Copyright

by

William Edward Adams

2000

Untangling the threads: reduction for a concurrent object-based programming model

by

William Edward Adams, BA, MS

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2000

Untangling the threads: reduction for a concurrent object-based programming model

Approved by
Dissertation Committee:

____

____

____

____

____

____

To Beverly

Acknowledgments

The time I have spent at the University of Texas has been the most enriching and stimulating period of my life. The Computer Sciences Department has the fortunate combination of excellent faculty, outstanding graduate students, and helpful support staff. Jayadev Misra has proved to be an excellent advisor. His insights have provided the foundations for my work, and I hope I have learnt some of his ability to reach the core of a problem, focusing on the important elements, and ignoring irrelevant details. Professors Allen Emerson and Edsger Dijkstra have also been major influences on my development as a computer scientist. I have learnt much from my discussions with various members of Dr Misra's research groups, in particular Ernie Cohen, Rajeev Joshi, Markus Kaltenbach, Jacob Kornerup, Al Carruth, and J R Rao. Gloria Ramirez in the Graduate Office was the first person I met in the Department, and she has been a regular source of information and help ever since.

Attending graduate school requires not only intellectual curiosity, but also financial support. This work was supported in part by National Science Foundation Award CCR–9803842, and by a research assistantship at the Computer Engineering Research Center, funded by Fujitsu Laboratories of America. Summer employment, with Schlumberger Laboratory for Computer Science, Microsoft Corporation, IBM T J Watson Research Center, and Fujistsu Laboratories of America, has provided income, and a welcome opportunity to travel around the United States. In the past

two years, I have been employed by IBM Austin Research Laboratory. The support, both material and moral, that I have received there has been invaluable in the final stages of this work. I thank Sani Nassif and Warren Hunt, Jr, in particular.

Finally, this dissertation would not have been written without the forbearance of my wife, Beverly, and my stepson, Summer. They have shown uncommon love and tolerance during the countless hours that I have spent writing.

William Edward Adams

The University of Texas at Austin
August 2000

Untangling the threads: reduction for a concurrent object-based programming model

Publication No. ____________________

William Edward Adams, Ph.D.
The University of Texas at Austin, 2000

Supervisor: Jayadev Misra

Reduction is a technique for simplifying reasoning about systems where a set of sequential executions, which we call *threads*, are executed concurrently. In a concurrent execution, steps from different threads are interleaved. A reduction theorem shows that a concurrent execution is equivalent to an execution in which all the steps of a given thread appear contiguously. These steps can be replaced by a single atomic step representing the complete execution of the thread.

Applying reduction to each thread in turn, we reduce a concurrent execution to an atomic execution, where every step is an atomic step. Reasoning about the atomic execution is significantly simpler than reasoning about the original concurrent execution. In the atomic execution, we do not need to consider the interactions of steps from different threads.

We describe a model for concurrent systems, called *Seuss*. In this model, a program is a set of independently executing boxes, which communicate using a form of remote procedure call. We show how to reduce a concurrent execution of a Seuss program to an atomic execution. Since every concurrent execution is equivalent to

an atomic execution, we can understand all possible executions of a Seuss program by understanding just its atomic executions.

We show three main results. One gives sufficient conditions to guarantee that all threads in an execution terminate. The other two concern reduction of executions in which all threads terminate.

We express the reduction results relative to restrictions on which pairs of threads can run concurrently. The first reduction theorem shows a restriction on concurrency that guarantees that every execution can be reduced to a sequential execution. The second reduction theorem gives a stronger restriction on concurrency (so less concurrency is allowed in an execution), but, in return, gives a reduced sequential execution that has stronger fairness properties (and thus better progress properties) than we get with the first theorem.

To show these results, we use an operational semantics for a simple Seuss language. The semantics is such that our results apply to any language implementing the Seuss model.

Contents

Acknowledgments v

Abstract vii

List of Tables xvi

List of Figures xvii

Chapter 1 Introduction 1

1.1 A model for concurrent systems 3

1.1.1 Action systems 3

1.1.2 The Seuss model 5

1.1.3 Partial and total procedures 7

1.1.4 Atomic executions of Seuss programs 11

1.1.5 Mutual exclusion on boxes 13

1.1.6 Concurrent execution in Seuss 14

1.2 Reduction 17

1.2.1 The two-phase locking protocol 18

1.2.2 Reduction for Seuss 20

1.3 Summary of the material 24

1.3.1 Chapter overview 25

1.4 Related work 26
1.4.1 Seuss 26
1.4.2 Reduction 27
1.4.3 Transaction processing 29
1.4.4 Concurrent object-based languages 30
1.5 Notation 31

Chapter 2 The definition of *TCB* **36**

2.1 Introduction 36
2.2 Syntax 37
2.2.1 Variables, states, values and expressions 37
2.2.2 Syntax for *TCBloc* 38
2.3 Syntax for *TCB* 41
2.4 Well-formed programs 43
2.5 Run-time errors 45
2.6 Operational semantics 45
2.7 Operational semantics for *TCBloc* 49
2.7.1 Program configuration for *TCBloc* 49
2.7.2 Semantic rules for *TCBloc* 49
2.8 Defining the semantics for *TCB* 52
2.8.1 Modelling boxes 52
2.9 Rendezvous semantics for *TCBtot* 54
2.9.1 Static information for a *TCBtot* box 54
2.9.2 Static infomation for a *TCBtot* program 56
2.9.3 Box configuration for *TCBtot* 57
2.9.4 Program configuration for *TCBtot* 60
2.9.5 Semantic rules for *TCBtot* 60
2.10 Rendezvous semantics for *TCB* 64

2.10.1 Static infomation for a *TCB* box . . . 64
2.10.2 Static information for a *TCB* program . . . 67
2.10.3 Box configuration for *TCB* . . . 68
2.10.4 Program configuration for *TCB* . . . 71
2.10.5 Rendezvous semantic rules for *TCB* . . . 71
2.11 Queue semantics for *TCB* . . . 74
2.11.1 Call queues . . . 76
2.11.2 Box and program configurations for the queue semantics . . . 77
2.11.3 Queue semantic rules for *TCB* . . . 78
2.12 Summary . . . 82
2.12.1 Run-time errors . . . 82

Chapter 3 Execution of *TCB* programs **83**
3.1 Introduction . . . 83
3.2 Box configurations . . . 84
3.3 Program Configurations . . . 87
3.3.1 Relations induced by calls . . . 90
3.3.2 Well-formed program configurations . . . 94
3.3.3 Call stacks . . . 94
3.3.4 Wait lines . . . 96
3.3.5 Persistent states . . . 98
3.4 Program steps . . . 99
3.4.1 Steps and configurations . . . 99
3.4.2 Step labels . . . 100
3.4.3 Enabled steps . . . 109
3.4.4 Enabled steps for a call stack . . . 115
3.5 Executions . . . 116
3.6 Discussion . . . 119

