all pairs of actions are compatible, and the scheduler can be eliminated. If only a few pairs of actions are incompatible, a distributed scheduler can be used that lets the processors run autonomously for executions of most actions, and the processors consult the scheduler only when there is possibility of concurrent executions of incompatible actions.

Overview of the chapter

We give a brief overview of the implementation strategy in section 11.2. In section 11.3, the scheduling problem is specified and an abstract scheduler designed. The scheduler is proved to be maximal in section 11.4. We refine the maximal scheduler in section 11.5 for centralized and distributed implementations. Section 11.6 shows how processors actually execute the actions. Section 11.7 contains strategies for optimizations that let a scheduler predict that certain action executions will be rejected; hence, it does not ask the processors to execute them. A tight execution often involves busy waiting —a rejected call is typically attempted over and over— which can mostly be avoided using such optimizations.

11.2 Outline of the Implementation Strategy

The implementation consists of (1) a *scheduler* that decides which action may next be scheduled for execution (see section 11.3), and (2) *processors* that carry out the actual executions of the actions (see section 11.6). The boxes of a program are partitioned among the processors. Each processor, thus, manages a set of boxes and is responsible for the executions of the actions of those boxes. The boxes may be partitioned in any manner over the processors. An implementation may choose a partition to optimize a specific performance measure, such as workload at the processors or the network traffic. It should be understood that the scheduler and the processors are logical entities in our discussion; they may be implemented on one or more physical processors. The outline of the implementation is as follows.

- The scheduler repeatedly chooses some action for execution. The choice is constrained by the requirement that only compatible procedures may be executed concurrently and by the fairness requirement. The scheduler sends a message to the corresponding processor to start execution of this action.

- A processor starts executing an action on receiving a message from the scheduler. It may call on methods of the other processors by sending messages and waiting for responses. Each call includes argument values, if any, as part of the message. It is guaranteed that each call

elicits a response, which is either *accept* or *reject*. The accept response is sent when the call is accepted (which is always the case for calls on total methods), and values of result-parameters, if any, are returned with the response. A reject response is possible for calls only on partial methods; no parameter values accompany such a response.

11.3 Design of the Scheduler

11.3.1 An abstraction of the scheduling problem

We are given a finite number of actions and a *compatibility* relation among the actions. Two actions may be concurrently executed provided that they are compatible. It is given that each executing action terminates eventually. The goal is to design a task scheduler that repeatedly selects actions for execution so that (1) only compatible actions are executed concurrently, and (2) each action is executed infinitely often.

The following abstraction captures the essence of the scheduling problem. Given is a finite undirected graph in which there are no self-loops; the graph need not be connected. Each node in the graph is *black* or *white*; all nodes are initially white. In this abstraction, a node denotes an action and a black node an executing action. Two nodes are neighbors if they are *incompatible*, i.e., not compatible. It is given that every black node eventually becomes white, i.e., each action execution terminates. It is required to devise a coloring (scheduling) strategy so that the following conditions hold.

- No two neighbors are simultaneously black (i.e., only compatible actions may be executed simultaneously).
- Every node becomes black infinitely often.

Note that the scheduler can only blacken nodes; it may not whiten a node.

A simple scheduling strategy is to blacken a single node, wait until it is whitened, and then blacken another node. Such a strategy implements the first requirement trivially because there is at most one black node at any time. The second requirement may be ensured by blackening the nodes in some fixed order, say, round-robin. However, such a protocol defeats the goal of concurrent execution. So we impose the additional requirement that the scheduling strategy be maximal (see chapter 7): any valid concurrent executions of the actions is a possible execution of the scheduler. A maximal scheduler is a most general scheduler, because any execution of another scheduler is a possible execution of the maximal scheduler. By suitable refinement of the maximal scheduler, we derive a centralized scheduler and a distributed scheduler (section 11.5).

11.3.2 Specification

The specification of the scheduler is written using a variable b, which is the set of black nodes at any stage in the execution.

For sets x, y and a node v, $x = y + v$ denotes that $v \notin y \ \wedge x = y \cup \{v\}$.

Specification *Scheduler*
initially $b = \emptyset$ (S0)
$\langle \forall u, v : \ u \ neighbor \ v : \ \neg(u \in b \ \wedge \ v \in b) \rangle$ (S1)
$b = B$ **co** $b = B \ \vee \ \langle \exists v :: \ b = B + v \ \vee \ B = b + v \rangle$, for any B (S2)
For all v, $true \ \mapsto v \in b$ and $true \ \mapsto v \notin b$ (S3)
end {*Scheduler*}

Condition (S0) states that no action is executing initially; (S1) states that neighbors are never simultaneously black; (S2) says that in a step at most one node changes color. In (S3), the second property, $true \ \mapsto v \notin b$, is established by the actions themselves (each action terminates, so eventually becomes white), and the scheduler has to implement the remaining progress property, $true \ \mapsto v \in b$.

11.3.3 A scheduling strategy

The following strategy is reminiscent of the strategy used to implement a fair unordered channel; see section 7.5.1. Assign a natural number, called *height*, to each node; let $H[u]$ denote the height of node u. Predicate $u.low$ holds if the height of u is smaller than all of its neighbors; i.e.,

$$u.low \ \equiv \ \langle \forall v : \ u \ neighbor \ v : \ H[u] < H[v] \rangle$$

The scheduling strategy is to set b to $\emptyset$ initially, and the node heights in such a way that neighbors have different heights. Then, the following steps are repeated.

- (Blackening rule) Eventually consider each node v for blackening; if $v \notin b \wedge v.low$ holds, then blacken v.
- (Whitening rule) Simultaneous with the whitening of a node v, increase $H[v]$ to a value that differs from $H[u]$, for all neighbors u of v.

Formally, the coloring strategy is described by the following program. There is an action $add(v)$ for each node v that adds v to b provided that $v \notin b \wedge v.low$. The termination of action v is simulated by $remove(v)$, which removes v from b and increases $H[v]$ to a value that differs from $H[u]$ for all neighbors u of v.

---

program *Scheduler*
  node u, v;
  set(node) $b = \emptyset$;
  array[node](nat) H
    {**initially** $\langle \forall u, v : \ u \ neighbor \ v : \ H[u] \neq H[v] \rangle$};

  $\langle \forall v ::$
    $add(v) :: \ v \notin b \ \wedge \ v.low \ \rightarrow \ b := b \cup \{v\}$

    $remove(v) :: \ v \in b \ \rightarrow \ b := b - \{v\}$;
     $H[v] :=?$ **st**
      $H[v] > {}'H[v] \ \wedge \ \langle \forall u : \ u \ neighbor \ v : \ H[u] \neq H[v] \rangle$
  $\rangle$
end {*Scheduler*}

Note $'H[v]$ is the value of $H[v]$ before the assignment. □

11.3.4 Correctness of the scheduling strategy

We show that neighbors have different heights at all times, i.e.,

invariant $\langle \forall : x, y : \ x \ neighbor \ y : \ H[x] \neq H[y] \rangle$. (P0)

Proposition (P0) holds initially. If (P0) holds prior to execution of $add(v)$, it holds following the execution, because $add(v)$ does not affect heights. If (P0) holds prior to execution of $remove(v)$, it holds afterward because only $H[v]$ changes, and $H[v] \neq H[u]$, for any neighbor u of v following $remove(v)$.

Proof of (S0)

Follows from the initialization.

Proof of (S1)

The coloring strategy described above maintains the following invariant: for all v, $v \in b \ \Rightarrow \ v.low$. Observe that this proposition holds initially since all nodes are then white. A blackening step (add) preserves the proposition because $v.low$ is a pre-condition for blackening. A whitening step ($remove$) preserves the proposition because the antecedent of the proposition becomes false.

From this invariant, if u and v are both black, they are both low, and, from the definition of low, u and v are not neighbors. Therefore, neighbors are not simultaneously black.

Proof of (S2)

In $add(v)$, the assignment $b := b \cup \{v\}$ has the pre-condition $v \notin b$. In $remove(v)$, the assignment $b := b - \{v\}$ has the pre-condition $v \in b$. Hence, S2 is satisfied.

Proof of (S3)

We show that every node becomes black infinitely often in every execution. First, we give an informal operational argument, and later, we give a proof in the style of chapter 6.

Suppose that there is a node x that becomes black only a finite number of times in a given execution. Each blackening and the subsequent whitening increases the height of a node. Therefore, if some neighbor y of x becomes black infinitely often, its height eventually exceeds $H[x]$, establishing $\neg y.low$, and y is never blackened subsequently. Hence, every neighbor of x is blackened finitely often.

Applying this argument repeatedly, no node connected to x by a path can become black infinitely often. Therefore, beyond some point q in an execution, all nodes in the component of the graph to which x belongs remain white forever. Let v be a node with the smallest height in this component at point q in the execution; since all nodes remain white beyond q, their heights do not change and v remains a node with the smallest height. Whenever v is considered for blackening beyond q, v is white and $v.low$ holds; therefore, v will be blackened, contradicting the conclusion that v remains white forever beyond q.

Now we give the outline of a formal proof for $true \mapsto x \in b$ for all x. Define the *relative height* of node x, $x.rh$, to be the sum of the height differences of x and all its neighbors of lower heights, i.e.,

$$x.rh \;=\; \langle +y :\; x \; neighbor \; y \;\wedge\; H[x] > H[y] :\; H[x] - H[y] \rangle$$

We assert the following properties without proof; they can be proved directly from the program text; each $\mapsto$ property is indeed an *ensures* property.

For all x, y, n,

1. $x.low \;\mapsto\; x \in b$
2. $x.rh = n \;\wedge\; (x \; neighbor \; y) \;\wedge\; y.low$
   $\mapsto\; (x.rh = n) \;\wedge\; (x \; neighbor \; y) \;\wedge\; y \in b$
3. $x.rh = n \;\wedge\; (x \; neighbor \; y) \;\wedge\; y \in b \;\mapsto\; x.rh < n$

We give informal arguments for the validity of these three properties.

Validity of (1): the height of a node does not change as long as it remains white. Therefore, if x is low and white, it remains low (because its neighbors' heights can only increase) and white, until it is blackened. Eventually, x is considered for blackening and then blackened, establishing property (1).

Validity of (2) is similar: node y of the lowest height among the neighbors of x will eventually be black, and until then $x.rh$ is unchanged.

Property (3) follows from the observation that node y, as described above, will eventually become white; then $x.rh$ will be decreased because the height of y is increased.

We prove $true \mapsto x \in b$, based on (1,2,3) as follows.

• Proof of $true \mapsto x \in b$:

$x.rh = n \ \wedge \ (x \ neighbor \ y) \ \wedge \ y.low$
$\quad \mapsto \ (x.rh = n) \ \wedge \ (x \ neighbor \ y) \ \wedge \ y \in b$
, from (2)
$x.rh = n \ \wedge \ (x \ neighbor \ y) \ \wedge \ y.low \ \mapsto \ x.rh < n$
, transitivity with (3)
$x.rh = n \ \wedge \ \langle \exists y :: \ (x \ neighbor \ y) \ \wedge \ y.low \rangle \ \mapsto \ x.rh < n$
, disjunction over all y
$x.rh = n \ \wedge \ \neg x.low \ \mapsto \ x.rh < n$
, use invariant (P0), definition of *low*
$x.rh = n \ \wedge \ x.low \ \mapsto \ x \in b$, strengthen left side of (1)
$x.rh = n \ \mapsto \ x.rh < n \ \vee \ x \in b$
, disjunction of the above two
$true \ \mapsto \ x \in b$, induction on the above

11.4 Proof of Maximality of the *Scheduler*

We use the techniques of chapter 7 to prove the maximality of the *Scheduler*. Let z be a sequence of sets, which denotes a possible sequence of values of b in an execution; assume that z is stutter-free, i.e., successive values in z are distinct. Let z satisfy the specification (S0, S1, S2, S3), that is, the following properties, (S0′, S1′, S2′, S3′), hold.

$z_0 = \emptyset$ (S0′)
For all i, $\langle \forall u, v : \ u \ neighbor \ v : \ \neg(u \in z_i \ \wedge \ v \in z_i) \rangle$ (S1′)
For all i, $\langle \exists v :: \ z_{i+1} = z_i + v \ \vee \ z_i = z_{i+1} + v \rangle$ (S2′)
For all v,
$\quad \langle \forall i :: \ (\exists j : \ i \leq j : \ v \in z_j) \rangle$ and $\langle \forall i :: \ (\exists j : \ i \leq j : \ v \notin z_j) \rangle$ (S3′)

We create the following constrained program that includes a variable t, which denotes the current point of computation. Variable $u.next$ is an abbreviation for the next value j above t, where u is in z_j. Formally,

$$u.next = \langle min \ j : j > t \ \wedge \ u \in z_j : j \rangle$$

Note that $u.next$ is always defined, on account of (S3′).

---

program *Scheduler′*
  node u, v;
  set(node) $b = \emptyset$;
  integer $t = 0$;
  array[node](nat) H {**initially** $\langle \forall v :: H[v] = v.next \rangle$};

  $\langle \forall v ::$
    $add'(v) :: z_{t+1} = z_t + v \ \rightarrow$
      $v \notin b \ \wedge \ v.low \ \rightarrow \ b := b \cup \{v\};\ t := t + 1$

    $remove'(v) :: z_t = z_{t+1} + v \ \rightarrow$
      $v \in b \ \rightarrow \ b := b - \{v\};\ H[v] := v.next;\ t := t + 1$
  $\rangle$
end {*Scheduler′*}

11.4.1 Invariants of the constrained program

The following invariants hold for *Scheduler′*. For all nodes v,

$$b \ = \ z_t \tag{P1}$$
$$z_{vh} \ = \ z_{vh-1} + v \quad \text{where } vh \text{ is } H[v] \tag{P2}$$
$$\langle \forall u, v : \ u \ neighbor \ v : \ H[u] \neq H[v] \rangle \tag{P3}$$
$$(v.next \geq H[v]) \ \wedge \ (v.next > t) \tag{P4}$$
$$(H[v] = v.next) \ \equiv \ (v \notin b) \tag{P5}$$

Proof of (P1)

Initially, $b = \emptyset$ and $t = 0$, and from (S0′), $z_0 = \emptyset$. Stability of $b = z_t$ follows from the text of *Scheduler′*.

Proof of (P2)

This follows from the text of *Scheduler′* and (S2′).

Proof of (P3)

This property is similar to invariant (P0) proved for *Scheduler*. However, first we have to show that the random assignment is correctly implemented before we can assert that this property is inherited by *Scheduler′*. Let uh and vh be abbreviations for $H[u]$ and $H[v]$, respectively. We show below that $(uh = vh) \ \Rightarrow \ (u = v)$.

$$\begin{array}{ll} & uh = vh \\ \Rightarrow & \{\text{apply (P2) for nodes } u \text{ and } v\} \\ & (uh = vh) \ \wedge \ z_{vh} = z_{vh-1} + v \ \wedge \ z_{uh} = z_{uh-1} + u \\ \Rightarrow & \{\text{replace } uh \text{ by } vh \text{ in the second disjunct}\} \end{array}$$

$$
\begin{array}{ll}
 & z_{uh} = z_{uh-1} + v \ \wedge \ z_{uh} = z_{uh-1} + u \\
\Rightarrow & \{\text{Set theory}\} \\
 & u = v
\end{array}
$$

Thus, for distinct nodes u and v, $H[u] \neq H[v]$. Hence, the same result applies for neighbors u and v.

Proof of (P4)

To see the first conjunct, note that

initially $\langle \forall v :: \ H[v] = v.next \rangle$.

The only assignment to $H[v]$ is in $remove'(v)$:

$H[v] := v.next;$

so $v.next \geq H[v]$ is preserved by this assignment. Also, $v.next$ is monotonic in t; therefore $v.next$ never decreases in *Scheduler*$'$ because t never decreases.

The second conjunct follows from the definition of $v.next$.

Proof of (P5)

Initially (P5) holds because b is $\emptyset$ and $\langle \forall v :: \ H[v] = v.next \rangle$.

Now we show that (P5) is preserved by the execution of $add'(v)$; it can be shown that (P5) is unaffected by the execution of $add'(u)$, $v \neq u$. Define

$v.next.i = \langle min \ j : j > i \ \wedge \ v \in z_j : j \rangle$

Thus, $v.next = v.next.t$. Rewrite condition (P5) as

$(H[v] = v.next.t) \ \equiv \ (v \notin b)$

This holds as a post-condition of the assignments

$b := b \cup \{v\}; \ t := t + 1$

provided that $H[v] \neq v.next.(t+1)$ holds as a pre-condition, from the axiom of assignment (see section A.4.1). We show below that the pre-condition of $add'(v)$, $z_{t+1} = z_t + v \ \wedge \ v \notin b \ \wedge \ v.low$, and condition (P5) together imply that $H[v] \neq v.next.(t+1)$.

$$
\begin{array}{ll}
 & z_{t+1} = z_t + v \ \wedge \ v \notin b \\
\Rightarrow & \{\text{from definition of } v.next, \ (z_{t+1} = z_t + v) \ \Rightarrow \ (v.next = t + 1)\} \\
 & v.next \ = \ t + 1 \ \wedge \ v \notin b \\
\Rightarrow & \{\text{(P5): } (H[v] = v.next) \ \equiv \ (v \notin b)\} \\
 & H[v] = t + 1 \\
\Rightarrow & \{\text{from definition, } v.next.(t+1) > t + 1\} \\
 & H[v] \ \neq \ v.next.(t+1)
\end{array}
$$

Observe from the text of $remove'(v)$ that $(v \notin b) \ \wedge \ (H[v] = v.next)$ is established; hence, (P5) is preserved.