3.6.1 Deterministic and nondeterministic steps 120
3.6.2 Threads 122

Chapter 4 Complete executions 123

4.1 Introduction 123
4.1.1 Procedure sets 125
4.2 Proper executions and complete executions 126
4.3 Deadlock 129
4.4 Infinite procedure calls 132
4.5 Executions with a finite number of threads 135
4.6 Thread fairness 136
4.6.1 Fairness for rendezvous procedure calls 137
4.7 The complete execution theorem 138
4.8 Control relations 142
4.8.1 Implementing control relations 144
4.9 Avoiding deadlock 145
4.9.1 Nonblocking control relations 147
4.10 Avoiding infinite threads 152
4.11 Implementing thread fairness 153
4.12 Summary 154
4.12.1 Avoiding run-time errors 154

Chapter 5 Reduction 155

5.1 Introduction 155
5.1.1 Right-movers and left-movers in *TCB* 156
5.1.2 Transforming an execution 157
5.2 Compound steps 158
5.2.1 Rendezvous calls 159

5.2.2 Atomic steps . . . 161
5.2.3 Executions with compound steps . . . 167
5.3 Step types . . . 170
5.3.1 Decision steps . . . 170
5.3.2 Right-movers and left-movers . . . 172
5.3.3 The format of a thread . . . 173
5.3.4 A strategy for reduction . . . 175
5.4 Reduction relations . . . 176
5.4.1 Accept decision steps . . . 176
5.4.2 Similar executions . . . 177
5.4.3 Swapping steps . . . 179
5.4.4 Reduce-equivalence . . . 181
5.4.5 Finite reduction . . . 183
5.4.6 The reduction relation . . . 186
5.5 Relating concurrent and sequential configurations . . . 190
5.6 Reduction rules . . . 195
5.7 Reduction rule for **tm** steps . . . 200
5.7.1 Weak compatibility . . . 203
5.7.2 Reduction that respects control relations . . . 207
5.8 The first reduction theorem . . . 208
5.8.1 Outline of the proof . . . 208
5.8.2 Proof of Lemma 5.58 . . . 210

Chapter 6 Fairness **218**

6.1 Introduction . . . 218
6.2 Fairness conditions for *TCB* . . . 221
6.2.1 Weak fairness . . . 222
6.2.2 Minimal fairness . . . 223

6.3 Program properties 224
6.4 Fairness for weak compatibility 229
6.5 Strong compatibility 232
6.6 The second reduction theorem 237
6.7 Scheduling *TCB* programs 241

Chapter 7 Conclusions **245**
7.1 Summary of the main results 245
7.2 Future work 246
7.2.1 Reasoning about Seuss programs 246
7.2.2 Concurrent termination 247
7.2.3 Negative alternatives 247

Appendix A Semantics for *TCB* languages **248**
A.1 Rendezvous semantics for *TCBtot* 248
A.2 Rendezvous semantics for *TCB* 250
A.3 Queue semantics for *TCB* 253

Appendix B Additional proofs **257**
B.1 Proofs for Chapter 3 257
B.1.1 Proof of Theorem 3.33 257
B.1.2 Proof of Theorem 3.51 262
B.1.3 Proof of Theorem 3.14 266
B.1.4 Proof of Theorem 3.15 267
B.1.5 Proof of Theorem 3.16 269
B.2 Proofs for Chapter 4 271
B.2.1 Proof of Theorem 4.7 271
B.2.2 Proof of Theorem 4.8 272
B.2.3 Proof of Theorem 4.12 274

B.2.4 Proof of Theorem 4.14 280
B.3 Proofs for Chapter 5 286
B.3.1 Proof of Theorem 5.56 286

Bibliography **288**

Vita **292**

List of Tables

3.1 Conditions on the box phase for one-locus rules. 101

3.2 Conditions on the box phase for two-locus rules. 103

3.3 Conditions on **C** to enable a conditional step for D. 112

5.1 Configurations for a method call from D to E 159

List of Figures

1.1 A producer-consumer program 6
1.2 Code for box *Buff* 7
1.3 Boxes *Prod* and *Cons* 9
1.4 Partial procedure with alternatives 10
1.5 Program with commuting and noncommuting methods 21

2.1 *TCBloc* code to divide x by d 39
2.2 Syntax for *TCBloc* 40
2.3 A *TCB* program with three boxes 41
2.4 Syntax for *TCB* 42
2.5 Semantics for *TCBloc* 50
2.6 Transition diagram for phases of a *TCBtot* box. 58
2.7 Semantics for *TCBtot*: action start and method call 61
2.8 Semantics for *TCBtot*: procedure body execution. 62
2.9 Semantics for *TCBtot*: procedure return 63
2.10 Transition diagram for phases of a *TCB* box. 69
2.11 Rendezvous semantics for *TCB*: starting and rejecting a partial action call 72
2.12 Rendezvous semantics for *TCB*: guard evaluation 73
2.13 Rendezvous semantics for *TCB*: test return 75

2.14 Queue semantic rules for *TCB*: procedure call 80

2.15 Queue semantic rules for *TCB*: procedure initialization 81

4.1 Program that can deadlock 129

4.2 Program with nonterminating procedure 133

5.1 Adding a probe to a program 195

6.1 Program with fairness-dependent unavoidability properties 227

6.2 The semaphore with a kill action 229

Chapter 1

Introduction

Implementing concurrent systems that function correctly is a difficult task. To understand the behaviour of two sequential programs executing concurrently, we must understand not only the behaviour of each separately, but also the ways in which they interact.

We can understand an execution of a sequential program as a transformation from its start state to its final state. In this view, the intermediate states are ignored. We separate *what* the program does, from *how* it does it. If we run two sequential programs one after the other, we can determine the transformation effected by the execution as a whole from the transformations for the separate programs.

There is not the same separation between what is done and how it is done when we consider concurrent execution of sequential programs. Consider the following programs.

$$\begin{array}{lll} \alpha & :: & x := x + 1 \, ; \, x := x + 1 \\ \alpha' & :: & x := x * 2 \end{array}$$

Program α adds 2 to x, and program α' doubles x. Executing the programs one after the other from a state where $x = 2$, we reach a state where $x = 8$ for the

execution $\alpha;\alpha'$, and a state where $x = 6$ for the execution $\alpha';\alpha$. Consider now executing the programs concurrently. We assume that each statement is atomic, so the interference occurs only if the statement from α' occurs between the two statements from α. In this case, $x = 7$ in the final state.

To understand the concurrent execution of α and α', we must understand the internal workings of each, and consider ways that they may interfere. Here the doubling of x by α' overwrites an intermediate result written by α to x, and so α' interferes with α's execution.

Note that we can replace the two assignments in α with a single assignment adding 2 to x, and the above interfering computation is no longer possible. The amount of interference possible depends on the internal structure of the programs.

There are short programs, so there is a limited number of interleavings of their statements. For longer programs, there is a larger number of interleavings, and consequently more opportunity for interference.

Note that not all sequential programs interfere when run concurrently. Programs that access disjoint parts of the state space do not interfere. Neither do certain programs that access shared variables. Consider running two copies of α concurrently. The overall effect is to add 4 to x, regardless of the interleaving of the statements. For noninterfering programs, running the programs concurrently is equivalent to running them sequentially, in some order.

We propose a model for concurrent systems, called *Seuss*, in which we separate the transformational aspects of the sequential programs from the issues of interference between programs. A Seuss program is a set of individual sequential programs, called *actions*, that access a shared state. An execution of a program consists of repeatedly choosing one of the actions and executing it to completion. One action is executed at a time, so we can understand the effect of each execution of an action without considering internal details. This gives us a simple model for

proving properties of programs.

In the implementation of Seuss, we allow actions to execute concurrently. But we do so in a controlled way. As we saw above, allowing any two actions to run concurrently allows interference. We allow only actions that do not interfere with each other to execute concurrently.

Building a system in Seuss follows the following outline. First, we define the actions, and show that a sequential execution of the program has the desired properties. Then we examine each pair of actions and decide if they are interfering or noninterfering. We implement the program so that only noninterfering actions run concurrently.

Our work addresses the second part of this outline. We define the Seuss model, and we give a condition on pairs of actions that guarantees noninterference. We show that for any execution where only noninterfering actions run concurrently, there is a sequential execution with the same behaviour.

1.1 A model for concurrent systems

1.1.1 Action systems

A sequential program α can be specified by a set of assertions, called *Hoare triples* [16], of the form $\{P\}\,\alpha\,\{Q\}$, where P and Q are predicates on the state. This assertion is true if every execution of α that starts from a state satisfying P ends in a state satisfying Q. We call P the *precondition*, and Q the *postcondition*. In the transformational view of sequential programs, two programs are equivalent if there is no assertion satisfied by one that is not satisfied by the other.

Representing sequential programs by the assertions they satisfy discards information about the internal computations of the program. So, for example, if α satisfies the Hoare triple $\{P\}\,\alpha\,\{Q\}$, where P is "A is an array of integers", and Q

is "A contains the same values as before, in sorted order", then this assertion gives no information about the algorithm used to implement the sort, or any intermediate values that array A takes during the execution of the program. Any program satisfying the assertion is equivalent for our purposes.[1]

An *action system* consists of a set of terminating sequential programs, called *actions*, that read and write a set of shared variables. A single execution of an action is called a *thread*. An execution of an action system is a sequence of threads.

An action system represents a simple concurrent system. Here, concurrency is represented by alternating threads for different actions. For example, consider a system with two actions, *Produce* and *Consume*. A thread for the former produces an item, and a thread for the latter consumes an item. An execution of this system consists of a sequence of threads, some for *Produce*, and some for *Consume*. We regard this as representing the concurrent execution of a producer process, which repeatedly executes action *Produce*, with a consumer process, which repeatedly executes action *Consume*. The interleaving in an execution of an action system is at the granularity of complete threads.

To prove properties of the executions of an action system, we first consider each action separately, and show a set of assertions satisfied by a single thread for the action. Given this, we can represent the complete execution of a thread as a single step, which satisfies the Hoare triples for the corresponding action. We call the step representing a complete thread an *atomic* step.

An *atomic* execution is an execution where every thread is represented by an atomic step. The properties of an atomic executions are derived from the assertions satisfied by the atomic steps.

Here there is a clean separation of concerns. We examine the internal structure of each action separately to determine the set of Hoare triples it satisfies. During

[1]Of course, the choice of sorting algorithm has a major impact on the time taken for the program to complete execution, but this issue does not concern us here.

this process, we must consider the states internal to a thread's execution, but the resulting assertions are given in terms of the start and final state alone. Having done this, we consider an atomic execution, in which all states internal to the execution of a thread have been elided.

The actions are terminating sequential computations, whereas an execution of an action system is generally nonterminating. The above scheme separates the sequential, terminating aspects of the execution of a single thread from the concurrent, nonterminating aspects of the overall system execution.

1.1.2 The Seuss model

We define a programming model, called *Seuss* [25], based on action systems. We extend the action system model in two ways. We distribute the state space and the code of the actions across a set of *boxes*, and we allow for concurrent execution of threads. We consider the distribution of the data and code first.

Boxes are rudimentary objects. A box contains some variables, and some procedures. A variable in a box can be read or written only by a procedure in that box. The procedures are of two types: *actions* and *methods*. Actions are top-level procedures. In this model, a thread is the complete execution of an action call. The code of a procedure contains statements reading and writing variables local to its box, and calls to methods on other boxes. Actions are not called by other procedures. Methods may take parameters, both input and output, but actions take none.

We use compound identifiers for the variables and procedures in a box, with the box name as part of the identifier. Thus $D.x$ is the name of a local variable x in box D, and $D.a$ is the name of an action a in D.

A thread for $D.a$ starts executing the code of a at box D. If the execution reaches a method call statement for method $E.m$, then execution at D is suspended

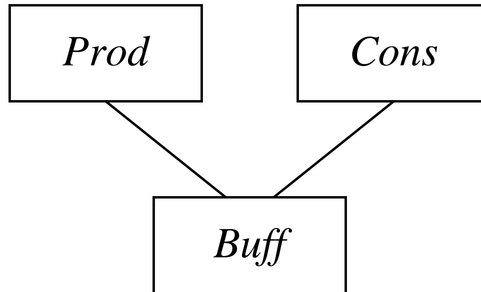

Figure 1.1: A producer-consumer program

while the code for m is executed at E. Execution continues at D when the execution of $E.m$ completes.

Consider a simple example of such a program with three boxes, shown pictorially in Figure 1.1. This program contains three boxes, called *Prod*, *Cons*, and *Buff*. The program represents a simple producer-consumer system. The box *Prod* produces some items which are consumed by box *Cons*. A buffer, implemented by box *Buff*, holds items that have been produced by *Prod*, but not yet consumed by *Cons*. We assume that *Buff* implements a buffer of unbounded size that obeys a first-in-first-out (FIFO) discipline.

The line in the diagram from box *Prod* to box *Buff* indicates that procedures in *Prod* call methods in *Buff*, and similarly for the line from *Cons* to *Buff*. There are no method calls between boxes *Prod* and *Cons*.

Box *Buff* has two methods, and no actions. Method $Buff.put$ takes a single item as an input parameter. The call $Buff.put(x)$ adds item x to back of the buffer. Method $Buff.get$ returns a single item as an output parameter. A call $Buff.get(y)$ removes the front item from the buffer and returns it in y.

Box *Prod* has one action, and no methods. Action $Prod.make$ produces an item, and calls $Buff.put$ to put this item in the buffer. Box *Cons* similarly has a single action, and no methods. Action $Cons.use$ calls $Buff.get$ to retrieve an item

```
box Buff
   var s : sequence of integer
   method put(in x : integer)
                 ::   s := s ◁ x
   method get(out x : integer)
                 ::   s ≠ ⊥ ⟶ x := first(s) ; s := rest(s)
end
```

Figure 1.2: Code for box *Buff*

from the buffer, and then consumes this item.