Rewriting the guard of $add'(v)$

We show from the given invariants that the augmenting guard of $add'(v)$, $z_{t+1} = z_t + v$, implies the original guard, $v \notin b \wedge v.low$. Hence, the original guard may be dropped in the constrained program. This result is needed for the proof of progress in chronicle correspondence; see section 11.4.3.

From $b = z_t$ (see P1) and $z_{t+1} = z_t + v$, we have $v \notin b$. We show that $v.low$ holds; i.e., for neighboring nodes u and v, $H[v] < H[u]$.

$$
\begin{array}{ll}
 & z_{t+1} = z_t + v \\
\Rightarrow & \{b = z_t \text{ from (P1)}\} \\
 & v \notin b \ \wedge \ v \notin z_t \ \wedge \ v \in z_{t+1} \\
\Rightarrow & \{\text{definition of } v.next\} \\
 & v \notin b \ \wedge \ v.next = t+1 \ \wedge \ v \notin z_t \ \wedge \ v \in z_{t+1} \\
\Rightarrow & \{\text{from (P5), } (H[v] = v.next) \ \equiv \ (v \notin b)\} \\
 & H[v] = t+1 \ \wedge \ v \notin z_t \ \wedge \ v \in z_{t+1} \\
\Rightarrow & \{\text{given } u, v \text{ are neighbors, } v \in z_{t+1} \ \Rightarrow \ u \notin z_{t+1}, \text{ from (S1}')\} \\
 & H[v] = t+1 \ \wedge \ v \notin z_t \ \wedge \ v \in z_{t+1} \ \wedge \ u \notin z_{t+1} \\
\Rightarrow & \{\text{given } v \notin z_t \ \wedge \ v \in z_{t+1} \ \wedge \ u \notin z_{t+1}, \text{ from (S2}') \ u \notin z_t\} \\
 & H[v] = t+1 \ \wedge \ v \notin z_t \ \wedge \ v \in z_{t+1} \ \wedge \ u \notin z_t \ \wedge \ u \notin z_{t+1} \\
\Rightarrow & \{\text{from (P1): } b = z_t; \\
 & \quad \text{from (P5): } (H[u] = u.next) \ \equiv \ (u \notin b); \\
 & \quad \text{from (P4): } u.next > t\} \\
 & H[v] = t+1 \ \wedge \ H[u] = u.next \ \wedge \ u.next > t \\
\Rightarrow & \{H[v] = t+1 \ \wedge \ H[u] > t. \text{ Apply (P3)}\} \\
 & H[v] < H[u]
\end{array}
$$

11.4.2 Correctness of random assignment implementation

The random assignment

$$H[v] :=? \ \textbf{st} \ \ H[v] > {}'H[v] \ \wedge \ \langle \forall u : \ u \ neighbor \ v : H[u] \neq H[v] \rangle$$

is implemented in the constrained program by

$$H[v] := v.next$$

The pre-condition of the assignment, $z_t = z_{t+1} + v$, and (from P1) $b = z_t$; these predicates together imply that $v \in b$. Hence, from (P4) and (P5), $H[v] < v.next$ holds prior to the assignment. Now, $H[v] = v.next$ holds after the assignment, thus establishing $H[v] > {}'H[v]$. The condition

$$\langle \forall u : \ u \ neighbor \ v : H[u] \neq H[v] \rangle$$

follows from (P3).

11.4.3 Proof of chronicle and execution correspondence

Proof of chronicle correspondence

1. (Safety) $b = z_t$ follows from (P1).
2. (Progress) $t = N \mapsto t = N+1$, for any natural N: exactly one guard of *Scheduler′* holds at any stage in the computation, because the guards are disjoint and their disjunction is *true*. Effective execution of either action increments t.

Proof of execution correspondence

1. (Safety) Guards of all the actions are disjoint.
2. (Progress) We have to show

$$\begin{array}{lll} \mathit{true} & \mapsto & z_{t+1} = z_t + v \\ \mathit{true} & \mapsto & z_t = z_{t+1} + v \end{array}$$

We sketch a proof. From (S3′) we deduce that

$$\begin{array}{l} \langle \forall i :: \ (\exists j : \ i \leq j : \ z_{j+1} = z_j + v) \rangle \\ \langle \forall i :: \ (\exists j : \ i \leq j : \ z_j = z_{j+1} + v) \rangle \end{array}$$

From the progress condition in chronicle correspondence of this section, t assumes values of successive natural numbers. Therefore, eventually $z_{t+1} = z_t + v$ and also eventually $z_t = z_{t+1} + v$.

11.5 Refining the Scheduling Strategy

We consider the situation where each action (node) is executed on a separate processor. First, we show how a centralized scheduler may schedule the actions given the compatibility relation. Next, we show how the scheduling may be distributed over the processors.

11.5.1 Centralized scheduler

A centralized scheduler maintains a list of nodes and their current colors and heights. Periodically, it scans through the nodes and blackens a node v provided that $v.low \wedge v \notin b$ holds. Whenever it blackens a node it sends a message to the appropriate processor specifying that the selected action may be executed. Upon termination of the action, the processor sends a message to the scheduler; the scheduler whitens the corresponding node and increases its height, ensuring that no two neighbors have the same

height. The scheduler may scan the nodes in any order, but every node must be considered eventually.

This implementation may be improved by having a set L of nodes that are both white and low; i.e., L contains all nodes v for which $v \notin b \wedge v.low$ holds. The scheduler blackens a node of L and removes it from L. Whenever a node x is whitened and its height increased, the scheduler checks x and all its neighbors to determine if any of these nodes qualify for inclusion in L; if some node y qualifies, y is added to L. It has to be guaranteed that every node in L is eventually scanned and removed; one possibility is to keep L as a queue in which additions are made at the rear and deletions from the front. Observe that once a node is in L it remains white and low until it is blackened.

11.5.2 Distributed scheduler

The proposed scheduling strategy can be distributed so that each node eventually blackens itself if it is white and low. The nodes communicate by messages of a special form, called *tokens*. Associated with each edge (x, y) is a token that is held by either x or y, whichever has the smaller height. Each token has a *value*, a positive integer equal to $H[y] - H[x]$ when the token for edge (x, y) is held by x.

It follows that a node that holds all incident tokens has height smaller than all of its neighbors; if such a node is white, it may color itself black. A node, upon becoming white, increases its height by a positive amount d, effectively reducing the value of each incident token by d (note that such a node holds all its incident tokens, so it can alter their values). The quantity d should be different from all token values so that neighbors do not have the same height, i.e., no token value becomes zero after a node's height is increased. If the value of token (x, y) becomes negative as a result of reducing it by d, which indicates that the holder x now has greater height than y, x resets the token value to its absolute value and sends the token to y.

Observe that the nodes need not query each other for their heights, because a token is eventually sent to a node of a lower height. Also, since the token value is the difference in heights between neighbors, it is possible to bound the token values, whereas the node heights are unbounded over the course of a computation. Initially, token values have to be computed and the tokens have to be placed appropriately based on the heights of the nodes. There is no need to keep the node heights explicitly from then on.

We have left open the question of how a node's height is to be increased when it is whitened. The only requirement is that neighbors should never have the same height. A particularly interesting scheme is to increase a node's height beyond all its neighbors' heights whenever it is whitened; this amounts to sending all incident tokens to the neighbors when a node is whitened. Under this strategy, the token values are immaterial: a white

node is blackened if it holds all incident tokens and upon being whitened, a node sends all incident tokens to the neighbors. Assuming that each edge (x, y) is directed from the token holder x to y, the graph is initially acyclic, and each blackening and whitening move preserves acyclicity. This is the strategy that was employed in solving the distributed dining philosophers problem in Chandy and Misra [29]; a black node is *eating* and a white node is *hungry*; constraint (S1) is the well-known requirement that neighboring philosophers do not eat simultaneously. The current problem has no counterpart of the *thinking* state, which added a slight complication to the solution in [29]. The tokens are called *forks* in that solution.

11.6 Designs of the Processors

In section 11.2, we described the operations of the processors as follows. A processor starts executing an action on receiving a message from the scheduler. It may call on methods of the other processors by sending messages and waiting for responses. Each call includes argument values, if any, as part of the message. It is guaranteed that each call elicits a response, which is either *accept* or *reject*. The accept response is sent when the call is accepted (which is always the case for calls upon total methods), and values of result-parameters, if any, are returned with the response. A reject response is possible for calls only on partial methods; no parameter values accompany such a response.

The description of the implementation can be simplified if we imagine that the scheduler is also a processor. It calls other processors for executions of their actions in the same way that processors call each other for executions of their methods. Thus, we treat actions and methods similarly in the implementation, except that only the former may be called from the scheduler.

Each processor has a buffer *reqq* (request queue) into which all requests sent to it are deposited and a buffer *resq* (response queue) into which all responses to its requests are deposited. In the following description, we assume that *reqq* is a fifo channel, though it can be implemented as a fair bag (see section 4.1.3). All we require is that every item from *reqq* is removed eventually if items are removed arbitrarily often, independent of the number of items added to *reqq*.

An entry in *reqq* is a triple (c, m, r), where c is the caller's id, m is the procedure to be executed, and r is the list of arguments of the call. On completion of execution of m, the processor stores its response in *resq* of caller c; an entry in *resq* is of the form (s, r), where s is the status (*accept* or *reject*) and r is the list of argument values returned if the status is *accept*. A response is guaranteed for each request. A processor with id i executes the following steps for each entry (c, m, r) of *reqq*.

- m is a total procedure:

  execute m; during its execution
  if procedure m' at c' is to be called with arguments r' **then**
  send (i, m', r') to c';
  on receiving a response (*accept*,r'') in $resq$,
  substitute the received argument values, r'', into i's state;
  endif
  on completion of m, send (*accept*,r) to c.

- m is a partial procedure:

  (Case 1) m has no alternative whose pre-condition holds:
  send (*reject*,–) to c.

  (Case 2) there is an alternative of the form $(p \rightarrow S)$ where p holds:

  execute S as in the case of a total procedure;
  on completion of execution of the body:
  if this is a positive alternative **then**
  send (*accept*,r) to c, where r is the parameter list
  else {this is a negative alternative} send (*reject*,–) to c
  endif

  (Case 3) there is an alternative of the form $(p; h \rightarrow S)$ where p holds:

  send (i, h, r') to c' where r' is the parameter list for h and
  c' is the id of the processor where action h is located;
  on receiving (*reject*,–) in $resq$, send (*reject*,–) to c;
  on receiving (*accept*,r') in $resq$,
  do the body of (Case 2), above.

11.7 Optimizations

We propose certain optimizations in this section that permit the scheduler to skip execution of an action because it can guarantee that the execution will be ineffective. This is particularly useful for Seuss because a tight execution typically introduces a busy form of waiting: if an action calls method P of a semaphore and is rejected, it calls P repeatedly as long as its pre-condition remains *true*. Similarly, a receiver from a channel repeatedly "polls" the channel for incoming messages. It is often more efficient for a caller to be queued; the semaphore box may queue all callers of P and serve them in order, and a channel may notify a receiver only when a message is ready to be delivered. Queue-based implementations can be encoded within

Seuss: instead of calling P, a different total method is called in order for the caller to be placed in a queue; when the semaphore is available the caller at the head of the queue is notified. Such an encoding makes programming cumbersome, so we have chosen tight execution as the basis for operational understanding of programs.

We can implement a tight execution in a way that mimics the more efficient scheme outlined above. We show how a scheduler can select actions for execution in such a way that a rejected call is not reattempted until there is some chance for its success. For example, if a pipeline is empty none of the actions that operate on its data can be executed effectively until some action introduces data into it. In this case, an action that fails to execute effectively is not chosen for execution until the pipeline is nonempty.

Note In the rest of this section we assume that each partial procedure has a single (positive) alternative. The changes that are needed to handle procedures with multiple alternatives and negative alternatives are sketched in section 11.7.4. □

Effective execution of an action depends on a set of predicates, the pre-conditions of all the partial methods that are called during its execution. For each action these pre-conditions can be determined from a static analysis of the program. If the scheduler has the exact values of these pre-conditions, it can decide which action will be executed effectively. However, ascertaining the exact values of the pre-conditions may be costly, requiring evaluations of (some of) these predicates after each effective execution of an action. We propose a strategy in which the scheduler knows only that some pre-conditions are definitely *false*. A pre-condition that is not known to be *false* may or may not be *false*.

The scheduler attempts execution of an action only when none of its pre-conditions is known to be *false*. The execution may still be ineffective, but the scheduler then learns that a certain pre-condition is *false*, and the call is not repeated until that pre-condition changes state (so that it is not known to be *false*).

Just like compatibility, we expect the user to declare the effects of actions on pre-conditions. We show that two forms of declarations —an action is guaranteed to establish a predicate, and guaranteed to preserve its truth— are sufficient to develop the optimization strategy. As before, the user is not *required* to make the declarations; they affect performance, not correctness.

11.7.1 Data structures for optimization

From a static analysis of a program we construct a *needs* graph. This is a bipartite graph in which the actions are one set of nodes and the pre-conditions of the partial procedures form the other set. There is an edge (f, p) where f is an action and p a pre-condition of some procedure that may

be called during an execution of f. More formally, using the terminology of chapter 10, there is an edge (f, p) if and only if

$$\langle \exists g :: \ f \ calls^* \ g \ \wedge \ (p \text{ is the pre-condition of } g) \rangle$$

In this formulation, g is a partial procedure (because total procedures have no pre-conditions). Note that g could be f itself since $calls^*$ is reflexive. We say f *needs* p if there is an edge (f, p). Execution of f is effective iff all pre-conditions p hold, where f *needs* p.

With each pre-condition p in the needs graph associate a boolean variable ps, the *shadow* of p. The shadow[1] ps gives some information about the value of p through the following invariant:

Shadow invariant: $p \ \Rightarrow \ ps$

The shadow invariant is equivalent to $\neg ps \Rightarrow \neg p$; that is, if ps is *false*, then so is p. However, the truth of ps tells us nothing about p's value. If f *needs* p and $\neg ps$ holds, the execution of f will be ineffective.

11.7.2 Operation of the scheduler

We describe a centralized scheduler here; an outline of a distributed scheduler is given on page 357.

In the earlier sections of this chapter, we have described the operation of the scheduler as follows: choose an action for execution, send a message to the appropriate processor, receive a message on completion of the execution, and then update the internal data structures (for heights). These steps are now augmented as follows.

- on choosing an action f for execution:
  if there is an edge (f, p) and $\neg ps$ holds
  then skip
  else send a message to the appropriate processor
  endif

- on receiving a message from a processor (on completion of execution): update the shadow variables so that the shadow invariant holds; see section 11.7.3.

The first operation can be implemented efficiently by keeping a count of the number of (f, p) edges for which $\neg ps$ holds, for each action f. This count can be updated efficiently, and only an action with zero count is selected for execution.

The correctness of these operations is obvious from the preceding discussion. Next, we show how the shadow invariant is maintained.

---

[1] We ought to write $p.s$ for the shadow; we have dropped the "." for readability.

11.7.3 Maintaining the shadow invariant

The simplest strategy for maintaining the shadow invariant is to strengthen the invariant to $p \equiv ps$ and require that each processor inform the scheduler of the changes in pre-condition values following execution of an action. This strategy has the benefit that the scheduler can decide exactly if an action execution will be effective. Therefore, calls to processors result in only effective executions. However, performance may suffer since the pre-condition of each partial procedure may have to be evaluated following the execution of a procedure in that box. Therefore, we suggest an alternative strategy based on the original shadow invariant, $p \Rightarrow ps$.

We ask that a processor return the following information to the scheduler upon completion of each action execution.

1. If the execution is ineffective, the identity of the pre-condition that was found to be *false*.

2. If the execution is effective, the names of the executed procedures. The sequence in which the procedures are executed is immaterial.

One-way scheme

We ask that the user declare a set of procedures $p.np$ for each pre-condition p, as follows. Procedure g is in $p.np$ only if the following conditions hold.

(1) g and p are from the same box.
(2) g preserves $\neg p$, i.e., $\{\neg p\}\ \ g\ \ \{\neg p\}$ holds.

(The mnemonic np in $p.np$ stands for negation preserver.) Note that, if g's execution is ineffective whenever $\neg p$ holds, it preserves $\neg p$.

The scheduler updates the shadow variables after receiving a message from a processor, as follows.

- If the execution is rejected because pre-condition p is *false*,
  $ps := \mathit{false}$

- If the execution is accepted, for each executed procedure g,
  $\langle \forall p :\ p \text{ in } g\text{'s box} :\ g \notin p.np\ \rightarrow\ ps := \mathit{true} \rangle$

It is obvious that the step corresponding to rejection preserves the invariant, $p \Rightarrow ps$, since both predicates are *false*. The acceptance step also preserves the invariant because setting ps to *true* preserves $p \Rightarrow ps$. For performance reasons, we would like to set ps to *true* only if it is essential. In the accepting execution case, if $g \in p.np$ then g preserves $\neg p$, so it is safe to leave ps at *false*. If $g \notin p.np$, we do not know if $\neg p$ is *true* or *false*; so we set ps to *true* to maintain the shadow invariant.