1.1.3 Partial and total procedures

The first question that arises is what to do in the case that a thread for *Cons.use* is called when there is no item in the buffer. In this case the call to *Buff.get* cannot return an item. We handle this by putting a *guard* in the code for *Buff.get*. The guard contains a predicate on the local state. The code for box *Buff* is shown in Figure 1.2. For simplicity, we assume that the buffer holds integer items. The code for *Buff* declares a variable s, which is a sequence of integers, representing the contents of the buffer. The header for method *put* declares an input integer parameter. We use :: to separate the header from the procedure body. The body of this method consists of a single assignment that appends the input parameter to the end of s. We use the operator $\triangleleft$ to add an item to the end of a finite sequence. The header for method *get* declares an output integer parameter. The code of this procedure contains a guard and a couple of assignments. We use $\longrightarrow$ to separate the guard from the statements. The guard is a check that the buffer is nonempty. We use $\perp$ for the empty sequence. The assignment statements assign the first element of s to the output parameter, and remove this item from s. We use $\mathrm{first}(X)$ and $\mathrm{rest}(X)$ for the first element of sequence X, and the remaining elements, respectively.

The assignment statements for *get* can only be executed if the predicate in the guard is true. If *Buff.get* is called when s in nonempty, we say that the call *accepts*. In this case, the assignments are executed, and an item is returned to the box that called the procedure. If *Buff.get* is called when s is empty, we say that the call *rejects*. In this case, the assignments are not executed, and no item is returned to the box that called the procedure.

Consider now the action *Cons.use*. This has a call to *Buff.get* to retrieve an item. If the call accepts, then the action proceeds to consume the item returned by the call, but if the call rejects, then there is no item to consume, and the action cannot proceed.

We make a rule that if, during the execution of a procedure call, a call to a method rejects, then the procedure call rejects as well. We further require that, as with *Buff.get*, a rejecting call to a procedure makes no change to the program state.

The method *Buff.put* has no guard, since it is always possible to append an item to an unbounded buffer. A call to this method always accepts. Note that if we implement a bounded buffer, method *Buff.put* may reject, since an item cannot be added to a full buffer.

We call a procedure that never rejects *total*, and one that may reject *partial*. From this definition and the above rule, a total procedure calls only total methods.

To ensure that rejecting procedure calls do not change the system state, we allow only the first method call in the execution of a partial procedure call to be a call to a partial method. All method calls other than the first are to total methods. This gives only two ways that a partial procedure call can reject. The first is if the local predicate in the guard is false, and the second is if the (single) partial method call rejects. Both these occur before any change is made to the system state, in particular, before any calls to total methods.

```
box Prod
   var i : integer
   action make  ::  (* ··· *) ; Buff.put(i)
end

box Cons
   var k : integer
   action use  ::  true & Buff.get(k) ⟶ (* ··· *)
end
```

Figure 1.3: Boxes *Prod* and *Cons*

The code for boxes *Prod* and *Cons* is shown in Figure 1.3. The code for action *Prod.make* contains no guard, since the action is total. The commented ellipsis represents the omitted code that sets the value of i. The final statement is a call to *Buff.put*. The code for action *Cons.use* contains a guard which has two components, separated by & . The first component is a predicate on the local state, as before. We call this the *condition*. In this case, the condition is the predicate *true*. The second component is a call to a partial method. We call this the *test*.[2] Here, the test is a call to *Buff.get*. If this accepts, the action call continues to execute the statements after the guard (again, we have omitted these), and itself accepts. There are no calls to partial methods in the statements after the guard. If the test rejects, then the action call rejects without executing further.

The structure of partial procedures ensures that no local variable is updated unless a procedure is accepting. This enforces the requirement that a rejecting call leave the state unchanged.

Models for transaction processing (see [14]) also have this notion of accepting and rejecting, under the names *committing* and *aborting*. During the execution of a

[2] In [25], the components of the guard are called the *precondition* and the *preprocedure*, respectively.

```
box Cons2
  var j : integer
      k : integer
      b : boolean init false
  action use  ::  ¬b & Buff0.get(j)  ⟶ b := true
                | b & Buff1.get(k)   ⟶ (∗ ··· ∗) ; b := false
end
```

Figure 1.4: Partial procedure with alternatives

transaction, the changes that are made to the system state are tentative. When a transaction commits, all the changes are made permanent. A transaction aborts if it discovers that a required resource is not available, so it is unable to complete its execution. If it aborts, all the tentative changes made up to the point of abortion are discarded. Implementing this often requires complex mechanisms for rolling back the state of a system to a previous consistent state. In contrast, the restriction we give, that the decision to accept or reject be made before the state is altered, means we do not have to provide for rollback.

The restriction, however, means that we cannot write an action that requires two independent resources. We extend the syntax of partial procedures to provide for this. We allow a partial procedure to be written as a set of *alternatives*. In Figure 1.4 we show a partial procedure with two alternatives. Box *Cons2* is part of a program that contains boxes *Buff0* and *Buff1*, both of which are copies of box *Buff*. Action *Cons2.use* requires an item from each buffer before it can execute. The items are retrieved one at a time, using calls to *Buff0.get* and *Buff1.get*. The action has two alternatives, separated by |. One retrieves an item from *Buff0*, and the other retrieves an item from *Buff1*, and then consumes both items. The items are retrieved into variables j and k. We use a boolean variable b to sequence the retrieval of the items. Initially, b is *false*, and it is *true* only when the first item has been retrieved but the second has not.

The first alternative accepts if b is *false*, and an item is successfully retrieved from *Buff0* into item j. If it accepts, b is set to *true*. The second alternative accepts if b is *true*, and an item is successfully retrieved from *Buff1* into item k. If it accepts, the items are consumed, and then b is set to *false* again. It takes two accepting calls to *Cons.use2* to retrieve and consume two items.

We allow any number of alternatives in a partial procedure. Each alternative has a guard, which contains a condition, and, optionally, a test. We require that the conditions on any pair of alternatives be disjoint, so that, in any state, the condition holds for at most one alternative. The execution of a partial procedure is described by the following algorithm.

- If no alternative has a condition that holds in the current state, then reject.
- Otherwise, choose the (single) alternative with a condition that holds in the current state.
- If this alternative has a test, then call the test.
- If the test rejects, then reject.
- Otherwise, if there no test, or the test accepts, execute the body of the alternative, and accept.

1.1.4 Atomic executions of Seuss programs

The programming model presented above is a form of action system. To reason about atomic executions of a Seuss program, we use the separation described above between the sequential properties of the individual actions, and the concurrent properties of the atomic execution.

The encapsulation of data and code in boxes is intended as an aid to the proving sequential properties of the individual actions. A box such a *Buff* that

contains no method calls can be specified as an *abstract data type* [1]. That is, we define the methods it provides, and how each updates the state of the box.

This abstract data type viewpoint supports hierarchical reasoning about the behaviour of an action. We prove sequential properties for the methods of *Buff*. We then do the same for the actions in *Prod* and *Cons*, regarding each call to a method in *Buff* as a single step satisfying the same assertions as the method.

Execution of action *Prod.make* updates the state of *Prod* and *Buff*. A complete specification of this action gives the changes to both boxes. In general, an action can change the state of any box where a procedure is executed during a thread for the action. But the box structure provides a way to divide a program into parts, where each part consists of one or a few boxes, and the interface between the parts is well-defined. The model facilitates compositional reasoning.

The code given for box *Buff* is itself an abstraction of a real implementation of a buffer. It serves as a specification of the behaviour we expect from a buffer. We can use *refinement* [17] to define more detailed and realistic code. A real implementation of a buffer may best be expressed as a set of boxes, one of which provides the *Buff* interface. We can refine the data and code within a box, and we can define sets of boxes that are refinements of a single box.

As noted above, we reason about the atomic execution in terms of the specifications for the individual actions. The techniques for reasoning about such executions have been extensively studied. Various forms of temporal logic (see [11]) can be used to express and derive properties. The application of these techniques to atomic executions in the Seuss model has been less widely studied, though there is some promising preliminary work.

The work presented here does not address these issues, important though they are. The main conclusion we draw from this discussion is that developing a system for reasoning about atomic executions of Seuss system appears to be a

tractable problem, and the structure of the Seuss model supports many of the reasoning styles that have proven useful in other programming models.

1.1.5 Mutual exclusion on boxes

An abstract data type is often specified, in part, by an *invariant.* This is a predicate which holds at the start of the execution, and which holds after every execution of a method on the box. For box *Buff*, suppose P is the sequence of values passed as values to *put*, and G is the sequence of values returned by accepting calls to *get*, then an invariant is

$$P \quad = \quad G \circ s$$

Here $\circ$ is the append operator on sequences.

To show that an invariant holds for a box, we show that if it holds before a call to a procedure in the box, it holds after the call. We do not require that in invariant be maintained during the execution of the call, only that it be reëstablished at the end. For this reason, it is important that the execution of a procedure call at a box is not interrupted, since the box's state may not satisfy the invariant.

We make the rule that once a box starts executing a procedure call, it does not start another procedure call until the first is complete. A box encapsulates *control*, by ensuring mutual exclusion on the execution of its procedures. It is a form of *monitor* [18].

Consider the execution of a single thread, where the thread starts executing at box D. Suppose that during the execution, there is a call to a method on box E, and during the execution of this method call, there is a call to a method on box D. At this point, D is part way through executing a procedure call, and it cannot execute the method call from E until this procedure call completes. But the procedure call cannot complete until the method call is executed. Thus box D is

stuck, waiting for itself. This is *deadlock*.

We call a method call by a thread to a box already executing for a thread a *cyclic call*. One implication of the the mutual exclusion on boxes is that we must avoid cyclic calls if we are to avoid deadlock.

1.1.6 Concurrent execution in Seuss

In Seuss, a program consists of a set of boxes. Suppose we implement a Seuss program so that each box is allocated a separate processor, and procedure calls are implemented using a communication network. We identify a box and its processor, using the name "box" for either. A scheduler program decides when threads should be started, and sends a message to the appropriate box to start the thread.

Initially, every box in the system is idle. Box D starts executing a thread for action $D.a$ when it receives a message to do so from the scheduler. If there is a call to method $E.m$ during execution of the code of $D.a$, then D sends a message to E and suspends execution until it receives a message back from E, indicating that the call is complete. When D reaches the end of $D.a$'s code, D sends a message to the scheduler that the thread has ended, and becomes idle again.

Each box in this model is largely independent. There is only synchronization between boxes for procedure call and return. The scheduler decides which threads to run, and when to start them, but has no further control over the execution of a thread. The scheduler can be defined to execute a single thread at a time, as in an action system. We call this a *sequential* execution. In a sequential execution, at most one box is executing at any time. The rest are either idle, or suspended, waiting for another box to complete a method call. Note that replacing each complete thread in a sequential execution by an atomic step gives an atomic execution.

Consider boxes *Prod* and *Cons*. The code in *Prod.make* that produces an item does not involve calls to boxes *Buff* or *Cons*. Similarly, the code in *Cons.use*

that consumes the item does not involve call to boxes *Buff* or *Prod*. These sections of code execute on disjoint boxes.

Box *Prod* is idle during the execution of a thread for *Cons.use*. Consider starting a thread for *Prod.make* while a thread for *Cons.use* is executing. The thread executes initially at *Prod*, which is otherwise idle. The call to *Buff.put* at the end of the code obeys the mutual exclusion at *Buff* with the call to *Buff.get* from the thread for *Cons.use*.

Suppose there is an item in the buffer at the start. The thread for *Cons.use* accepts, retrieving the first item from the buffer and consuming it, and the thread for *Prod.make* produces an item and puts it at the back of the buffer. The item retrieved is in the buffer at the start, so the order of the method calls to *Buff* from the two threads does not affect the item retrieved, or the final state of the buffer. The final state is the same as is reached by executing the threads sequentially, in either order.

Suppose, on the other hand, that the buffer is empty at the start. In this case the thread for *Cons.use* rejects if it calls *Buff.get* before the thread for *Prod.make* calls *Buff.put*, and it accepts if it calls after. In the first case, an item is produced, but none is consumed, and there is a single item in the buffer at the end. This is what happens if we execute the thread for *Cons.use* followed by the thread for *Prod.make*. In the second case, an item is produced, added to the buffer, removed from the buffer, and consumed, leaving the buffer empty. This is what happens if we execute the threads in the opposite order.

Thus, the final state after the concurrent execution of the two threads is a state reachable by executing the threads sequentially. We can extend this by considering executions containing n threads for *Prod.make* and *Cons.use*, where threads are executed concurrently, obeying the mutual exclusion on boxes. We claim that, for any execution in this set, there is a sequential execution of the same

set of threads with the same start and final states.

In general, if we take two sequential programs with shared variables and run them concurrently, we can expect to reach an final state that is not reachable by running the programs one after the other. This interference between separate threads of control is what makes concurrent programming difficult. But in this example, it seems that a concurrent execution of these procedures has the same behaviour as a sequential execution.

Since the concurrent execution allows execution at more than one box, it allows a more efficient use of the computing resources. In the producer-consumer example, we might expect that producing and consuming the items in parallel takes about half the time of an equivalent sequential execution.

For the example, it seems we have the following happy circumstances. The program can be implemented concurrently, for reasons of efficiency, but the concurrent executions introduce no behaviours that are not observable in sequential executions. If we show that all sequential executions have a certain behaviour, then we can conclude that all concurrent executions have that behaviour. To show properties of the sequential executions, we use the separation noted above, and give sequential specifications for the actions, and use these to show properties for atomic executions. We have a dual view of execution for the program: concurrent execution for implementation, and atomic execution for proving properties of the program.