Note A shadow variable *ps* may be assigned several times because more than one procedure that affects *p* may have been executed during an effective execution. However, there is no conflict because the only value assigned is *true*. We call this scheme one-way because *ps* is set only to *true* following an effective execution. □

For the optimizations to be effective, *ps* should be made *false* as often as possible, so that the scheduler can reject an action without executing it. Hence *p.np* should be as large a set as possible. However, the user is not required to list *all* the procedures *g*, which meet the conditions for *p.np*, but the bigger the set, the better is the potential for performance improvement.

Example Consider the fifo channel described in section 4.1.1. The pre-condition *p* for procedure *get* is $r \neq \langle\rangle$. Negation of this predicate, $r = \langle\rangle$, is preserved only by *get*. Therefore, whenever *put* is executed —$put \notin p.np$— *ps* is set to *true*. Once *get* is rejected, *ps* is set to *false* and no further attempt is made to execute an action that calls *get* until another *put* has been executed. □

Two-way scheme

We sketch a generalization of the one-way scheme where *ps* is set to either *true* or *false* after effective execution of an action. In the one-way scheme, *ps* could only be set to *true*. Setting *ps* to *false* avoids executions of actions which are guaranteed to be ineffective.

Clearly, we need more information from the user; we ask the user to declare a set *p.ne*, analogous to *p.np*, for each pre-condition *p*. Informally, procedure *g* is in *p.ne* if its execution is guaranteed to result in a state where $\neg p$ holds; i.e., *g* is in *p.ne* only if the following conditions hold.

(1) *g* and *p* are from the same box.
(2) *g* establishes $\neg p$, i.e., $\{true\}\ \ g\ \ \{\neg p\}$ holds.

(The mnemonic *ne* in *p.ne* stands for negation establisher.) As before, *p.ne* may be a subset of the procedures that satisfy these conditions. Note that *p.np* can be augmented with all procedures in *p.ne* because the conditions for the latter are stricter than for the former. Henceforth, assume that $g \in p.ne \ \Rightarrow\ g \in p.np$. Note that the first condition in the definition of *p.ne* is superfluous, because $\neg p$ can be established only by a procedure in *p*'s box.

In the one-way scheme, the processor returns the *set* of executed procedures to the scheduler in case an action accepted. The order of execution of the procedures does not matter to the scheduler since *ps* is set only to *true*. In the two-way scheme, *ps* can be set to *true* or *false*, and the order of execution does matter. Therefore, the processors return the *sequence* in which the procedures are executed.

The operation of the scheduler is as follows.

- If execution is rejected because a pre-condition p is *false* :
  $ps := \mathit{false}$
- If execution is accepted, scan the sequence of accepted procedures in order. Let g be the next procedure in the sequence.

$$\begin{array}{l}\langle \forall p: \ p \text{ in } g\text{'s box}: \\ \quad g \notin p.np \ \rightarrow \ ps := \mathit{true} \\ [\!] \ g \in p.ne \ \rightarrow \ ps := \mathit{false} \\ \rangle\end{array}$$

The guards shown above, $g \notin p.np$ and $g \in p.ne$, are disjoint because $g \in p.ne \ \Rightarrow \ g \in p.np$. The argument that the shadow invariant is preserved in case of rejected calls and $g \notin p.np$ is as before. For the case of $g \in p.ne$, execution of g establishes $\neg p$, from $\{\mathit{true}\} \ \ g \ \ \{\neg p\}$. Therefore, setting ps to *false* satisfies $p \Rightarrow ps$.

Example Consider the one-item buffer, called *word*, from page 60. There are two pre-conditions, *full* and $\neg$*full*, for procedures *get* and *put*.

$$\begin{array}{l} full.ne \ = \ \{get\}, \ \neg full.ne \ = \ \{put\} \\ full.np \ = \ \{get\}, \ \neg full.np \ = \ \{put\} \end{array}$$

Once *put* (or *get*) is rejected another call is not attempted until an effective execution of *get* (*put*). □

11.7.4 Notes on the optimization scheme

More elaborate shadow variables

The scheduler can have a more accurate estimate of the value of a pre-condition p by having more possible values for ps and more user directives about how p is changed. For instance, let ps take on three possible values denoting that p is definitely *true*, definitely *false*, and unknown. Then the user may classify the procedures into those that (1) preserve p, (2) preserve $\neg p$, (3) establish p, (4) establish $\neg p$, (5) flip p to $\neg p$, (6) flip $\neg p$ to p, and (7) none of the above. The value of ps is transformed depending on the kind of procedure that has been executed. Such elaborate methods are appropriate only when the cost of communication and action rejection are high.

Distributing the scheduler

The scheduler can be distributed by dividing the needs graph so that each individual scheduler is assigned a portion of the graph for a subset of the actions. The nodes corresponding to a pre-condition p may be duplicated at several schedulers if actions at those schedulers need p. Also, updates of ps have to be synchronized.

Effect of concurrent execution on shadow invariant

The argument about the preservation of the shadow invariant is valid when there is at most one executing action; the action completes and the shadow variables are then updated. When actions are executed concurrently there seems to be a possibility that a pre-condition p is set to different values by two concurrently executing actions. Suppose one of them completes and causes the scheduler to set ps to *false* while the other one has set it to *true* in the meantime. Then the scheduler may decide against starting an action that needs p based on the value of ps, which is *false*. Fortunately, this possibility does not actually arise because concurrently executing actions are compatible; therefore, they do not ultimately assign conflicting values to p.

Dead Actions

The scheduler can identify certain actions that will never again be executed effectively. Let A be a set of actions and P a set of pre-conditions that actions in A need (effective execution of each action in A requires some pre-condition in P to hold). Let p be a pre-condition in P. Suppose every action whose execution can make p *true* is in A. Then all predicates in P remain *false* once they are *false* because no action in A can be executed effectively and set some p to *true*. Action f can set p to *true* only if it $calls^*$ some procedure g in p's box that is not known to preserve $\neg p$. Formally, all actions in A are dead if all members of P are *false* and

$$
\begin{array}{l}
\langle \forall p\colon p \in P\colon \\
\quad (\forall f\colon f \text{ is an action } \wedge\ \langle \exists g :: \ g \text{ in } p\text{'s box},\ f\ calls^*\ g,\ g \notin p.np \rangle : \\
\qquad\quad f \in A \\
\quad) \\
\rangle
\end{array}
$$

Multiple Alternatives

A partial procedure with multiple alternatives is executed effectively if *one* of the alternatives has a *true* pre-condition. Also, the system state can change even though the call is rejected, because a negative alternative may reject the call. The needs graph then becomes more elaborate.

Represent each alternative of each partial procedure as a node. Call an alternative *disabled* if the corresponding shadow variable is *false* or its pre-procedure is disabled. A procedure is disabled if *all* its alternatives are disabled. A disabled action's execution is ineffective.

With negative alternatives, a processor has to inform the scheduler if the rejection comes from a positive alternative because its pre-condition was *false* or from a negative alternative, in which case the sequence of executed procedures is returned by the processor. The first case is handled in the same way rejections were handled earlier; the second case is treated similarly to an effective execution.

11.8 Concluding Remarks

We expect the components of most concurrent programs to be loosely coupled, in the sense that they interact only occasionally and in a way that satisfies the compatibility requirements of chapter 10. Such programs — which include programs in which communications are over channels with one sender— are easily distributed over several processors. There is no need for arbitration among conflicting actions; therefore, the processors can operate autonomously. A scheduler has to be introduced only when there are incompatible actions and the scheduler's role is to decide which of the incompatible actions should be selected for execution. We surmise that most incompatibilities are localized, so distributed schedulers will be quite effective. One reason for developing a maximal scheduler is to have the option of customizing the scheduler for specific problems.

The *wait* and *signal* primitives in a monitor [90] are quite similar to the declarations *p.np* and *p.ne* that we have suggested for optimization. In a monitor for a one-word buffer, for instance, the *put* operation issues a *wait* for the pre-condition of *put* and a *signal* for the pre-condition of *get*; the *get* operation issues the opposite *wait* and *signal*. The declarations in the example on page 357 are the exact equivalents of these *wait*s and *signal*s.

Optimization is meaningful when there are choices. Seuss programs provide choice in the order of selection of actions for execution. This flexibility and the structure of an action —its dependence on the *conjunction* of a set of pre-conditions— have been exploited for optimizations in this chapter. The optimization scheme may be viewed as a program that executes on an abstraction of the original program's data space to gain some information about the program state at a low cost. The suggested optimizations look promising, but no honest assessment can be made without extensive empirical investigations.

11.9 Bibliographic Notes

The general scheduling strategy of section 11.3.3 is due to Rajeev Joshi. The distributed scheduling algorithm of section 11.5.2 is inspired by similar algorithms in the literature; see Gafni and Bertsekas [74], Chandy and Misra [29], and Barbosa and Gafni [17].

An implementation of Seuss using C^{++} as the host language is reported in Krüger [109], and with Java as the host language appears in Joshi [98] and Alvisi et al. [8]. The latter implementation includes a module construct to allow hierarchical structures of boxes and a declaration mechanism to specify the link constraints of section 9.3.1.

12
A Logic for Seuss

12.1 Introduction

The logic of action systems, developed in chapters 5 and 6, allowed us to specify safety and progress properties of a single box; the logic is extended in chapters 8 and 9 for specifications of ensembles of boxes. Properties such as **co** and *leads-to* specify the collective effect of the executions of the actions of a box or a set of boxes; the individual actions are not identified in a specification. Therefore, it is not possible to deduce from earlier specifications how a specific action affects the program state.

The earlier specifications are inadequate for Seuss programs. Because methods can be called from other boxes, the callers have to know the semantics of each method they call. A semaphore box, for instance, has to specify the meanings of P and V procedures. In this chapter, we develop the machinery for such specifications. As is traditional, we describe the meaning of a procedure by a pair of predicates, its pre-condition and post-condition; see Gries [79, chapter 12]. The semantics of a procedure depends on the semantics of the procedures it calls; additionally, for Seuss, we have to distinguish between the accepting and rejecting executions of a procedure, and specify the semantics for each.

In wide-area systems it is difficult to identify all the components (boxes), because the components are added and removed on a regular basis. Yet we would like to assert properties of a system without complete knowledge of all its components. Closure properties, introduced in section 9.3, form an ideal vehicle for such specifications. As long as each component obeys cer-

tain systemwide constraints, we can derive closure properties of individual components and assert that such properties are inherited by the system. For instance, a strong semaphore box may assume that each process that is granted the semaphore relinquishes it eventually and then assert that each persistent caller is granted the semaphore; see section 12.5 for such a specification and its proof. This is a closure property that is inherited by any system in which the strong semaphore is embedded.

Restrictions In this chapter, we restrict the Seuss model by requiring that procedures have only value parameters. This restriction simplifies the exposition. The general case is not fundamentally different; it merely adds complexity to the inference rules. Also, we restrict the discussion to partial procedures only. Total procedures have no rejecting executions. Therefore, they may be specified by pre-condition, post-condition pairs using standard techniques; see Gries [79, chapter 12] or Martin [129]. □

Overview of the chapter

In section 12.2, we show how to specify procedures that have no pre-procedures. We show proof of a progress property using such specifications for the readers-writers program of section 4.8.1. In section 12.3, we develop proof rules for partial procedures in the general form. Quite often a procedure is called again following a rejection; we term this form of calling *persistence* and develop inference rules in section 12.4 for persistence. A strong semaphore whose P method is called persistently is specified and proved in section 12.5. A resource allocation algorithm from section 4.10 is proved in section 12.6; the algorithm calls a strong semaphore, and the proof makes use of the specification of strong semaphore.

12.2 Specifications of Simple Procedures

We define a procedure by a pre-condition and post-condition. Our approach is slightly more elaborate than the traditional ones [79, chapter 12] or [129], because a procedure may accept or reject a call. In this section, we consider *simple* procedures, ones that have a single alternative and no pre-procedure. The specification scheme is extended to general procedures in section 12.3.

Definition: A state in which predicate p holds is called a p-state. For a procedure g,

$$\{p\} \;\; g^{+} \;\; \{q\}$$

denotes that any accepting execution of g starting in a p-state ends in a q-state. Similarly,

$$\{p\} \;\; g^{-} \;\; \{q\}$$

denotes that any rejecting execution of g starting in a p-state ends in a q-state. We write

$$\{p\}\ \ g\ \ \{q\} \quad \text{for} \quad \{p\}\ \ g^+\ \ \{q\} \text{ and } \{p\}\ \ g^-\ \ \{q\}$$

That is, $\{p\}\ \ g\ \ \{q\}$ stands for "any execution of g starting in a p-state ends in a q-state", because each execution is either accepting or rejecting. □

A procedure execution may affect the states of several boxes. Therefore, there is no restriction on the variables named in p and q, above. In practice, the variables are often limited to a single box.

The specification of a box is given by (1) a set of variables and their initial values, (2) the semantics of each procedure (given by a pre-condition, post-condition pair), and (3) a set of box properties, which are usually conditional and/or closure properties.

Auxiliary variables: number of procedure calls

For procedure g, let $\#g$, $\#g^+$, $\#g^-$ denote, respectively, the number of calls to g in an execution, and the number of accepted and rejected calls in that execution. Each of these are auxiliary variables; they may be defined by adding the appropriate code to the text of g. Note that each variable is nondecreasing and $\#g = \#g^+ + \#g^-$. We write $\#g(x)$ for the number of calls to g with argument x; $\#g^+(x)$ and $\#g^-(x)$ are similarly defined.

Deriving properties from a specification

The safety and progress properties of a box can be deduced from the specifications of its procedures. We need to apply the definitions from chapters 5 and 6 in such deductions. More importantly, closure properties of section 9.3 can also be deduced from the code of a single box. This is possible because the state of a box can be altered only by the procedures of that box. Therefore, it is possible to derive certain properties of a box independent of its environment. But the environment can dictate which methods of a box do get executed and when; therefore, the deduction is not entirely straightforward.

To illustrate the derivation of properties and some of the difficulties associated with it, consider the following example.

---

box *simple*
  integer $n = 0$;
  total method *inc* :: $n := n + 1$
  partial method *dec* :: $n > 0 \ \rightarrow\ n := n - 1$
end {*simple*}

For any positive integer k,

$$n = k \;\; \textbf{co} \;\; k-1 \leq n \leq k+1 \;\; \text{in } \textit{simple}$$

can be proved easily. However, the corresponding closure property does not hold; an external action may call *inc* twice during an invocation, thus increasing n by two. Hence, we cannot deduce some closure properties without knowledge of the external procedures. However, some closure properties can still be deduced from a single box, as shown next. In particular, we show that

$$n = k \;\; \textbf{cco} \;\; k-1 \leq n \;\; \text{in } \textit{simple}$$

and using lifting —see section 9.3.4— the property holds in any system in which *simple* is embedded.

Now we show how p **cco** q can be established for a box F, where p and q are predicates over the states of F and $p \Rightarrow q$. Suppose,

$\{q\} \;\; g \;\; \{q\}$, for every total method g in F , and
$\{p\} \;\; h \;\; \{q\}$, for every other procedure h in F .

Then p **cco** q in F.

We justify this claim as follows. Execution of an action h of F, partial or total, establishes a q-state starting in a p-state, from $\{p\} \;\; h \;\; \{q\}$. Next, consider execution of an action external to F starting in a p-state. Any such execution invokes a partial method of F at most once and total methods of F any finite number of times (including zero). Execution of any partial method of F establishes q, from $\{p\} \;\; h \;\; \{q\}$, and non-execution of any partial method also establishes q, from $p \;\Rightarrow\; q$. From then on, executions of total methods of F preserve q, from $\{q\} \;\; g \;\; \{q\}$. Therefore, the state following the execution of this action satisfies q.

Let p be $n = k$ and q be $k - 1 \leq n$ for any k, $k > 0$, for box *simple*.

$$\begin{array}{lll} \{n = k\} & dec^- & \{n = k\} \\ \{n = k\} & dec^+ & \{n = k-1\} \end{array}$$

Hence

$$\{n = k\} \qquad dec \qquad \{k - 1 \leq n\}$$

Also

$$\{k - 1 \leq n\} \quad inc \quad \{k - 1 \leq n\}$$

Therefore, $n = k \;\; \textbf{cco} \;\; k - 1 \leq n \;\; \text{in } \textit{simple}$.

The **cstable** and **cinvariant** properties are proved similarly. For **cen** properties, the closure theorem corollary (from page 301) is

$$p \;\; \textbf{cen} \;\; q \;\; \text{in } F \;\equiv\; (\textbf{transient } p \wedge \neg q \;\; \text{in } F) \;\wedge\; (p \wedge \neg q \;\; \textbf{cco} \;\; p \vee q \;\; \text{in } F)$$

The **cco** property in this formula can be established as explained above, and the progress property, **transient** $p \wedge \neg q$ in F, can be established by demonstrating an action of F that falsifies $p \wedge \neg q$, a proof that is entirely local to F.