The remainder of this work is an investigation of the circumstances under which this dual view of a program's execution is possible. We show sufficient conditions on programs, and restrictions on executions, such that every concurrent execution is equivalent to an atomic execution.

1.2 Reduction

A concurrent execution of a program in the Seuss model consists of a sequence of steps, each involving one or two boxes. Each step advances one of the threads active at that point in the execution. We represent the concurrent execution of different threads by interleaving the steps of the threads.

In an atomic execution, the system state is represented by the values of the program variables. For a concurrent execution, we need to record more information about the system state between two steps. In addition to the values of the box variables, we record, for each box, which procedure, if any, is currently being executed, and where the box has reached in the execution of the procedure's code. We call the sum of the relevant information about the system state in a concurrent execution the *configuration* of the system. None of the extra information is necessary for an atomic execution, since every box is idle after every step.

An atomic execution has simpler states, and larger steps than a concurrent execution. We can reason about an atomic execution at a higher level of abstraction, where many of the details recorded in a concurrent execution have been hidden.

To show that there is an atomic execution corresponding to every concurrent execution, we use *reduction*. This is a technique for transforming an execution into an equivalent execution by applying local transformations to the sequence of steps in a concurrent execution. We prove rules that allow particular pairs of adjacent steps to be swapped, without affecting the remainder of the execution, and we apply these rules repeatedly to reach an execution where the steps of a given thread appear in an unbroken sequence, with no interleaved steps from other threads. If we can do this with every thread, the resulting execution is sequential, which, as we have seen, corresponds to an atomic execution.

Lipton [24] introduced the term "reduction" for this process of rearranging the steps in a concurrent execution to bring together all the steps for a single thread.

His theory applied to a restricted class of programs using semaphores, and to properties of the form “the program never becomes deadlocked”. We apply reduction to a wider class of programs, but the essential ideas from Lipton's theory guide our work.

1.2.1 The two-phase locking protocol

For an example of reduction, consider *two-phase locking* protocol from database theory [12]. The execution model is one in which data items (records in a database, for example) have *resources* associated with them. All transactions acquire the associated resources before accessing a data item. The transaction releases the resources after it has accessed the data item. An action that accesses multiple items at the same time first acquires the associated resources for each. Resources that allow read access to an item are *shared*, meaning that multiple transactions can read the item concurrently, whereas resources that allow write access to an item are *exclusive*, meaning that only one transaction can write to the item at a time. For now, we consider just exclusive resources.

An action is called *two-phase* if in any execution, it acquires no new resources after it has released a resource. This means that every execution of a two-phase transaction can be divided into two parts. In the first part, resources are acquired, and in the second part they are released. The two-phase locking theorem says that for any concurrent execution of a set of two-phase transactions there is a sequential execution of the same set of transactions where each data item has exactly the same operations applied to it, in the same order, as in the concurrent execution.

The following example shows the ideas behind the two-phase locking theorem. The execution below shows a thread for a two-phase action α, executing concurrently with threads for other actions. The steps a_i, b, and c_i are the steps of the thread

for α, with ellipses representing the steps from other threads.

$$a_0 \,;\, \cdots \,;\, a_1 \,;\, \cdots \,;\, a_2 \,;\, \cdots \,;\, b \,;\, \cdots \,;\, c_2 \,;\, \cdots \,;\, c_1 \,;\, \cdots \,;\, c_0$$

Resource acquisition steps can fail, because the resource is unavailable. Only successful acquisition steps are shown in the execution. Note that a step that accesses an item is from a thread that holds the necessary resources.

The thread for α acquires three resources, R_0, R_1, and R_2, with steps a_0, a_1, and a_2. These allow it to access some items with step b. It then releases the resources with steps c_2, c_1, and c_0.

Consider step a_2. Before this step, no thread holds resource R_2, and after it step, the thread for α holds it. The resources available after a_2 are a subset of those available before. If s is the step after a_2 in the execution, and s is from a different thread, s does not acquire R_2. Since it succeeds after a_2 it will succeed before, since any resources it acquires are available. Resource R_2 is available after s if it is available before, so both steps succeed if we reverse their order. We swap the steps, and repeat the argument with every step between a_2 and b, and then repeat the whole process with steps a_1 and a_0. This gives us the following execution.

$$\cdots \,;\, a_0 \,;\, a_1 \,;\, a_2 \,;\, b \,;\, \cdots \,;\, c_2 \,;\, \cdots \,;\, c_1 \,;\, \cdots \,;\, c_0$$

Here there are no steps from other threads between the first four steps of the thread.

Now consider step c_2. This releases resource R_2, so the resources available after it executes are a superset of those before. This step can be moved left over all the intervening steps until it is next to b. Likewise, we can move c_1 and c_0. We get the following execution.

$$\cdots \,;\, a_0 \,;\, a_1 \,;\, a_2 \,;\, b \,;\, c_2 \,;\, c_1 \,;\, c_0 \,;\, \cdots$$

Here all steps from the thread are contiguous. Any step from another thread that was before step b in the original execution is before it in this reduced exection, and similarly with the steps after b. Thus the relative order in which the threads access the items is the same as in the original execution. If all the actions in the program are two-phase, we can apply the above reduction to any concurrent execution by applying it to each action in turn, and this gives a sequential execution in which actions access items in the same order as in the original execution.

Lipton used the names *right-movers* for steps such as a_i in the above example, and *left-movers* for steps such as c_i. He identified right-movers as steps that acquire resources, and left-movers as those that release them.

1.2.2 Reduction for Seuss

In Seuss, the items accessed by thread are the box variables. The resources that allow access to these items are the boxes themselves. A thread holds a box if it is executing a procedure call at that box. Because of the mutual exclusion on boxes, a box is an exclusive resource.

From this, we can see that a thread is two-phase, in the above sense, only if it calls at most one method during its execution, and the method called itself only call one method, and so on. The class of two-phase Seuss programs with this restriction is limited. It is difficult to imagine that we can find a two-phase program "equivalent" to any Seuss program. Thus, for a useful reduction theory for Seuss, the two-phase condition is too strict.

The following example demonstrates the issues involved in reducing Seuss programs, and provides some motivation for the approach we take. Consider the program in Figure 1.5. The program contains a box X, which has a local variable x, and two methods, *add*, and *mult*. Both take a single input parameter. The first adds the parameter value to x, while the second multiplies x by the parameter value.

```
box X
   var x : integer
   method add(in a : integer)
                  ::  x := x + a
   method mult(in a : integer)
                  ::  x := x * a
end

box D
   action aa  ::  X.add(3) ; X.add(4)
end

box E
   action a   ::  X.add(2)
   action b   ::  X.mult(2)
end
```

Figure 1.5: Program with commuting and noncommuting methods

Boxes D and E have no variables. Action $D.aa$ makes two calls to method $X.add$, and action $E.a$ makes one call. Action $E.b$ makes one call to method $X.mult$.

Consider a concurrent execution of actions $D.aa$ and $E.a$ with the following outline.

$$init_D \; ; \; init_E \; ; \; X.add(3)_D \; ; \; X.add(2)_E \; ; \; X.add(4)_D \; ; \; term_D \; ; \; term_E$$

Here $init_D$ represents the steps initializing the thread at D, $term_D$ represents the steps to complete the thread at D after the method calls, and $X.add(3)_D$ represents all the steps of a successful call to $X.add$. The other steps have a similar interpretation, with the subscript E meaning that a step is taken on behalf of box E. In this execution, the call to $X.add$ from $E.a$ occurs between the calls from $D.aa$.

Clearly, $D.aa$ is not two-phase, since it releases box X and then reacquires it. However, we note that any two calls to $X.add$ *commute*, meaning that the final

effect on $X.x$ is the same regardless of the order. Thus, we argue that we can replace

$$X.add(2)_E; X.add(4)_D$$

with

$$X.add(4)_D; X.add(2)_E$$

since the overall effect in either case is to add 6 to $X.x$.

Using this, we reduce the above execution to the following, equivalent, execution.

$$init_D \; ; \; init_E \; ; \; X.add(3)_D \; ; \; X.add(4)_D \; ; \; X.add(2)_E \; ; \; term_D \; ; \; term_E$$

Now we argue that, since $init_E$ affects only box E, and $X.add(3)_D$ affects only boxes D and X, that these steps can be exchanged. We apply a similar argument for $init_E$ and $X.add(4)_D$, so we end up with the following execution.

$$init_D \; ; \; X.add(3)_D \; ; \; X.add(4)_D \; ; \; init_E \; ; \; X.add(2)_E \; ; \; term_D \; ; \; term_E$$

Finally, we argue as above that $term_D$ can be exchanged with $X.add(2)_E$ and $init_E$, since again the steps affect disjoint parts of the program. The end result is a sequential execution, as follows.

$$init_D \; ; \; X.add(3)_D \; ; \; X.add(4)_D \; ; \; term_D \; ; \; init_E \; ; \; X.add(2)_E \; ; \; term_E$$

Here we were able to reduce the original execution to a sequential execution by using two facts:

- Any two calls to $X.add$ commute.

- Any two steps that affect disjoint parts of the state space commute.

Note that the effect of the final sequential execution is to add 7 to $X.x$, and the original concurrent execution has exactly the same effect on $X.x$. So this reduction seems reasonable.

Consider now a similar execution to the above, but with actions $D.aa$ and $E.b$.

$$init_D \; ; \; init_E \; ; \; X.add(3)_D \; ; \; X.mult(2)_E \; ; \; X.add(4)_D \; ; \; term_D \; ; \; term_E$$

Now if we try to apply the above argument, we get stuck at the first step, because $X.mult(2)_E; X.add(4)_D$ is not equivalent to $X.add(4)_D; X.mult(2)_E$, because the effect of doubling number and then adding 4 is not the same as that of adding 4 and then doubling.

If the above execution of $D.aa$ and $E.b$ is started from a configuration where $X.x = 3$, then the execution ends in a configuration where $X.x = 16$. The sequential execution $D.aa; E.b$ starting from the same configuration, ends with $X.x = 20$, and the execution $D.aa; E.b$ ends with $X.x = 13$. Thus there is no way to represent the above concurrent execution as a sequential execution of $D.aa$ and $E.b$.

This suggests that we distinguish pairs of actions, such as $D.aa$ and $E.a$, which are "well-behaved" with regards to reduction, from pairs such as $D.aa$ and $E.b$ which are not. A pair of actions is well-behaved if a procedure called during a thread for one commutes with every procedure called during a thread for the other. If we execute the program so that only well-behaved pairs of actions run concurrently, then method calls can be reordered, as we did above to show the reduction for $D.aa$ and $E.a$.

This is our approach to reduction. We define *control relations* as a general mechanism for controlling concurrency. A control relation contains pairs of actions that may have concurrent threads. An execution *respects* a control relation if, at

all times, any pair of concurrently executing threads are for actions in the control relation. We define control relations that allow as much concurrency as possible, while allowing the reduction of every execution that respects the relation to an atomic execution. In particular, we run the program above under a control relation that contains $(D.aa, E.a)$ but not $(D.aa, E.b)$.

1.3 Summary of the material

The main results we prove are the following.

Complete execution theorem A *complete* execution is one where all threads terminate without errors. The theorem gives some conditions on executions that ensure that every thread terminates. We define conditions on a program, and on an implementation of the program, such that every execution is complete.

First reduction theorem We define a control relation in terms of commutativity conditions on procedure calls. The theorem shows that every complete execution respecting this control relation can be reduced to an atomic execution.

Second reduction theorem We consider the fairness conditions satisfied by a concurrent execution and its atomic reduction. The theorem shows a control relation for any subset of the actions, such that a complete execution respecting this control relation can be reduced to an atomic execution, where the original and reduced executions satisfy the same fairness conditions for actions in the subset.

The results are stated and proven in terms of a specific language, *TCB*, that implements the Seuss model. We define an operational semantics for this language, and we prove the above results for executions under this semantics. However, the results are applicable to *any* language implementing the Seuss model. The basic

mechanisms for starting and ending threads, implementing method call and return, and ensuring mutual exclusion on the boxes are common to all these languages, and these are the features of the semantics that are important in proving the theorems.

1.3.1 Chapter overview

Chapter 2 defines a syntax and an operational semantics for *TCB*. We define a *program configuration* for representing the system state during a concurrent execution, and we give the semantics as a relation on program configurations, defined by a set of inference rules.

Chapter 3 proves some basic results about the program configurations and inference rules in the semantics for *TCB*. We define the *label* for a step, and use this to define a program execution.

Chapter 4 investigates the causes of nonterminating threads. We prove the *complete execution theorem*, which defines exactly the conditions necessary for termination of all threads in an execution. We define *control relations*, and show how they can be used to avoid nonterminating threads due to deadlock or infinite execution.

Chapter 5 contains the *first reduction theorem*, and its proof. We define a control relation, called *weak compatibility*, using a commutativity condition on procedure calls. We show that any execution that respects this control relation can be reduced to an atomic execution. We show a correspondence between the original concurrent execution and the reduced atomic execution, which can be used to infer properties of the concurrent execution from properties of the atomic execution.

Chapter 6 discusses the impact of fairness conditions on the progress properties that can be proven of an atomic execution. We define two types of fairness, *weak fairness* and *minimal fairness*, and show that executions respecting the first have more progress properties than executions respecting the second. We show that there

are weakly fair executions respecting weak compatibility where the reduction guaranteed by the first reduction theorem gives an atomic execution that is not weakly fair. We show a stronger control relation, and the *second reduction theorem*, which shows that weakly fair executions satisfying this control relation can be reduced to weakly fair atomic executions.

Chapter 7 contains some concluding remarks and observations. Appendix A contains the semantic rules for *TCB*. Appendix B contains some of the proofs omitted from earlier chapters.