12.2.1 readers-writers *with progress for* writers

Consider the *readers-writers* program of section 4.8.1. The solution guarantees that *reader*s do not permanently overtake *writer*s: if there is a waiting *writer*, some *writer* gains access to the resource eventually. The strategy is to reject calls upon *StartRead* if there is some *writer* attempting to execute *StartWrite*. A boolean variable, *WriteWait*, is set to *true* whenever a call upon *StartWrite* is rejected because there are active *reader*s. Once *WriteWait* holds, further calls to *StartRead* are rejected, preventing new *reader*s from commencing execution. Eventually, all executing *reader*s complete their executions; then the next call to *StartWrite* is accepted. The informal argument shows that if each rejected call on *StartWrite* is followed by a later call —i.e., *writer*s are persistent— then not all *writer*s are permanently blocked; i.e., some call on *StartWrite* will eventually succeed. We give a formal proof of this fact in this section.

We reproduce the program from section 4.8.1, using the following abbreviations: *StartRead*, *StartWrite*, *EndRead*, *EndWrite*, and *WriteWait* by *sr*, *sw*, *er*, *ew*, and *ww*, respectively.

---

box *ReaderWriter1*
  integer $nr, nw = 0, 0$;
  boolean $ww = \mathit{false}$;

  partial method *sr* ::
    $nw = 0 \ \wedge \ \neg ww \ \rightarrow \ nr := nr + 1$

  partial method *sw* ::
    $nr = 0 \ \wedge \ nw = 0 \ \rightarrow \ nw := 1; \ ww := \mathit{false}$
  $\not|$ $nr \neq 0 \ \rightarrow \ ww := \mathit{true}$

  total method *er* :: $nr := nr - 1$

  total method *ew* :: $nw := 0$
end {*ReaderWriter1*}

Specification of ReaderWriter1

In the following specification, variables *nr* and *nw* have the same meaning as in the program: *nr* is the number of active *reader*s and *nw* the number of active *writer*s. We assume that *er* and *ew* are called by *reader*s and *writer*s that are executing; these procedures have no effect if $nr = 0$ and $nw = 0$, respectively. In the following, free variable k ranges over natural numbers.

Specification *ReaderWriter1*
integer $nr, nw = 0, 0$;
boolean $ww = false$;

$\{nr = k \ \wedge \ nw = 0 \ \wedge \ \neg ww\} \ \ sr^+ \ \ \{nr = k + 1\}$

$\{nr = 0 \ \wedge \ nw = 0\} \ \ sw^+ \ \ \{nw = 1 \ \wedge \ \neg ww\}$

$\{nr \neq 0\} \ \ sw^- \ \ \{ww\}$

$\{nr = k\} \ \ er \ \ \{nr = k \ominus 1\}$
$\{k \ominus 1$ is 0 if k is 0, and $k - 1$ otherwise$\}$

$\{true\} \ \ ew \ \ \{nw = 0\}$

end $\{$*ReaderWriter1*$\}$

We use the convention that variables that are not mentioned in a post-condition are unchanged by the execution of the corresponding procedure. This specification is consistent with the code *ReaderWriter1* of section 4.8.1; prove this fact using standard techniques for sequential programs.

Derived properties of ReaderWriter1

All the properties of *ReaderWriter1* derived below are closure properties (see section 9.3.2). Therefore, each of these properties holds in any program of which *ReaderWriter1* is a component. Below, i and k are arbitrary natural numbers.

(P0) **cinvariant** $nr \geq 0 \ \wedge \ nw \geq 0$

(P1) **cinvariant** $(nr = 0 \ \vee \ nw = 0) \ \wedge \ (0 \leq nw \leq 1)$

(P2) $ww \ \wedge \ nr = k$ **cco** $nr \leq k$

(P3) $ww \wedge nr, nw = 0, 0 \wedge \#sw = i$ **cco** $(nr, nw = 0, 0 \wedge \#sw = i) \vee \neg ww$

(P4) **If**
(*reader*s complete) $nr = k \ \ \mathbf{c}\!\!\mapsto \ \ nr \neq k$
(*writer*s complete) $true \ \ \mathbf{c}\!\!\mapsto \ \ nw = 0$
(*writer*s are persistent) $ww \ \wedge \ \#sw = i \ \ \mathbf{c}\!\!\mapsto \ \ \#sw \neq i$
then
(*writer*s progress) $ww \ \ \mathbf{c}\!\!\mapsto \ \ \neg ww$

Properties (P0, P1) are intuitively obvious. (P2) says that no new *reader* is allowed to start reading if ww holds. (P3) asserts that an effective execution of *StartWrite* falsifies ww. (P4) specifies that persistent calling by the *writer*s eventually succeeds, provided that each *reader* and *writer* completes its execution.

Proofs of the derived properties

(P0) By inspection of the specification.

(P1) By inspection of the initial condition and pre- and post-conditions of each procedure.

(P2) The only methods that change nr are sr^+ and er. In a state where $ww \wedge nr = k$ holds, sr^+ is not executed effectively; hence, it satisfies $nr \leq k$. Total method er preserves $nr \leq k$, from its specification.

(P3) In a state where $ww \wedge nr, nw = 0, 0$ holds, only sw^+, er, or ew may be executed effectively. The result follows by inspecting the post-conditions of each of these actions.

(P4) The given progress property is quite nontrivial. So we give a considerably more formal proof than for (P0–P3).

$$
\begin{array}{lll}
ww \ \mathbf{c}\!\!\mapsto \ (ww \wedge nr = 0) \vee \neg ww & , \text{see lemma 1} \\
ww \wedge nr = 0 \ \mathbf{c}\!\!\mapsto \ (ww \wedge nr, nw = 0, 0) \vee \neg ww & \\
 & , \text{see lemma 2} \\
ww \wedge nr, nw = 0, 0 \ \mathbf{c}\!\!\mapsto \ \neg ww & , \text{see lemma 3} \\
ww \ \mathbf{c}\!\!\mapsto \ \neg ww & , \text{cancellation on above three}
\end{array}
$$

Lemma 1 $ww \ \mathbf{c}\!\!\mapsto \ (ww \wedge nr = 0) \vee \neg ww$

Proof: In the following, k is any natural number. We assume that nr and nw are natural numbers (see P0).

$$
\begin{array}{lll}
ww \wedge nr = k \ \mathbf{cco} \ nr \leq k & , \text{(P2)} \\
nr = k \ \mathbf{c}\!\!\mapsto \ nr \neq k & , \text{from the hypothesis of (P4)} \\
ww \wedge nr = k \ \mathbf{c}\!\!\mapsto \ nr < k \vee \neg ww & , \text{PSP and weaken rhs} \\
ww \wedge nr = k \ \mathbf{c}\!\!\mapsto \ (ww \wedge nr < k) \vee \neg ww & \\
 & , \text{rewrite rhs} \\
ww \wedge nr > 0 \ \mathbf{c}\!\!\mapsto \ (ww \wedge nr = 0) \vee \neg ww & \\
 & , \text{induction on positive integers} \\
ww \wedge nr = 0 \ \mathbf{c}\!\!\mapsto \ ww \wedge nr = 0 & , \text{implication} \\
ww \wedge nr \geq 0 \ \mathbf{c}\!\!\mapsto \ (ww \wedge nr = 0) \vee \neg ww & \\
 & , \text{disjunction} \\
ww \ \mathbf{c}\!\!\mapsto \ (ww \wedge nr = 0) \vee \neg ww & , \text{from (P0), } nr \geq 0 \equiv \mathit{true} \ \square
\end{array}
$$

Lemma 2 $ww \ \wedge \ nr = 0 \ \ \mathbf{c}\!\!\mapsto \ \ (ww \ \wedge \ nr, nw = 0, 0) \ \vee \ \neg ww$

Proof:

$ww \ \wedge \ nr = 0 \ \ \mathbf{cco} \ \ nr \leq 0$, replace k by 0 in (P2)
$ww \ \wedge \ nr = 0 \ \ \mathbf{cco} \ \ nr = 0$, use (P0: $nr \geq 0$) in the rhs
$true \ \ \mathbf{c}\!\!\mapsto \ \ nw = 0$, from the hypothesis of (P4)
$ww \ \wedge \ nr = 0 \ \ \mathbf{c}\!\!\mapsto \ \ (ww \ \wedge \ nr, nw = 0, 0) \ \vee \ \neg ww$
, PSP and weaken rhs □

Lemma 3 $ww \ \wedge \ nr, nw = 0, 0 \ \ \mathbf{c}\!\!\mapsto \ \ \neg ww$

Proof:

$ww \wedge nr, nw = 0, 0 \wedge \#sw = i \ \ \mathbf{cco} \ \ (nr, nw = 0, 0 \wedge \#sw = i) \vee \neg ww$
, (P3)
$ww \ \wedge \ \#sw = i \ \ \mathbf{c}\!\!\mapsto \ \ \#sw \neq i$, from the hypothesis of (P4)
$ww \ \wedge \ nr, nw = 0, 0 \ \wedge \ \#sw = i \ \ \mathbf{c}\!\!\mapsto \ \ \neg ww$
, PSP and weaken rhs
$ww \ \wedge \ nr, nw = 0, 0 \ \ \mathbf{c}\!\!\mapsto \ \ \neg ww$, disjunction over all i □

12.2.2 readers-writers *with progress for both*

Consider the readers-writers box, *ReaderWriter2*, of section 4.8.2; it guarantees progress for both *readers* and *writers*. An informal argument for progress was given in that section. Here, we develop a formal proof. The program from page 76 is given next, using the abbreviations used for *ReaderWriter1*, see page 365. Also, we abbreviate *ReadWait* to *rw*.

---

box *ReaderWriter2*
  integer $nr, nw = 0, 0$;
  boolean $ww, rw = false, false$;

  partial method *sr* ::
    $nw = 0 \ \wedge \ \neg ww \ \ \rightarrow \ \ nr := nr + 1; \ rw := false$
  $\not|$ $nw \neq 0 \ \ \rightarrow \ \ rw := true$

  partial method *sw* ::
    $nr = 0 \ \wedge \ nw = 0 \ \wedge \ \neg rw \ \ \rightarrow \ \ nw := 1; \ ww := false$
  $\not|$ $nr \neq 0 \ \ \rightarrow \ \ ww := true$

  total method *er* :: $nr := nr - 1$

  total method *rw* :: $nw := 0$
end {*ReaderWriter2*}

Next, we give a specification and prove (P4) of section 12.2.1 as a progress property for *ReaderWriter2*. The specification is analogous to the specification of *ReaderWriter1* given on page 366.

In the following specification, variable k ranges over natural numbers.

Specification *ReaderWriter2*
  integer $nr, nw = 0, 0$;
  boolean $ww, rw = \mathit{false}, \mathit{false}$;

$\{nr = k \ \wedge \ nw = 0 \ \wedge \ \neg ww\} \ \ sr^+ \ \ \{nr = k + 1 \ \wedge \ \neg rw\}$

$\{nw \neq 0\} \ \ sr^- \ \ \{rw\}$

$\{nr = 0 \ \wedge \ nw = 0 \ \wedge \ \neg rw\} \ \ sw^+ \ \ \{nw = 1 \ \wedge \ \neg ww\}$

$\{nr \neq 0\} \ \ sw^- \ \ \{ww\}$

$\{nr = k\} \ \ er \ \ \{nr = k \ominus 1\}$
  $\{k \ominus 1$ is 0 if k is 0, and $k - 1$ otherwise$\}$

$\{\mathit{true}\} \ \ ew \ \ \{nw = 0\}$

end $\{$*ReaderWriter2*$\}$

Derived properties of ReaderWriter2

We show that the four safety properties, (P0–P3) on page 366, hold for *ReaderWriter2*. Therefore, (P4) holds as well, because its proof depends only on (P0–P3).

Additionally, we have the following progress property for *reader*s, analogous to the progress property (P4) for *writer*s. Its proof is similar to the proof of (P4); we leave the proof to the (human) reader.

(P5) **If**

(*reader*s complete)	$nr = k \ \ \mathbf{c}\!\!\mapsto \ \ nr \neq k$
(*writer*s complete)	$\mathit{true} \ \ \mathbf{c}\!\!\mapsto \ \ nw = 0$
(*reader*s are persistent)	$rw \ \wedge \ \#sr = j \ \ \mathbf{c}\!\!\mapsto \ \ \#sr \neq j$

then

(*reader*s progress)	$rw \ \ \mathbf{c}\!\!\mapsto \ \ \neg rw$

Proofs of properties (P0–P3*)*

(P0) By inspection.

(P1) We prove a stronger invariant from which (P1) follows:

$$Q::\ \langle (nr = 0\ \wedge\ \neg ww)\ \vee\ (nw = 0\ \wedge\ \neg rw)\rangle\ \wedge\ \langle 0 \leq nw \leq 1\rangle$$

Observe from the post-conditions of the procedures that $0 \leq nw \leq 1$ is established following effective execution of each procedure. For the remaining term, $(nr = 0\ \wedge\ \neg ww)\ \vee\ (nw = 0\ \wedge\ \neg rw)$, note that:

sr^+:	establishes $\neg rw$; preserves pre-condition $nw = 0$
sr^-:	$nr = 0\ \wedge\ \neg ww$ is a pre-condition from Q; it is preserved
sw^+:	establishes $\neg ww$; preserves pre-condition $nr = 0$
sw^-:	$nw = 0\ \wedge\ \neg rw$ is a pre-condition from Q; it is preserved
er:	preserves both disjuncts, $nr = 0\ \wedge\ \neg ww$, $nw = 0\ \wedge\ \neg rw$
ew:	preserves both disjuncts, $nr = 0\ \wedge\ \neg ww$, $nw = 0\ \wedge\ \neg rw$

(P2) Analogous to the proof of (P2) on page 367.

(P3) Analogous to the proof of (P3) on page 367.

12.3 Specifications of General Procedures

The procedures considered in the previous section had no alternatives or pre-procedures. A specification mechanism for general procedures is developed in this section.

As before, $\{p\}\ g^+\ \{q\}$ denotes that any accepting execution of g starting in a p-state ends in a q-state; $\{p\}\ g^-\ \{q\}$ is defined analogously. We define a procedure *pskip*, the counterpart of *skip*, as follows.

$$pskip ::\quad true\ \rightarrow\ skip$$

To simplify discussion every partial procedure, except *pskip*, that has no pre-procedure is taken to have *pskip* as its pre-procedure. The semantics of *pskip* is given by, for any p and q,

pskip rule

$$(\{p\}\ pskip^+\ \{q\})\ \equiv\ (p \Rightarrow q)$$
$$\{p\}\ pskip^-\ \{q\}$$

For accepting executions the pskip rule is easy to see: since the effective execution of *pskip* has the same effect as *skip*, they have the same semantics. Since *pskip* has no rejecting executions, all its rejecting executions perform "magic", transforming any state to any state.

In the following rules, g is a partial procedure whose alternatives have disjoint pre-conditions. Also, assume that the alternatives are exhaustive,

i.e., that the disjunction of their pre-conditions is *true*. This condition can be met by adding a negative alternative, $e \rightarrow skip$, to each procedure where e holds when all other pre-conditions are *false* (i.e., $e \equiv \langle \forall i :: \neg c_i \rangle$, where c_i is the pre-condition of alternative i and i ranges over all alternatives).

Acceptance rule

$$\frac{\langle \text{For every positive alternative } c; h \rightarrow S \text{ of } g :: \{p \wedge c\}\ h^+;\ S\ \{q\} \rangle}{\{p\}\ g^+\ \{q\}}$$

Rejection rule

$$\frac{\begin{array}{c}\langle \text{For every negative alternative } c; h \rightarrow S \text{ of } g :: \{p \wedge c\}\ h^+;\ S\ \{q\} \rangle \\ \langle \text{For every alternative } c; h \rightarrow S \text{ of } g :: \{p \wedge c\}\ h^-\{q\} \rangle\end{array}}{\{p\}\ g^-\ \{q\}}$$

As before,

$\{p\}\ g\ \{q\}$ is $\{p\}\ g^+\ \{q\}$ and $\{p\}\ g^-\ \{q\}$

The acceptance and rejection rules encode the execution strategy of Seuss. An accepting execution of g entails execution of a positive alternative $c; h \rightarrow S$, which means h first accepts and then S is executed. The reasons for g rejecting a call are twofold: (1) a negative alternative $c; h \rightarrow S$ is executed where h accepts, or (2) some alternative $c; h \rightarrow S$ is executed where h rejects. The first case gives rise to conditions as in the acceptance rule, and in the second case the body of the alternative, S, is not executed.

Note It is often convenient to write

$\{p \wedge c\}\ h^+;\ S\ \{q\}$ as $\{p \wedge c\}\ h^+\ \{wlp.q.S\}$

where $wlp.q.S$ is the weakest liberal pre-condition, introduced by Dijkstra [55]. Since S is terminating, $wlp.q.S$ and $wp.q.S$ are identical. □

12.3.1 Derived rules

The following observations, similar to the ones for **co** (see page 100), are immediate from the properties of wlp [61]. We show the rules for g; the rules apply to g^+ and g^- as well.

- $\{false\}\ g\ \{p\}$
- $\{p\}\ g\ \{true\}$
- (conjunction; disjunction) $\dfrac{\{p\}\ g\ \{q\}\ ,\ \{p'\}\ g\ \{q'\}}{\begin{array}{c}\{p \wedge p'\}\ g\ \{q \wedge q'\} \\ \{p \vee p'\}\ g\ \{q \vee q'\}\end{array}}$

The conjunction and disjunction rules follow from the conjunctivity and monotonicity properties of *wlp* [61] and of logical implication. These rules generalize in the obvious manner to any set —finite or infinite— because *wlp* and logical implication are universally conjunctive and universally disjunctive. As corollaries of conjunction and disjunction we obtain

- (lhs strengthening) $\dfrac{\{p\}\ \ g\ \ \{q\}}{\{p \wedge p'\}\ \ g\ \ \{q\}}$
- (rhs weakening) $\dfrac{\{p\}\ \ g\ \ \{q\}}{\{p\}\ \ g\ \ \{q \vee q'\}}$

12.3.2 Simplifications of the derived rules

The following special cases of the inference rules for acceptance and rejection often simplify the proofs in practice.

Locality

Suppose predicates p and q do not name any variable of h's box. Then

$$\{p\}\ \ h^+;\ S\ \ \{q\} \quad \equiv \quad \{p\}\ \ S\ \ \{q\}$$

We leave the proof of this result to the reader. Its most useful application is when procedure g has a single alternative, $c; h \ \rightarrow\ S$, and predicates p and q name only variables of g's box. Then the acceptance rule is simplified to

$$\frac{\{p \wedge c\}\ \ S\ \ \{q\}}{\{p\}\ \ g^+\ \ \{q\}}$$

and the rejection rule becomes

$$\frac{p\ \Rightarrow\ q}{\{p\}\ \ g^-\ \ \{q\}}$$

The locality rule can be applied when a call has no explicit result parameter: g calls h to receive a signal of some kind —granting of a resource, for instance— to proceed, and the signal is transmitted by h accepting the call.

Absence of explicit negative alternative

Suppose g explicitly mentions only positive alternatives,. Let c_i be the precondition of alternative i. Then g has a hidden negative alternative

$$e\ \rightarrow\ skip$$

where $e\ \equiv\ \langle \forall i ::\ \neg c_i \rangle$.

This alternative contributes the following conditions to the antecedent of the rejection rule.

$\{p \wedge e\}\ \ pskip^+;\ skip\ \ \{q\}$
$\{p \wedge e\}\ \ pskip^-\ \ \{q\}$

These two conditions simplify to $p \wedge e\ \Rightarrow\ q$. Therefore, the rejection rule is simplified to

$$\frac{\langle \text{For every positive alternative } c; h\ \rightarrow\ S \text{ of } g ::\ \{p \wedge c\}\ \ h^-\ \ \{q\}\rangle \qquad p \wedge e\ \Rightarrow\ q}{\{p\}\ \ g^-\ \ \{q\}}$$

12.4 Persistence and Relative Stability

If a procedure is called *persistently*, then each rejected call on the procedure is eventually followed by another call. This is the essence of "busy waiting", and we formalize this notion in this section. We specify a strong semaphore in section 12.5 as one that eventually grants the semaphore, i.e., it accepts calls on its P method, provided that it is called persistently. A persistently-called procedure may never accept, thus ensuring that calls to it continue forever. Such is the case with weak semaphore, in which no guarantee can be given that a particular caller of P is eventually granted the semaphore, even though it repeats the calls endlessly.

Each action in a Seuss program is called persistently because it is called infinitely often by a scheduler, independent of whether it accepts or not. A procedure $g ::\ c; h\ \rightarrow\ S$ may be called persistently and yet h may not be called persistently, because pre-condition c may not always hold when g is being called. We introduce *relative stability* on page 374 to state that once c becomes *true* it remains *true* at least until g accepts a call.

12.4.1 Persistence

For procedure g the predicate ∂g^+ denotes that the most recent call on g was accepted, and ∂g^- denotes that it was rejected. Initially, ∂g^+ is *true* and ∂g^- is *false*; therefore, ∂g^+ and ∂g^- are negations of each other. Both predicates are auxiliary variables like $\#g^+$ and $\#g^-$.

Predicate p *holds whenever* g *is called* is given by, for any m,

$$\#g = m\ \wedge\ \neg p\ \ \mathbf{cco}\ \ \#g = m$$

That is, any step that has $\neg p$ as a pre-condition does not change $\#g$, i.e., g is not called if $\neg p$ holds.

Procedure g is *called persistently* if for every integer n, $n \geq 0$,

$$\partial g^-\ \wedge\ \#g = n\ \ \mathbf{c}\!\mapsto\ \ \#g > n$$

Eventually g accepts is given by

$$true \;\; \mathbf{c}\!\!\mapsto \;\; \partial g^+$$

We write $p \rhd g$ to denote

if
  p holds whenever g is called and
  g is called persistently
then
  eventually g accepts

Formally, $p \rhd g$ stands for

$$\begin{array}{l} \langle \forall m, n :: \\ \quad \#g = m \;\wedge\; \neg p \;\; \mathbf{cco} \;\; \#g = m \\ \quad \partial g^- \;\wedge\; \#g = n \;\; \mathbf{c}\!\!\mapsto \;\; \#g > n \rangle \\ \Rightarrow \\ \quad \langle true \;\; \mathbf{c}\!\!\mapsto \;\; \partial g^+ \rangle \end{array}$$

For procedure g with parameters, g is *called persistently* means that each call to g that is rejected is eventually followed by another call with the same argument values. This is particularly useful when the argument encodes the identity of the caller. Then, persistent calling means that each rejected caller calls again. The definition of persistent calling in this case is, for all possible argument values x,

$$\partial g^-(x) \;\wedge\; \#g(x) = n \;\; \mathbf{c}\!\!\mapsto \;\; \#g(x) > n$$

The other definitions are similarly modified.

Note $(p \;\rhd\; g) \;\Rightarrow\; (p \wedge q \;\rhd\; g)$. □

12.4.2 Relative stability

A predicate is stable if it remains *true* once it becomes *true*. We consider a variation of this notion called relative stability. Predicate p is *stable relative to* action g if p can be falsified only by an accepting execution of g. In practice, p is usually a predicate over g's box. Then it is sufficient to show that no *procedure* of that box, except possibly g^+, falsifies p. Since no method falsifies p —g is an action— no external action can falsify p by calling a method of g's box. Therefore, once p holds, it continues to hold as long as g is not executed.

12.4.3 Inference rules

We show two inference rules that we have found to be useful.

(PC1) Let g be an action.

$$\frac{(p \rhd g),\ \{p\}\ \ g^+\ \ \{q\},\ p \textit{ stable relative to } g}{p\ \ \mathbf{c}\!\!\mapsto\ \ q}$$

We justify this inference rule in operational terms. Consider a state in which p holds. From relative stability of p, it continues to hold at least until g has an accepting execution. Therefore, p is a pre-condition whenever g is called, and since g is an action, it is called persistently. Hence, from $p \rhd g$, eventually g accepts; from $\{p\}\ \ g^+\ \{q\}$ it establishes q.

Note The operator $\mathbf{c}\!\!\mapsto$ in the consequent of (PC1) may be strengthened to **cen** because p remains *true* until q is established, and g^+ establishes q. □

The following corollary of (PC1) is an important generalization. It may be justified similarly. Recall, from section 5.4.5, that for properties α and β of program F, we write $\alpha\ \Rightarrow\ \beta$ to mean that taking α as an additional premise β can be inferred as a property of F.

Corollary of PC1 Let g be an action and R be a system property.

$$\frac{R\ \Rightarrow\ (p \rhd g),\ \{p\}\ \ g^+\ \ \{q\},\ p \textit{ stable relative to } g}{R\ \Rightarrow\ (p\ \ \mathbf{c}\!\!\mapsto\ \ q)}$$

Note We note without proof

$$\frac{R \wedge p\ \Rightarrow\ (q \rhd g)}{R\ \Rightarrow\ (p \wedge q\ \rhd\ g)}$$

□

The next inference rule defines the effect of persistent calling on a procedure in terms of persistent calls on its pre-procedure.

(PC2) Suppose procedure g has a positive alternative $c; f\ \rightarrow\ S$ and g is the only caller of f.

$$\frac{(p \rhd f)}{(p \wedge c\ \rhd\ g)}$$

To justify this rule we start with the consequent, $(p \wedge c\ \rhd\ g)$. Assume that (1) $p \wedge c$ holds whenever g is called and (2) g is called persistently. The goal is to show that g accepts eventually. Since $c; f\ \rightarrow\ S$ is an alternative of g, using (1), f is called each time g is called, and p holds whenever f is called.[1] Hence, from $p \rhd f$, procedure f accepts eventually, and g as the sole caller of f accepts too.

The condition that g is the sole caller of f does not restrict the applicability of this rule. Whenever a procedure is called with distinct arguments

[1] Actually, $p \wedge c$ holds whenever f is called, but since c is local to g's box c has little relevance for f.

we regard the calls as being made to distinct procedures. Therefore, processes that pass their ids as arguments can claim to be the sole caller of that procedure with that argument.

Analogous to the corollary of (PC1) we have

Corollary of PC2 Let procedure g have a positive alternative $c; f \rightarrow S$, and g be the only caller of f. Let R be a system property.

$$\frac{R \;\Rightarrow\; (p \rhd f)}{R \;\Rightarrow\; (p \wedge c \;\rhd\; g)}$$

12.5 Strong Semaphore

The example of strong semaphore illustrates various aspects of persistence. We start with the code from section 4.9.2, propose a specification (that includes persistent calling) and prove the consistency of the specification with the code. In section 12.6 we consider a resource allocation algorithm (of section 4.10) in which the resources are identified with strong semaphores. We use the specification given in this section to prove absence of starvation in the resource allocation algorithm.

The following code for strong semaphore is taken from page 81. The only alteration is to replace the identifier *avail* by b to shorten the formulae in the proofs. Variable b is *true* if the semaphore is available, and q is the queue of process-ids whose most recent calls on P have been rejected.

---

cat *StrongSemaphore*
  seq(id) $q = \langle\rangle$;
  boolean $b = true$;

  partial method $P(i$: id) ::
    $b \;\wedge\; i = q.head \;\rightarrow\; b, q := false, q.tail$
    $\not| \; i \notin q \;\rightarrow\; q := q +\!\!+ i$ {i is appended to q}

  total method V:: $b := true$
end {*StrongSemaphore*}

12.5.1 Specification of strong semaphore

The specification is written using variable b only; variable q is not exposed. Specifications of P^+, P^-, and V are straightforward: P^+ falsifies b, P^- does not disturb b, and V sets b to *true*. We discuss the progress property later. In the following, free variable B is boolean and i is quantified over all process-ids.

Specification *StrongSemaphore*
boolean $b = true$;

$\{b\}\ \ P^{+}(i)\ \ \{\neg b\}$

$\{b = B\}\ \ P^{-}(i)\ \ \{b = B\}$

$\{true\}\ \ V\ \ \{b\}$

$\langle true\ \ \mathbf{c}\!\!\mapsto\ \ b\rangle\ \Rightarrow\ \langle true\ \rhd P(i)\rangle$

end {*StrongSemaphore*}

The progress property, $\langle true\ \ \mathbf{c}\!\!\mapsto\ \ b\rangle\ \Rightarrow\ \langle true\ \rhd P(i)\rangle$, says that if b is infinitely often *true* —i.e., the semaphore is always eventually available because every holder of the semaphore releases it eventually— and P is called persistently, then P will eventually accept a call from every caller, i.e., every caller eventually holds the semaphore.

Note The specification of $P^{+}(i)$ does not say that $P^{+}(i)$ accepts if b holds; it says that if b holds *and* $P(i)$ accepts, then $\neg b$ holds as a post-condition. The condition under which P accepts is embedded in the progress property in the specification.

We remarked toward the end of section 4.9.2 that a strong semaphore requires every caller to be persistent. It can be proved that there is no implementation of strong semaphore in the presence of even one nonpersistent caller —which calls P, gets rejected, and never calls again, for instance. □

12.5.2 Proof of the specification

The first three properties in the specification, which give the semantics of P^{+}, P^{-}, and V, are easy to establish. We show a proof of

$$\langle true\ \ \mathbf{c}\!\!\mapsto\ \ b\rangle\ \Rightarrow\ \langle true\ \rhd P(i)\rangle$$

The proof is developed through a series of propositions, (S0–S8).

(S0) Elements in q are distinct:
This proposition is *true* initially, and an item is added to q, in the rejecting alternative of P, only if it is not already in q.

(S1) $i = q.head\ \ \mathbf{cco}\ \ i = q.head\ \vee\ i \notin q$:
The head item remains in the queue until removed. This is a property of any queue whose items are all distinct.

(S2) $(i = q.head \;\; \mathbf{c}\!\!\mapsto \;\; i \notin q) \;\Rightarrow\; (true \;\; \mathbf{c}\!\!\mapsto i \notin q)$:
If the head item of a queue is eventually removed, every item is eventually removed. This is a property of any queue.

We derive properties (S3–S5) from the program.

(S3) $i \in q \;\equiv\; \partial P^{-}(i)$.

(S4) $i \in q \;\; \mathbf{cco} \;\; i \in q \;\vee\; \partial P^{+}(i)$.

(S5) $b \wedge i = q.head \wedge \#P(i) = n \;\; \mathbf{cco} \;\; (b \wedge i = q.head \wedge \#P(i) = n) \vee (i \notin q)$.

We show only the proof of (S3). The remaining proofs are similar.

• Proof of (S3), $i \in q \;\equiv\; \partial P^{-}(i)$:
This proposition holds initially because $q = \langle\rangle$ (i.e., $i \in q$ is *false*) and $\partial P^{-}(i)$ is *false*. We show that the proposition is stable, i.e.,

$$\begin{array}{ccc}
\{i \in q \;\equiv\; \partial P^{-}(i)\} & P^{+}(i) & \{i \in q \;\equiv\; \partial P^{-}(i)\} \\
\{i \in q \;\equiv\; \partial P^{-}(i)\} & P^{-}(i) & \{i \in q \;\equiv\; \partial P^{-}(i)\} \\
\{i \in q \;\equiv\; \partial P^{-}(i)\} & V & \{i \in q \;\equiv\; \partial P^{-}(i)\}
\end{array}$$

Execution of $P(j)$, $j \neq i$, does not affect $i \in q$ or $\partial P^{-}(i)$. That is why we consider only $P(i)$.

We show only the proof of the first assertion; the remaining two are similarly proved. For the first proof, add the assignments to the auxiliary variables explicitly and use (S0) as an additional conjunct —writing qd to denote that all items in q have distinct values— and show

$$\begin{array}{c}
\{(i \in q \equiv \partial P^{-}(i)) \;\wedge\; b \;\wedge\; i = q.head \;\wedge\; qd\} \\
b, q, \partial P^{-}(i), \partial P^{+}(i) := \; false, q.tail, false, true \\
\{i \in q \;\equiv\; \partial P^{-}(i)\}
\end{array}$$

Using the axiom of assignment (see appendix A.4.1), the proof amounts to showing:

$$\langle (i \in q \equiv \partial P^{-}(i)) \;\wedge\; b \;\wedge\; i = q.head \;\wedge\; qd \rangle \;\Rightarrow\; \langle i \;\notin\; q.tail \rangle$$

This follows from $(i = q.head \;\wedge\; qd) \;\Rightarrow\; (i \;\notin\; q.tail)$. □

We are now ready to prove the progress property in the specification

$$\langle true \;\; \mathbf{c}\!\!\mapsto \;\; b \rangle \;\Rightarrow\; \langle true \;\rhd P(i) \rangle$$

We assume the antecedents of this property — $true \;\; \mathbf{c}\!\!\mapsto \;\; b$, see (S6), and the antecedent of $true \;\rhd P(i)$, see (S7)— and prove the consequent of $true \;\rhd P(i)$, see (S8).

(S6) $true \;\; \mathbf{c}\!\!\mapsto \;\; b$

(S7) $\partial P^{-}(i) \;\wedge\; \#P(i) = n \;\; \mathbf{c}\!\!\mapsto \;\; \#P(i) > n$

(S8) $true \;\; \mathbf{c}\!\!\mapsto \;\; \partial P^{+}(i)$

- Proof of (S8), $true \;\; \mathbf{c}\!\!\mapsto \;\; \partial P^{+}(i)$:

$\partial P^{-}(i) \;\wedge\; \#P(i) = n \;\; \mathbf{c}\!\!\mapsto \;\; \#P(i) > n$, (S7)
$b \;\wedge\; i = q.head \;\wedge\; \#P(i) = n \;\; \mathbf{cco}$
$\quad (b \;\wedge\; i = q.head \;\wedge\; \#P(i) = n) \;\vee\; (i \notin q)$
, (S5)
$b \;\wedge\; i = q.head \;\wedge\; \#P(i) = n \;\wedge\; \partial P^{-}(i) \;\; \mathbf{c}\!\!\mapsto \;\; i \notin q$
, PSP and weaken rhs
$b \;\wedge\; i = q.head \;\wedge\; \partial P^{-}(i) \;\; \mathbf{c}\!\!\mapsto \;\; i \notin q$, disjunction over n
$i = q.head \;\Rightarrow\; i \in q$ and $i \in q \;\equiv\; \partial P^{-}(i)$, queue property, S3
$b \;\wedge\; i = q.head \;\; \mathbf{c}\!\!\mapsto \;\; i \notin q$, from above two (1)
$i = q.head \;\; \mathbf{cco} \;\; i = q.head \;\vee\; i \notin q$, (S1)
$true \;\; \mathbf{c}\!\!\mapsto \;\; b$, (S6)
$i = q.head \;\; \mathbf{c}\!\!\mapsto \;\; (b \;\wedge\; i = q.head) \;\vee\; i \notin q$, PSP and weaken rhs
$i = q.head \;\; \mathbf{c}\!\!\mapsto \;\; i \notin q$, cancellation with (1)
$true \;\; \mathbf{c}\!\!\mapsto \;\; i \notin q$, (S2) and above
$i \in q \;\; \mathbf{cco} \;\; i \in q \;\vee\; \partial P^{+}(i)$, (S4)
$i \in q \;\; \mathbf{c}\!\!\mapsto \;\; \partial P^{+}(i)$, PSP and weaken rhs
$\partial P^{-}(i) \;\; \mathbf{c}\!\!\mapsto \;\; \partial P^{+}(i)$, (S3): $i \in q \;\equiv\; \partial P^{-}(i)$
$\partial P^{+}(i) \;\; \mathbf{c}\!\!\mapsto \;\; \partial P^{+}(i)$, implication
$true \;\; \mathbf{c}\!\!\mapsto \;\; \partial P^{+}(i)$, disjunction, above two □

12.6 Starvation Freedom in a Resource Allocation Algorithm

We consider the resource allocation algorithm given in section 4.10. The program given on page 87 and reproduced in section 12.6.1 is deadlock-free when each resource is managed by a weak semaphore. We claimed in section 4.10 that the program is starvation-free when the resources are managed by strong semaphores. We prove this claim here, using the specification of strong semaphore from section 12.5.

The proof of absence of starvation is quite subtle. The action *acquire* in a *user* program calls procedure P of semaphore $r[d]$ (that manages resource numbered d) provided that the *user* is *hungry*, needs resource d, and has already acquired all lower-numbered resources. It has to be shown in the proof that repeated calls to $r[d].P$ result in the *user* being granted the

corresponding semaphore. But from the specification of strong semaphore (see section 12.5), such a claim can be made only if the semaphore is eventually released by its holder; a holder releases a semaphore only after it eats, i.e., only if the solution is starvation-free. The seeming circularity of this argument is resolved by having the *user*s acquire the semaphores in a particular order.

Typical proofs of this algorithm in the literature derive a contradiction if there is starvation, along the lines sketched in section 4.10. Here, we avoid proof by contradiction, using induction on resource numbers to establish the theorem. The theorem says that every holder of a semaphore, j or above, releases it eventually, and every *user* that is *hungry* and has acquired all resources below j will eventually enter the *eating* state. The proof uses the specification of strong semaphore and the inference rules of section 12.4.3.

12.6.1 The resource allocation program

In the following, $user_i$ is the box for *user* with id i; this id appears as an argument in the call to $r[d].P$.

---

box $r[0..N]$: *StrongSemaphore*

box $user_i$
  array$[0..N]$**(boolean)** $needs = false$;
  enum $(0..N+1)$ $d = 0$;
  enum $(thinking, hungry, eating)$ $state = thinking$;

  partial action *acquire* ::
      $hungry \ \wedge \ d \leq N \ \wedge \ \neg needs[d] \quad \rightarrow \ d := d+1$
    | $hungry \ \wedge \ d \leq N \ \wedge \ needs[d]; \ r[d].P(i) \ \rightarrow \ d := d+1$

  partial action *eat* ::
      $hungry \ \wedge \ d > N \ \rightarrow$ $state := eating$; use resources

  partial action *release* ::
      $thinking \ \wedge \ d > N \ \rightarrow$
        while $d \neq 0$ **do**
          $d := d-1$; **if** $needs[d]$ **then** $r[d].V$ **endif**
        enddo
end $\{user_i\}$

Note As noted in section 4.10, the transition from *thinking* to *hungry* is not shown; it is part of an underlying program that sets the boolean array *needs*, where $needs[i]$ holds if the process needs resource i. Also, the

eating-thinking transition is not shown explicitly in this program. Every *eating* process eventually transits to *thinking*, and *needs* and d remain unchanged by the transition. In particular, we assume the following closure property of $user_i$:

$$e \ \wedge \ needs[k] \ \ \mathbf{c}\!\!\mapsto \ \ t \ \wedge \ d > N \ \wedge \ needs[k] \qquad \square$$

12.6.2 Proof of absence of starvation

We prove that every *user* eats eventually if it is *hungry*, i.e., with h_i and e_i denoting that *user* i is *hungry* and *eating*, respectively, $h_i \ \ \mathbf{c}\!\!\mapsto \ e_i$ (see corollary to the main theorem of this section).

Notation Henceforth, j and k are integers, $0 \leq j \leq N+1$ and $0 \leq k \leq N$. We employ the following abbreviations in the proof:

$$b_k \ \equiv \ r[k].b$$
$$V_j \ \equiv \ \langle \wedge k : j \leq k \leq N : \ true \ \ \mathbf{c}\!\!\mapsto \ b_k \rangle$$

Thus, b_k says that resource k is available, and V_j says that eventually every resource numbered j or higher is available. We write t, h, and e instead of $thinking, hungry$, and $eating$. All the variables in the proof —t, h, e, d, and *needs*— refer to an arbitrary *user* program; we add a subscript when necessary to identify a particular *user*. $\square$

The following lemma says that if all resources numbered j or higher are eventually available, then a *user* who has acquired all needed resources below j and needs j will acquire resource j.

Lemma 4 $V_j \ \Rightarrow \ (h \ \wedge \ d \leq N \ \wedge \ needs[d] \ \wedge \ d \geq j \ \ \mathbf{c}\!\!\mapsto \ h \ \wedge \ d > j)$.

Proof: We apply the corollary of (PC1), see page 375, to establish this result. Let g be action *acquire*, and

$$p \ \equiv \ h \ \wedge \ d \leq N \ \wedge \ needs[d] \ \wedge \ d \geq j$$
$$q \ \equiv \ h \ \wedge \ d > j$$
$$R \equiv \ V_j$$

The consequent of the corollary, $R \ \Rightarrow \ (p \ \ \mathbf{c}\!\!\mapsto \ q)$, establishes lemma 4. We have to show the following properties that appear in the antecedent of that corollary:

1. $V_j \ \Rightarrow \ (p \rhd g)$
2. $\{p\} \ \ g^+ \ \ \{q\}$
3. *p stable relative to g*

The last two properties are easily established. For (2), note that only the second alternative of *acquire* may accept under the given pre-condition p,

and its acceptance increases the value of d, thus establishing q. For (3), no action (or alternative) except the second alternative of *acquire* is executed effectively under pre-condition p.

Next, we prove (1) by applying the corollary of (PC2). Action g, which is *acquire*, has a positive alternative c; $f \ \rightarrow \ S$, where

$c \ \equiv \ h \ \wedge \ d \leq N \ \wedge \ needs[d]$
f is $r[d].P(i)$

As noted earlier,

$p \ \equiv \ h \ \wedge \ d \leq N \ \wedge \ needs[d] \ \wedge \ d \geq j$
$R \equiv \ V_j$

First, assume $V_j \ \wedge \ p$.

$d \geq j$	, from p
$true \ \ \mathbf{c}\!\!\mapsto \ \ r[d].b$	, from V_j and above
$true \ \ \rhd \ r[d].P(i)$	, above and specification of $r[d]$ (section 12.5.1)

Hence,

$V_j \ \wedge \ p \Rightarrow \langle true \ \ \rhd \ r[d].P(i)\rangle$	, from above proof
$V_j \ \Rightarrow \ \langle p \ \rhd \ r[d].P(i)\rangle$	, above, note after PC1-corollary
$V_j \ \Rightarrow \ \langle p \wedge c \ \rhd \ g\rangle$	, corollary of PC2
$V_j \ \Rightarrow \ \langle p \ \rhd \ g\rangle$	, above, using $p \wedge c \ \equiv \ p$

Lemma 5 $h \ \wedge \ d \leq N \ \wedge \ \neg needs[d] \ \wedge \ d \geq j \ \ \mathbf{c}\!\!\mapsto \ \ h \ \wedge \ d > j$

Proof:

$h \ \wedge \ d \leq N \ \wedge \ \neg needs[d] \ \wedge \ d = j \ \ \mathbf{cen} \ \ h \ \wedge \ d > j$
, with the lhs pre-condition the only effective execution is for the first alternative of *acquire*
$h \ \wedge \ d \leq N \ \wedge \ \neg needs[d] \ \wedge \ d = j \ \ \mathbf{c}\!\!\mapsto \ \ h \ \wedge \ d > j$
, basis rule for $\mathbf{c}\!\!\mapsto$
$h \ \wedge \ d \leq N \ \wedge \ \neg needs[d] \ \wedge \ d > j \ \ \mathbf{c}\!\!\mapsto \ \ h \ \wedge \ d > j$
, implication
$h \ \wedge \ d \leq N \ \wedge \ \neg needs[d] \ \wedge \ d \geq j \ \ \mathbf{c}\!\!\mapsto \ \ h \ \wedge \ d > j$
, disjunction □

Lemma 6 $h \ \wedge \ d > N \ \ \mathbf{c}\!\!\mapsto \ \ e$

Proof:

$h \ \wedge \ d > N \ \ \mathbf{cen} \ \ e$, under the lhs pre-condition the only action executed effectively is *eat*
$h \ \wedge \ d > N \ \ \mathbf{c}\!\!\mapsto \ \ e$, basis rule for $\mathbf{c}\!\!\mapsto$ □

Next we state two lemmas without proof; they are easily proved from the program text. The first one says that a *thinking* process has either

released all resources or it holds all of the resources used in the previous eating session. The second invariant says that if semaphore k is unavailable, some process i needs it ($needs_i[k]$) and has acquired it ($d_i > k$).

Lemma 7 $t \;\Rightarrow\; (d = 0 \;\vee\; d > N)$

Lemma 8 $\neg b_k \;\Rightarrow\; \langle \exists i :: \; d_i > k \;\wedge\; needs_i[k] \rangle$

We are now ready to prove the main theorem from which absence of starvation follows as a corollary.

Theorem For all j, $0 \leq j \leq N+1$: V_j and $h \;\wedge\; d \geq j \;\; \mathbf{c}\!\!\mapsto \;\; e$.
Proof: By backward induction on j.

Case $j = N + 1$**:**

$V_{N+1} \equiv true$, definition of V_{N+1}
$h \;\wedge\; d > N \;\; \mathbf{c}\!\!\mapsto \;\; e$, lemma 6

Case $j = k,\; 0 \leq k \leq N$**:** We assume

V_{k+1} and $h \;\wedge\; d > k \;\; \mathbf{c}\!\!\mapsto \;\; e$

and show

V_k and $h \;\wedge\; d \geq k \;\; \mathbf{c}\!\!\mapsto \;\; e$

• Proof of V_k :

$t \;\wedge\; d > N \;\wedge\; needs[k] \;\; \mathbf{cen} \;\; b_k$, consider action *release*
$t \;\wedge\; d > N \;\wedge\; needs[k] \;\; \mathbf{c}\!\!\mapsto \;\; b_k$, basis rule for $\mathbf{c}\!\!\mapsto$ (1)
$e \;\wedge\; needs[k] \;\; \mathbf{c}\!\!\mapsto \;\; t \;\wedge\; d > N \;\wedge\; needs[k]$
, see note in section 12.6.1
$e \;\wedge\; needs[k] \;\; \mathbf{c}\!\!\mapsto \;\; b_k$, transitivity (above, 1) (2)
$e \;\wedge\; d > k \;\wedge\; needs[k] \;\; \mathbf{c}\!\!\mapsto \;\; b_k$, strengthen lhs (3)
$t \;\wedge\; d > k \;\Rightarrow\; t \;\wedge\; d > N$, from lemma 7
$t \;\wedge\; d > k \;\wedge\; needs[k] \;\Rightarrow\; t \;\wedge\; d > N \;\wedge\; needs[k]$
, above and predicate calculus
$t \;\wedge\; d > k \;\wedge\; needs[k] \;\; \mathbf{c}\!\!\mapsto \;\; b_k$, strengthen (1) with above (4)
$h \;\wedge\; d > k \;\; \mathbf{c}\!\!\mapsto \;\; e$, induction hypothesis
$h \;\wedge\; needs[k] \;\; \mathbf{cco} \;\; (h \vee e) \;\wedge\; needs[k]$
, program text
$h \;\wedge\; d > k \;\wedge\; needs[k] \;\; \mathbf{c}\!\!\mapsto \;\; e \;\wedge\; needs[k]$
, PSP on above two
$h \;\wedge\; d > k \;\wedge\; needs[k] \;\; \mathbf{c}\!\!\mapsto \;\; b_k$, transitivity with (2) (5)
$d > k \;\wedge\; needs[k] \;\; \mathbf{c}\!\!\mapsto \;\; b_k$, disjunction of (3,4,5)
$\langle \exists i :: \; d_i > k \;\wedge\; needs_i[k] \rangle \;\; \mathbf{c}\!\!\mapsto \;\; b_k$
, disjunction over all i (6)

$\neg b_k \ \ \mathbf{c}\!\!\mapsto \ b_k$, strengthen lhs, use lemma 8
$true \ \ \mathbf{c}\!\!\mapsto \ b_k$, disjunction with $b_k \ \mathbf{c}\!\!\mapsto b_k$
V_{k+1} , induction hypothesis
V_k , above two, from definition:
$V_k \ \equiv \ V_{k+1} \ \wedge \ (true \ \ \mathbf{c}\!\!\mapsto \ b_k)$ □

• Proof of $h \ \wedge \ d \geq k \ \ \mathbf{c}\!\!\mapsto \ e$:

$h \ \wedge \ d \leq N \ \wedge \ needs[d] \ \wedge \ d \geq k \ \ \mathbf{c}\!\!\mapsto \ h \ \wedge \ d > k$
, from V_k and lemma 4
$h \ \wedge \ d \leq N \ \wedge \ \neg needs[d] \ \wedge \ d \geq k \ \ \mathbf{c}\!\!\mapsto \ h \ \wedge \ d > k$
, from lemma 5
$h \ \wedge \ d \leq N \ \wedge \ d \geq k \ \ \mathbf{c}\!\!\mapsto \ h \ \wedge \ d > k$, disjunction of above two
$h \ \wedge \ d > k \ \ \mathbf{c}\!\!\mapsto \ e$, induction hypothesis
$h \ \wedge \ d \leq N \ \wedge \ d \geq k \ \ \mathbf{c}\!\!\mapsto \ e$, transitivity on above two
$h \ \wedge \ d > N \ \wedge \ d \geq k \ \ \mathbf{c}\!\!\mapsto \ e$, lemma 6: strengthen lhs
$h \ \wedge \ d \geq k \ \ \mathbf{c}\!\!\mapsto \ e$, disjunction of above two □

The following corollary concludes the progress proof.

Corollary $h \ \ \mathbf{c}\!\!\mapsto \ e$.
Proof: Set j to 0 in the statement of the above theorem.

$h \ \wedge \ d \geq 0 \ \ \mathbf{c}\!\!\mapsto \ e$
$\Rightarrow$ {Substitution axiom: **cinvariant** $d \geq 0$}
$h \ \ \mathbf{c}\!\!\mapsto \ e$ □

12.7 Concluding Remarks

The proof rules developed in this chapter allow us to state facts about specific procedures, unlike the proof rules of chapters 5 and 6 which describe only program properties. The inference rules for persistence, for instance, name the procedures explicitly. We have also employed auxiliary variables, like $\#g$, that count the number of procedure executions. We believe that state-based reasoning in the style of earlier chapters is inadequate for object-based programming models like Seuss. This chapter represents a preliminary attempt at integrating state-based and procedure-based reasoning.

Mathematical systems often evolve through a long process of experimentation. The right set of axioms in an algebra, constructs in a programming language, or inference rules of a logic are often arrived at after experimenting with the alternatives. The goal of experimentation is to devise a system that is elegant and effectively applicable to the problems in its intended domain. The logic for action systems, described in chapters 5 and 6, has been

"engineered" over a long period, and its effectiveness on a variety of problems has been demonstrated. The material in this chapter has had a much shorter gestation period. We present it now in the hope that its publication will inspire other researchers to undertake the necessary experimentation.

12.8 Bibliographic Notes

The notions of pre- and post-conditions are from Floyd [71] and Hoare [89]. The *wp*-calculus and an elaborate treatment of predicate transformers and their applications in program semantics are in Dijkstra and Scholten [61]. The notion of persistence was first formulated by Rajeev Joshi. The result that *all* callers of P in a strong semaphore must be persistent for a solution to exist appears in [100]. A promising approach for integrating state-based and procedure-based reasoning, called "computation calculus", has been developed by Rutger Dijkstra [63].

In Retrospect

The ostensible goal of this book is to propose a methodology and programming constructs for the development of distributed applications. The book turns out to be more about good programming practices: modularization, loose coupling of modules, clean interfaces, and the centrality of specification in program design. These concepts are not new, and they have been known to be essential in both sequential and concurrent programming. I have developed each concept within a concrete programming model and logic. Modules are codified as boxes, interfaces among boxes are limited to communication through procedure calls, and specifications are written as properties in a fragment of linear temporal logic. A module is a wholly self-contained unit that can be specified, designed, and understood in complete isolation from the other modules, and its properties are inherited by any system in which it is embedded. The key to this self-containment is to include the specification of the environment of a module within a module specification (see conditional and closure properties of chapter 9), and to eliminate interference among concurrently-executing actions by restricting such actions to be compatible. The ultimate goal is to develop a software design methodology in which a system specification is partitioned over its modules and the modules are then designed independently.

A large portion of this book has been devoted to the development of a logic. For a researcher, such a development is the ultimate test of the consistency and simplicity of the ideas. For a practitioner, though, it is the effectiveness of the logic in problem solving that counts. A programmer should use logic to aid the mental process and save time, much as an engineer uses algebra and calculus in preference to intuitive leaps of imag-

ination for the most expeditious design. A lot of work has gone into "engineering" the logic in this book so that it is a help rather than a hindrance in specification and design. A large number of proofs have been analyzed to ensure that the logic does not force a proof to state any more than the bare essentials, and that the proof steps can be combined effectively to deduce new properties. Much of this engineering remains to be done for the general logic of Seuss (see chapter 12). It is heartening that practicing engineers have begun to use sophisticated theorem provers [105, 106, 147], and action-system logics [118, 119], affirmation that "if you build it they will come".

This book leaves much for future research. I have remarked in the preface about the need for empirical confirmation of the programming model, a necessary first step not only for developing confidence in the model but for setting the future research direction. A number of enhancements to the model are required for practical work, among them permitting hierarchy in the constructions of boxes and dynamic creations of boxes. Some of the hardest problems have to do with efficient implementations, only a sketch of which appears in chapter 11. Even though the boxes are designed for autonomous executions, some form of control (central or distributed) is essential for efficient task scheduling, enforcement of link constraints (section 9.3.1) and implementation of a directory look-up service, which would permit multiple copies of the same box to coexist to speed up accesses from different parts of the network.

The number and variety of amazing applications that are yet to be designed constitute the enormous potential of the Internet. This book is a small step toward realizing that potential.

Appendix A
Elementary Logic and Algebra

A.1 Propositional Calculus

We assume that the reader is familiar with propositional logic; we merely give a short list of propositional identities that have been used in this book; for a thorough treatment see Dijkstra and Scholten [61, chapter 5] or Gries and Schneider [80, chapters 3, 4, 5].

We consider the following propositional operators: $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\equiv$ (equivalence), and $\Rightarrow$ (implication). The equality operator ($=$) is defined over all domains. Traditionally, it is written as $\equiv$ when applied to booleans; operator $\equiv$ has the lowest binding power among all logical operators, whereas operator $=$ has higher binding power than all logical operators except negation ($\neg$).

- (Commutativity and Associativity) $\wedge$, $\vee$, $\equiv$ are commutative and associative.

- (Idempotence) $\wedge, \vee$ are idempotent:
  $p \vee p \equiv p$
  $p \wedge p \equiv p$

- (Distributivity) $\wedge$, $\vee$ distribute over each other:
  $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
  $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

- (Absorption)
  $p \wedge (p \vee q) \equiv p$
  $p \vee (p \wedge q) \equiv p$

- (Laws with constants)

$p \wedge true$	$\equiv$	p	$p \wedge false$	$\equiv$	$false$
$p \vee true$	$\equiv$	$true$	$p \vee false$	$\equiv$	p
$p \vee \neg p$	$\equiv$	$true$	$p \wedge \neg p$	$\equiv$	$false$
$p \equiv p$	$\equiv$	$true$	$p \equiv \neg p$	$\equiv$	$false$

- (Double negation)
  $\neg\neg p \equiv p$

- (De Morgan)
  $\neg(p \wedge q) \equiv (\neg p \vee \neg q)$
  $\neg(p \vee q) \equiv (\neg p \wedge \neg q)$

- (Implication operator)
  $(p \Rightarrow q) \equiv (\neg p \vee q)$
  $(p \Rightarrow q) \equiv (\neg q \Rightarrow \neg p)$
  If $(p \Rightarrow q)$ and $(q \Rightarrow r)$, then $(p \Rightarrow r)$, i.e.,
  $\langle (p \Rightarrow q) \wedge (q \Rightarrow r) \rangle \Rightarrow \langle p \Rightarrow r \rangle$

- (Equivalence)
  $(p \equiv q) \equiv (p \wedge q) \vee (\neg p \wedge \neg q)$
  $(p \equiv q) \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$

- (Monotonicity)
  Let $p \Rightarrow r$. Then,
  $(p \wedge q) \Rightarrow (r \wedge q)$
  $(p \vee q) \Rightarrow (r \vee q)$

Strengthening, Weakening Predicate r *strengthens* (or, is a strengthening of) p if $r \Rightarrow p$; therefore, $p \wedge q$ strengthens p. Similarly, r *weakens* (or, is a weakening of) p if $p \Rightarrow r$; therefore, $p \vee q$ weakens p. □

Priorities of Operators The logical operators in decreasing order of priorities (binding powers) are $\neg$, $=$, $\wedge$ and $\vee$, $\Rightarrow$, $\equiv$. Note that $=$ and $\equiv$ have different priorities though they have the same meaning when applied to boolean operands. Therefore, $p \wedge q = r \wedge s$ is equivalent to $p \wedge (q = r) \wedge s$, whereas $p \wedge q \equiv r \wedge s$ is $(p \wedge q) \equiv (r \wedge s)$. Operators $\wedge$ and $\vee$ have the same priorities, so we use parentheses if there is a possibility of ambiguity (as in $p \wedge q \vee r$). To aid the reader in parsing logical formulae visually, we often put extra space around operators of lower priorities, as in $p \wedge q \equiv r \vee s$. Predicate $x, y = m, n$ is an abbreviation for $x = m \wedge y = n$. □

A.2 Predicate Calculus

A.2.1 Quantification

We use quantification in writing arithmetic and boolean expressions. In all cases, a quantified expression is of the following form: $\langle \otimes x :\ q(x) :\ e(x) \rangle$. Here, $\otimes$ is any commutative, associative binary operator, x is the *bound* variable (or a list of bound variables), $q(x)$ is a predicate that determines the *range* of the bound variables, and $e(x)$ is an expression called the *body*. A quantified expression in which the range is implicit is written in the following form: $\langle \otimes x ::\ e(x) \rangle$. We use other brackets in addition to angular brackets "⟨" and "⟩" to delimit the quantified expressions. Some examples of quantified expressions are given below.

$$\langle + i :\ 0 \le i \le N :\ A[i] \rangle \tag{1}$$
$$\langle \forall i :\ 0 \le i < N :\ A[i] \le A[i+1] \rangle \tag{2}$$
$$\langle \forall i, j :\ 0 \le i \le N \ \wedge\ 0 \le j \le N \ \wedge\ i \ne j :\ M[i,j] = 0 \rangle \tag{3}$$
$$\langle min\ i :\ 0 \le i \le N \ \wedge\ (\forall j :\ 0 \le j \le N :\ M[i,j] = 0) :\ i \rangle \tag{4}$$
$$\langle max\ p :\ p \in P :\ p.next(t) \rangle \tag{5}$$

To evaluate a quantified expression, (1) compute all possible values of the bound variable x that satisfy range predicate $q(x)$, (2) instantiate the body $e(x)$ with each value computed in (1), and (3) combine the instantiated expressions in (2) using operator $\otimes$. If the range is empty, the value of the expression is the unit element of operator $\otimes$; unit elements of some common operators are as given next, in parentheses following the operator: $+$ (0), $\times$ (1), $\wedge$ (*true*), $\vee$ (*false*), $\equiv$ (*true*), min ($+\infty$), max ($-\infty$) .

The values of the example expressions are as follows. Expression (1) is the sum of the array elements $A[0], \ldots, A[N]$. Expression (2) is *true* iff $A[0], \ldots, A[N]$ are in ascending order. Expression (3) has two bound variables; this boolean expression is *true* iff all off-diagonal elements of matrix $M[0..N, 0..N]$ are zero. Expression (4) is the smallest-numbered row in M all of whose elements are zero; if there is no such row the expression evaluates to ∞. Expression (5) is the maximum of all $p.next(t)$ where p is in P.

A.2.2 Textual substitution

For an expression q, $q[x := e]$ is the expression obtained by replacing in q all free occurrences of x by e. Textual substitution plays an essential role in establishing properties from program text; see section A.4.1.

A.2.3 Universal and Existential quantification

In quantified boolean expressions, we often use the existential quantifier $\exists$ and universal quantifier $\forall$ in place of $\vee$ and $\wedge$. The following are some of the useful identities.

- (Empty range)

$$\begin{array}{lcl} \langle \forall i : \mathit{false} : b \rangle & \equiv & \mathit{true} \\ \langle \exists i : \mathit{false} : b \rangle & \equiv & \mathit{false} \end{array}$$

- (Trading)

$$\begin{array}{lcl} \langle \forall i : q : b \rangle & \equiv & \langle \forall i :: q \Rightarrow b \rangle \\ \langle \exists i : q : b \rangle & \equiv & \langle \exists i :: q \wedge b \rangle \end{array}$$

- (Move-out) Given that i does not occur as a free variable in p,

$$\begin{array}{lcl} p \vee \langle \forall i : q : b \rangle & \equiv & \langle \forall i : q : p \vee b \rangle \\ p \wedge \langle \exists i : q : b \rangle & \equiv & \langle \exists i : q : p \wedge b \rangle \end{array}$$

- (De Morgan)

$$\begin{array}{lcl} \neg \langle \exists i : q : b \rangle & \equiv & \langle \forall i : q : \neg b \rangle \\ \neg \langle \forall i : q : b \rangle & \equiv & \langle \exists i : q : \neg b \rangle \end{array}$$

- (Range weakening) Given that $q \Rightarrow q'$,

$$\begin{array}{lcl} \langle \forall i : q' : b \rangle & \Rightarrow & \langle \forall i : q : b \rangle \\ \langle \exists i : q : b \rangle & \Rightarrow & \langle \exists i : q' : b \rangle \end{array}$$

- (Body weakening) Given that $b \Rightarrow b'$,

$$\begin{array}{lcl} \langle \forall i : q : b \rangle & \Rightarrow & \langle \forall i : q : b' \rangle \\ \langle \exists i : q : b \rangle & \Rightarrow & \langle \exists i : q : b' \rangle \end{array}$$

A number of identities can be derived from the trading rule (consult Gries and Schneider [80, chapter 9]); we show two below.

$$\begin{array}{lcl} \langle \forall i : q \wedge r : b \rangle & \equiv & \langle \forall i : q : r \Rightarrow b \rangle \\ \langle \exists i : q \wedge r : b \rangle & \equiv & \langle \exists i : q : r \wedge b \rangle \end{array}$$

The following duals of the move-out rule are valid iff range q is not *false*.

$$\begin{array}{lcl} p \wedge \langle \forall i : q : b \rangle & \equiv & \langle \forall i : q : p \wedge b \rangle \\ p \vee \langle \exists i : q : b \rangle & \equiv & \langle \exists i : q : p \vee b \rangle \end{array}$$

A.3 Proof Format

The proof format shown below, due to W.H.J. Feijen, is a convenient tool for writing detailed proofs. Let $\Rightarrow$ denote any transitive relation (not necessarily implication over predicates) over proof terms. A proof term may be a predicate, an arithmetic expression (in which case an arithmetic relation like $<$ or $\leq$ is used in place of $\Rightarrow$), or a property in Seuss logic. A proof of $p \Rightarrow s$ may be structured as follows.

$$
\begin{array}{ll}
 & p \\
\Rightarrow & \{\text{why } p \Rightarrow q\} \\
 & q \\
\Rightarrow & \{\text{why } q \Rightarrow r\} \\
 & r \\
\Rightarrow & \{\text{why } r \Rightarrow s\} \\
 & s
\end{array}
$$

Seuss properties **co** and $\mapsto$ (see chapters 5 and 6) can have their left side strengthened and right side weakened; also each one is transitive. Therefore, a proof using these operators and implication over predicates is valid.

This is not the only proof format we employ. In many proofs, we write one proof term per line, where the term may have been derived from a combination of several previous proof terms. In such cases, the justification of the proof term explains how the term was derived.

A.4 Hoare Logic and Weakest Pre-conditions

A.4.1 Hoare logic

We use Hoare logic [89] to reason about procedure bodies. Write $\{p\}\ s\ \{q\}$ for predicates p and q and a sequential program s to denote that if program s is started in a state that satisfies p, it terminates in a state that satisfies q. We restrict the discussion to terminating programs only. Assume $\{p\}\ s\ \{true\}$ and $\{false\}\ s\ \{p\}$ hold for all p and s.

We prove $\{p\}\ skip\ \{q\}$ by showing $p \Rightarrow q$, because *skip* does not modify the program state.

Axiom of Assignment If s is an assignment statement, say $x := e$, $\{p\}\ s\ \{q\}$ can be established by proving $p \Rightarrow q[x := e]$, where $q[x := e]$ is the predicate obtained by replacing in q all free occurrences of x by e. □

For a guarded command $C :: g \rightarrow s$, $\{p\}\ C\ \{q\}$ is shown by proving $\{p \wedge g\}\ s\ \{q\}$ and $p \wedge \neg g \Rightarrow q$. An important special case —corresponding to **co** properties in section 5.2— arises when $p \Rightarrow q$; then it is sufficient to prove $\{p \wedge g\}\ s\ \{q\}$ to establish $\{p\}\ C\ \{q\}$.

To show that p is transient under weak fairness —see, for instance, section 6.3.2— we have to prove $\{p\}\ \ C\ \ \{\neg p\}$, for some guarded command $C :: \ g\ \rightarrow\ s$. The assertions to be proven in this case reduce to $\{p \wedge g\}\ \ s\ \ \{\neg p\}$ and $p\ \Rightarrow\ g$.

A.4.2 Weakest pre-conditions

The weakest pre-condition of program s with respect to predicate q is written as $wp.s.q$, see Dijkstra [55]. A state satisfies $wp.s.q$ iff starting an execution of s in that state results in a state (on termination of s) that satisfies q. Thus, $\{p\}\ s\ \{q\}$ is the same as $p\ \Rightarrow\ wp.s.q$. We regard $wp.s$ as a predicate transformer. The notion of weakest liberal pre-condition (wlp) is also developed in [55]. This notion differs from the weakest pre-condition only when s is nonterminating; since we consider only terminating sequential programs —the bodies of procedures— the two notions are identical for our purposes. The properties of interest, given that $wp\ \equiv\ wlp$, are

- $wp.s$ is universally conjunctive and universally disjunctive.
- $wp.s.false\ \ \equiv\ false$.
- $wp.s$ is monotonic, i.e., $(p \Rightarrow q)\ \Rightarrow\ (wp.s.p \Rightarrow wp.s.q)$.

A.5 Elementary Relational Calculus

This is a very brief and elementary introduction to relational calculus. The reader should consult a standard book on modern algebra, such as MacLane and Birkhoff [126], for a treatment of relations, and a source, such as Rutger Dijkstra [62], for more recent developments in relational calculus and program semantics. The material presented here is sufficient for understanding the proofs in chapter 10, the only chapter in which relational calculus is used.

A *binary relation* (henceforth, simply called a *relation*) r over domain D is a subset of $D \times D$. The relation that corresponds to the empty set is ϵ. The *union* of two relations is a relation that is obtained by taking the union of the corresponding sets. For relations r and s over the same domain, their *product* is written as rs or $r \circ s$; it is defined as follows:

$$rs\ =\ \{(x,z)|\ \langle \exists y ::\ (x,y) \in r\ \wedge\ (y,z) \in s\rangle\}$$

The transitive closure r^* of relation r is the smallest relation that satisfies

$$r^*\ =\ r\ \cup\ rr^*\ .$$

As is traditional for sets, we write $r \subseteq s$ to denote that relation r is contained in relation s.

The following properties of relations are used in chapter 10. For relations r, s, and t over the same domain,

- Union is commutative and associative.
- Product is associative.
- (ϵ is zero of relational algebra)
  $r \cup \epsilon = r$
  $r\epsilon = \epsilon$ and $\epsilon r = \epsilon$
- (Monotonicity of union)
  $(r \subseteq s) \Rightarrow (t \cup r \subseteq t \cup s)$
- (Monotonicity of product)
  $(r \subseteq s) \Rightarrow (rt \subseteq st)$
  $(r \subseteq s) \Rightarrow (tr \subseteq ts)$
- (Distributivity of product over union)
  $t(r \cup s) = (tr \cup ts)$
  $(r \cup s)t = (rt \cup st)$
  Distributivity holds as well for unions of infinite number of relations.

We are interested in relational calculus mainly because of the way programs can be represented as relations. Let domain D be the state space of a program. Predicate p on D is a subset of D, and we represent p by a relation P where

$$P = \{(x, x) |\ x \in p\}$$

Predicate *true* is $D \times D$ and *false* is ϵ. Observe that $p \vee q$ is represented by $P \cup Q$ where P and Q correspond to p and q. Similarly, $\neg p$ is $(D \times D) - P$.

A program is a relation; the pair (s, t) is in the relation iff there is an execution of the program that transforms state s to state t. The state space is augmented by a special state $\perp$ that designates the result of a nonterminating execution. Various operators for program composition have direct counterparts in relational calculus: sequential composition corresponds to relational product; a conditional statement, **if** b **then** r **else** s **endif**, corresponds to $(BR \cup B'S)$, where B and B' are the relations corresponding to predicates b and $\neg b$, and R and S are the relations corresponding to r and s. Looping constructs can be modeled by transitive closure.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[3] Martín Abadi and Leslie Lamport. An old fashioned recipe for real time. *TOPLAS*, 16(5):1543–1571, September 1994.

[4] Marc Abrams, Ernest H. Page, and Richard E. Nance. Simulation program development by stepwise refinement in UNITY. In *Proceedings of the Winter Simulation Conference*, pages 233–242, December 1991.

[5] William Adams. *Untangling the Threads: Reduction for a Concurrent Object-Based Programming Model*. PhD thesis, University of Texas at Austin, August 2000.

[6] Yehuda Afek, George Brown, and Michael Merritt. Lazy caching. *TOPLAS*, 15(1):182–206, January 1993.

[7] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 24(4):181–185, 1985.

[8] Lorenzo Alvisi, Rajeev Joshi, Calvin Lin, and Jayadev Misra. Seuss: What the doctor ordered. In *Second International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 284–290, 1999.

[9] Flemming Andersen, Kim Dam Petersen, and Jimmi S. Pettersson. Program verification using HOL-UNITY. In *HUG'93: HOL User's Group Workshop*, volume 780 of *LNCS*, pages 1–17. Springer-Verlag, 1993.

[10] Flemming Andersen, Kim Dam Petersen, and Jimmi S. Pettersson. A graphical tool for proving UNITY progress. In *1994 International Tutorial and Workshop on the HOL Theorem Proving System and Its Applications*, volume 859 of *LNCS*. Springer-Verlag, 1994.

[11] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[12] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1997.

[13] Ralph-Johan R. Back. Refining atomicity in parallel algorithms. In *PARLE 89 Parallel Architectures and Languages Europe*, volume 366 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[14] Ralph-Johan R. Back and Reino Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10:513–554, October 1988.

[15] Ralph-Johan R. Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3(2):73–87, 1989.

[16] Rajive Bagrodia. Process synchronization: design and performance evaluation of distributed algorithms. *IEEE Trans. Software Eng.*, 15(9):1053–1065, 1989.

[17] Valmir Barbosa and Eli Gafni. Concurrency in heavily loaded neighborhood-constrained systems. *TOPLAS*, 11(4):562–584, October 1989.

[18] Kenneth Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, Reston, VA, 1968. AFIPS Press.

[19] Arthur J. Bernstein and Philip M. Lewis. *Concurrency in programming and database systems*. Jones and Bartlett, 1993.

[20] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[21] Robert S. Boyer and J. Strother Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. *Journal of Automated Reasoning*, 4(2):117–172, 1988.

[22] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[23] Per Brinch Hansen. Structured multiprogramming. *CACM*, 15(7):574–577, July 1972.

[24] Per Brinch Hansen. *Operating System Principles*. Prentice Hall, 1973.

[25] J.C. Browne et al. A language for specification and programming of reconfigurable parallel computation structures. In *Int. Conf. of Parallel Processing, Bellaire, Michigan*, pages 142–149. IEEE, August 1982.

[26] Manfred Broy. A logical basis for modular systems engineering. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F*, pages 101–130. IOS Press, 1999.

[27] J. Allen Carruth. Real-time UNITY. Technical Report TR94-10, University of Texas at Austin, April 1994.

[28] J. Allen Carruth and Jayadev Misra. Proof of a real-time mutual-exclusion algorithm. *Parallel Processing Letters*, 6(2):251–257, 1996.

[29] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.

[30] K. Mani Chandy and Jayadev Misra. How processes learn. *Journal of Distributed Computing*, 1:40–52, 1986.

[31] K. Mani Chandy and Jayadev Misra. Systolic algorithms as programs. *Distributed Computing*, 1:177–183, 1986.

[32] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[33] K. Mani Chandy and Jayadev Misra. Proofs of distributed algorithms: an exercise. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, University of Texas at Austin Year of Programming, pages 305–332. Addison-Wesley, 1990.

[34] K. Mani Chandy and Beverly A. Sanders. Predicate transformers for reasoning about concurrent programs. *Science of Computer Programming*, 24:129–148, 1995.

[35] K. Mani Chandy and Beverly A. Sanders. Reasoning about program composition. Technical report, CISE, University of Florida, 2000. Manuscript available for download via `http://www.cise.ufl.edu/ sanders/pubs/composition.ps`.

[36] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming*. Jones and Bartlett, 1992.

[37] M. Charpentier and K. Mani Chandy. Towards a compositional approach to the design and verification of distributed systems. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems, FM '99*, volume LNCS 1708, pages 570–589. Springer-Verlag, 1999.

[38] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[39] Edmund M. Clarke and Ernest Allen Emerson. Design and synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, May 1981.

[40] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[41] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[42] Ernest S. Cohen. *Modular Progress Proofs of Concurrent Programs*. PhD thesis, University of Texas at Austin, August 1992.

[43] Ernest S. Cohen. A guide to reduction. Unpublished note, 1994.

[44] Ernie Cohen and Leslie Lamport. Reduction in TLA. In David Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1998. Compaq SRC Research Note 1998-005.

[45] Pierre Collette. Application of the composition principle to UNITY-like specifications. In *Theory and Practice of Software Development*, volume LNCS 668. Springer-Verlag, 1993.

[46] Pierre Collette. Composition of assumption-commitment specifications in a UNITY style. *Science of Computer Programming*, 23(2-3):107–125, 1994.

[47] Pierre Collette and Edgar Knapp. A foundation for modular reasoning about safety and progress properties of state-based concurrent programs. *TCS*, 183(2):253–279, 1997.

[48] Stephen Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.

[49] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, October 1967.

[50] C. Creveuil and Gruia-Catalin Roman. Formal specification and design of a message router. *ACM Transactions on Software Engineering and Methodology*, 3(4):271–307, October 1994.

[51] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Academic Press, 1972.

[52] Angela Dappert-Farquhar. A correction on a family of 2-process mutual exclusion algorithms: Notes on UNITY: 13-90. Notes on UNITY: 22–90, University of Texas at Austin, 1990.

[53] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001. To appear.

[54] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, 22(6):644–654, 1976.

[55] Edsger W. Dijkstra. Guarded commands, nondeterminacy, and the formal derivation of programs. *Communications of the ACM*, 8:453–457, 1975.

[56] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:83–89, 1959.

[57] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[58] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[59] E.W. Dijkstra. Hierarchical ordering of sequential processes. In *Operating Systems Techniques*. Academic Press, 1972.

[60] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[61] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[62] Rutger M. Dijkstra. Relational calculus and relational program semantics. Eindhoven Institute of Technology, 1992.

[63] Rutger M. Dijkstra. Computation calculus—bridging a formalization gap. *Science of Computer Programming*, 37(1-3):3–36, 2000.

[64] David L. Dill. The murϕ verification system. In *Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, pages 390–393. Springer-Verlag, July 1996.

[65] Thomas W. Doeppner Jr. Parallel program correctness through refinement. In *Proc. 4th Annual ACM Symposium on Principles of Programming Languages*, pages 155–169, January 1977.

[66] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *Real Time Systems*, 4:331–352, 1992.

[67] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[68] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about knowledge*. MIT Press, 1995.

[69] W.H.J. Feijen and A.J.M. van Gasteren. *On a method of multiprogramming*. Monographs in Computer Science. Springer-Verlag, 1999.

[70] M.J. Fischer. Real-time mutual exclusion. Unpublished.

[71] R.W. Floyd. Assigning meanings to programs. In T. Schwartz, editor, *Mathematical Aspects of Computer Science (Proc. Sym. in Applied Math)*, volume 19, pages 19–32. Amer. Math. Soc., 1967.

[72] Nissim Francez. *Fairness*. Springer-Verlag, 1986.

[73] Nissim Francez. *Program Verification*. Addison-Wesley, 1992.

[74] Eli Gafni and Dimitri P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Trans. Communication*, com-29(1):11–18, January 1981.

[75] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, January 1985.

[76] David Moshe Goldschlag. *Mechanically verifying concurrent programs*. PhD thesis, University of Texas at Austin, 1992.

[77] J. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume LNCS 66. Springer-Verlag, 1978. Also appears as *IBM Research Report RJ 2188*, August 1987.

[78] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[79] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[80] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag, 1994.

[81] John Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[82] A. Nico Habermann. Synchronization of communicating processes. *Communications of the ACM*, 15(3):171–176, March 1972.

[83] J.Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990. A preliminary version appeared in *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, 1984.

[84] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, 1998.

[85] E.C.R. Hehner. Another look at communicating processes. Technical Report CSRG-134, University of Toronto, September 1981.

[86] E.C.R. Hehner and C.A.R. Hoare. A more complete model of communicating processes. *Theoretical Computer Science*, 26, September 1983.

[87] Eric C.R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993.

[88] B. Heyd and P. Crégut. A modular coding of UNITY in COQ. In J. Grundy and J. Harrison, editors, *Theorem Proving in Higher Order Logics, TPHOLs '96*, volume LNCS 1125, pages 251–266. Springer-Verlag, 1996.

[89] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580,583, 1969.

[90] C.A.R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[91] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[92] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1984.

[93] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1997.

[94] J.D. Ichbiah et al. Rationale for the design of the Ada programming language. *SIGPLAN Notices, Part B*, 14(6), June 1979.

[95] Michael Jackson. *Software Requirements & Specifications*. Addison-Wesley, 1995.

[96] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.

[97] C.B. Jones. *Development Methods for Computer Programs Including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as Technical Monograph 25, Programming Research Group.

[98] Rajeev Joshi. Seuss for Java: language reference. University of Texas at Austin, February 1998.

[99] Rajeev Joshi. *Immediacy: A Technique for Reasoning about Asynchrony*. PhD thesis, University of Texas at Austin, December 1999.

[100] Rajeev Joshi and Jayadev Misra. On the impossibility of robust solutions for fair resource allocation. Technical Report CS-TR-99-14, University of Texas at Austin, Department of Computer Sciences, June 1999. Available for download as `ftp://ftp.cs.utexas.edu/pub/techreports/tr99-14.ps.Z`.

[101] Rajeev Joshi and Jayadev Misra. Maximally concurrent programs. *Formal Aspects of Computing*, 12(2):100–119, 2000.

[102] Charanjit S. Jutla and Josyula R. Rao. A methodology for designing proof rules for fair parallel programs. *Formal Aspects of Computing*, 9(4):359–378, 1997.

[103] C.S. Jutla, E. Knapp, and J.R. Rao. A predicate transformer approach to semantics of parallel programs. In *Proc. 8th ACM SIGACT/SIGOPS Symposium on Principles of Distributed Systems (PODC '89)*, pages 249–263, Edmonton, Alberta, Canada, 1989.

[104] Markus Kaltenbach. *Interactive Verification Exploiting Program Design Knowledge: A Model-Checker for UNITY*. PhD thesis, University of Texas at Austin, 1996. Available for download at `http://www.cs.utexas.edu/users/markus/diss.html`.

[105] M. Kaufmann, P. Manolios, and J. Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[106] M. Kaufmann, P. Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.

[107] E. Knapp. A predicate transformer for progress. *Information Processing Letters*, 33:323–330, 1990.

[108] E. Knapp. *Refinement as a Basis for Concurrent Program Design*. PhD thesis, University of Texas at Austin, May 1992.

[109] Ingolf H. Krüger. An experiment in compiler design for a concurrent object-based programming language. Master's thesis, University of Texas at Austin, 1996.

[110] H.T. Kung and C.E. Leiserson. Systolic arrays (for VLSI). In I. S. Duff and G. W. Stewart, editors, *Sparse Matrix Proc., 1978*, pages 256–282. SIAM, 1979.

[111] S.S. Lam and A.U. Shankar. Refinement and projection of relational specifications. In *Proc. REX Workshop on Stepwise Refinement of Distributed Systems, Plasmolen-Mook, The Netherlands*. LNCS 430, Springer-Verlag, May 1989.

[112] S.S. Lam and A.U. Shankar. A theory of interfaces and modules I: composition theorem. *IEEE Transactions on Software Engineering*, 20(1):55–71, January 1994.

[113] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, SE-3(2):125–143, March 1977.

[114] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668. IFIP, North-Holland, September 1983.

[115] Leslie Lamport. Basic concepts. In M. Paul and H.J. Siegert, editors, *Distributed Systems-Methods and Tools for Specification*, volume LNCS 190. Springer-Verlag, 1985.

[116] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[117] Leslie Lamport. *win* and *sin*: Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems*, 12(3):396–428, 1990.

[118] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[119] Leslie Lamport. Specifying concurrent systems with TLA+. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, pages 183–247. IOS Press, 1999.

[120] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical Report 44, Digital Systems Research Center, May 1989.

[121] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis,3/e.* McGraw Hill, 2000.

[122] D. Lehman, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: the ethics of concurrent termination. In O. Kariv and S. Even, editors, *Proc. 8th ICALP*. LNCS 115, Springer-Verlag, July 1981.

[123] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[124] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[125] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions.* Morgan Kaufmann, 1994.

[126] Saunders MacLane and Garrett Birkhoff. *Algebra.* Macmillan, 1970.

[127] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer-Verlag, 1992.

[128] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems, Safety.* Springer-Verlag, 1995.

[129] Alain Martin. A general proof rule for procedures in predicate transformer semantics. *Acta Informatica*, 20:301–313, 1983.

[130] José Meseguer. Rewriting logic and Maude: Concepts and applications. In *Proc. RTA 2000*, volume 1833 of *LNCS*, pages 1–26. Springer-Verlag, 2000.

[131] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[132] R. Milner. *Communication and Concurrency*. International Series in Computer Science, C.A.R. Hoare, series editor. Prentice-Hall International, 1989.

[133] Jayadev Misra. Distributed discrete event simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[134] Jayadev Misra. Notes on UNITY, a series. University of Texas at Austin, 1988–1992. Available for download at `http://www.cs.utexas.edu/users/psp/notesunity.html`.

[135] Jayadev Misra. A foundation of parallel programming. In Manfred Broy, editor, *Proc. 9th International Summer School on Constructive Methods in Computer Science*, volume F 55 of *NATO ASI Series*, pages 397–433. Springer-Verlag, 1989.

[136] Jayadev Misra. Specifying concurrent objects as communicating processes. *Science of Computer Programming*, 14(10):159–184, October 1990.

[137] Jayadev Misra. Loosely coupled processes. *Future Generation Computer Systems*, 8:269–286, 1992. North-Holland.

[138] Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[139] Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[140] Jayadev Misra. A walk over the shortest path: Dijkstra's algorithm viewed as fixed-point computation. *Information Processing Letters*, 77(2–4):197–200, February 2001.

[141] Jayadev Misra. A simple, object-based view of multiprogramming. *Formal Methods in System Design*, 2001, to appear.

[142] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE*, SE,7(4):417–426, July 1981.

[143] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *TOPLAS*, 18(3):325–353, May 1996.

[144] P. Newton and J.C. Browne. The code 2.0 graphical parallel programming language. In *Proc. ACM Int. Conf. on Supercomputing*, July 1992.

[145] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(1):319–340, 1976.

[146] S. Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[147] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[148] J. Pachl. A simple proof of a completeness result for *leads-to* in the UNITY logic. *Information Processing Letters*, 41:35–38, 1992.

[149] Lawrence C. Paulson. Mechanizing a theory of program composition for UNITY. Preliminary version, August 2000.

[150] Lawrence C. Paulson. Mechanizing UNITY in Isabelle. *ACM Trans. on Computational Logic*, 1(1):3–32, July 2000.

[151] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[152] A. Pizzarello. An industrial experience in the use of UNITY. In J. P. Banâtre and D. Le Métayer, editors, *Proc. Research Directions in High-level Parallel Programming Languages*, volume 574, pages 39–49, Mont Saint-Michel, France, 1991. Springer-Verlag.

[153] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *5th International Symposium on Programming*, volume 137 of *LNCS*. Springer-Verlag, April 1982.

[154] V. Ramachandran, B.Grayson, and M. Dahlin. Emulations between QSM, BSP and LogP: A framework for general-purpose parallel algorithm design. Technical Report TR98-22, CS Dept., University of Texas at Austin, November 1998. Summary in Proc. ACM-SIAM SODA'99.

[155] Josyula R. Rao. *Extensions of the UNITY Methodology: Compositionality, Fairness and Probability in Parallelism*, volume 908 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[156] J.R. Rao. On a notion of completeness for the *leads-to*. Notes on UNITY: 24–90, July 1991.

[157] R.L. Rivest, A. Shamir, and L. Adelman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[158] Beverly A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3:189–205, 1991.

[159] R.A. Scantlebury, K.A. Bartlett, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, May 1969.

[160] Fred Schneider, Bard Bloom, and Keith Marzullo. Putting time into proof outlines. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proc. of the REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *LNCS*, pages 618–639. Springer-Verlag, 1991.

[161] Fred B. Schneider. *On Concurrent Programming*. Springer-Verlag, New York, 1997.

[162] Natarajan Shankar. Verification of real-time systems using PVS. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 280–291, June–July 1993.

[163] A.K. Singh. A theorem relating *leads-to* and *unless*. Notes on UNITY: 04–88, December 1988.

[164] A.K. Singh. *Leads-to* and program union. Notes on UNITY: 06–89, June 1989.

[165] M.G. Staskauskas. *Specification and Verification of Large-Scale Reactive Programs*. PhD thesis, University of Texas at Austin, May 1992.

[166] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[167] J. van de Snepscheut. Personal communication.

[168] A.J.M. van Gasteren and G. Tel. On the proof of a distributed algorithm. *Information Processing Letters*, 35(6), 1990. Letter to the editor.

Index