1.4 Related work

1.4.1 Seuss

The Seuss model was first described by Jayadev Misra in [25]. Early drafts of that work stated, but did not completely prove, a reduction theorem similar to the first reduction theorem in Chapter 5.

We implement all features of Seuss as outlined by Misra, with the exception of *negative alternatives*. In the full Seuss model, the alternatives in a partial procedure are of two types: *positive* and *negative*. The alternatives shown in Section 1.1.3 are positive alternatives. The execution of a negative alternative is the same as for a positive alternative, except that a negative alternative always rejects, regardless of the evaluation of the guard and the body. A procedure with negative alternatives can change the value of box variables on a rejecting call.

Misra shows that negative alternatives add to the expressive power of the language. A *strong semaphore*, one that guarantees that every action attempting to acquire the semaphore eventually succeeds, cannot be defined with positive alternatives alone. With negative alternatives, the P operation on the semaphore is written so that the identity of the caller is recorded when the semphore is not avail-

able, but the call rejects, indicating to the calling procedure that the semaphore is not available.

Rajeev Joshi and Misra show in [20] that the correct functioning of a strong semaphore coded with negative alternatives requires that all actions that attempt to acquire the semaphore be *persistent*. An action α is persistent in a given execution if there are an infinite number of threads for α in the execution, or the final thread for α is accepting. Equivalently, α is persistent if every rejecting thread for α is followed by another thread for α.

Negative alternatives thus violate the rule that a rejecting thread does not change the state, and they introduce additional complications, so, in the interests of simplicity, we choose not to implement them in our work.

1.4.2 Reduction

Richard Lipton's paper [24], introduced the term *reduction* for the transformation of an execution to an equivalent one in which all the steps of a given thread appear contiguously, and thus may be regarded as atomic. The motivation for his work, as with the current work, is that the reduced execution has a coarser grain of interleaving than the original, so there is less interaction, and thus less possibility for interference, between concurrently executing threads.

Lipton introduced the terms *right-mover*, for a step that can be moved right (that is, delayed), and *left-mover*, for a step that can be moved left (that is, advanced). The result he gives applies only to two-phase programs, where every execution consists of a sequence of right-movers followed by a sequence of left-movers. The reduction theorem guarantees that the reduced execution is deadlock-free if and only if the original execution is.

Several researchers have extended Lipton's ideas. In [10], Doeppner defines *expansions* that allow a single large action in a program to be replaced with a series

of smaller steps. He defines various notions of *consistency* between programs and their expansions, and shows that an execution that can be reduced by Lipton's reduction theorem is a consistent expansion of its reduced execution.

In [22], Leslie Lamport gives a reduction theorem for two-phase threads, consisting of a sequence of right-movers, followed by a single *central step*, and a sequence of left-movers. In [23], Lamport and Fred Schneider give a theorem for a reduction of two-phase threads that preserves all safety properties.

Ralph-Johan Back, in [2] and, with Joakim von Wright, in [3], presents a technique for refining action systems by repeatedly replacing a single action with an equivalent action system. This is called *atomicity refinement.* If action system A is refined to action system A' by this technique, then executions of A' have a finer grain of interleaving than executions of A. The technique is aimed at terminating programs, and the refinement guarantees that A and A' have the same total correctness properties.

In [6], Ernie Cohen proves the reduction theorems of Lipton, Doeppner, and Lamport and Schneider, and the partial correctness part of Back's theorem, in terms of Kleene algebra (an algebra of regular expressions). Presenting the results in a uniform framework allows the relationships between the theorems to be explored.

In [7], Cohen and Lamport show a reduction theorem that preserves not only safety (or partial correctness) properties, but also weak and strong fairness properties. As with Lipton's theorem, and Lamport's earlier reduction results, the theorem applies only to two-phase threads.

The above reduction theorems apply to *any* execution of a set of two-phase threads. Threads in Seuss that call total methods are not two-phase. We cannot show a reduction theorem that applies to any execution of an arbitrary Seuss program. Instead, we define a restriction on concurrency, and we show a reduction for all executions obeying this restriction.

In [28], Susan Owicki and David Gries give sufficient conditions between sequential programs that they do not interfere with each other's execution if run concurrently. Their method involves checking every statement in one program against every statement in the other. The restriction on concurrency that we define to ensure reduction is defined by a similar condition between actions, except that we exploit the mutual exclusion on procedure execution at a box. In the condition we define, each procedure called during execution of a thread for one action is checked against every procedure called by a thread for the other.

1.4.3 Transaction processing

The two-phase locking theorem says that any set of two-phase transactions can be run concurrently and the resulting execution is serializable. However, the two-phase format is a severe restriction on the allowed form of transactions. Transactions are often written so that a lock is acquired long before, or released long after, the associated data is accessed. This decreases the opportunity for concurrent execution.

There has been some research into non-two-phase locking protocols, with the aim of increasing the concurrent execution of transactions. In [26], C Mohan, Don Fussell and Avi Silberschatz describe a locking scheme using *invisible* locks, in addition to standard shared and exclusive locks. The invisible locks allow neither reading nor writing of the associated data, but a transaction must hold some invisible locks before it can acquire an exclusive lock.

The locking protocol with invisible locks assures serializability of accesses to the data items, even for non-two-phase transactions. The protocol does not allow a transaction to reacquire a lock it has released, but it may acquire other locks. One of the interesting results presented in this work is that every deadlock involves a transaction holding only invisible locks, and this transaction has not changed to value of any data. Thus to break the deadlock, the transaction with only invisible

locks can be aborted. There is no need for a mechanism to roll back the state of the system on such an abort, since the aborted transaction has made no state changes.

Eliot Moss's work on *nested transactions* [27], extends the single-level transaction model to one in which a transaction may call several subtransactions during its execution. Each of these subtransactions is itself a transaction, and each may commit or abort independently. A transaction is not obliged to abort if a subtransaction aborts. The intention is to provide a robust mechanism for implementing programs on unreliable hardware. The implementation of the model ensures that each part of a computation can be regarded as an uninterrupted atomic action.

The nested transaction model allows for changes to the underlying state prior to a decision to abort a transaction. If a transaction is aborted, all changes made to the state by the local code of the transaction, and by any committed subtransactions, is rolled back. Much of Moss's research concerns the managing of dependencies between transactions. This is required so that during rollback for an abort of a transaction, any other transaction that has read values written by the aborted transaction is itself aborted.

Seuss procedures can be regarded as transactions, and method calls are thus calls to nested transactions. However, Seuss requires no mechanisms for rollback on rejection, since we ensure that the state is never changed unless a thread has committed.

1.4.4 Concurrent object-based languages

There are a number of languages that offer both concurrent execution and object-based encapsulation of data and code. Many of these use an existing language as a basis. For example, COOL [4] is one of a number of proposals for extending C++ with concurrent execution, and ABCL [33], and its various descendents, are based on Lisp. We are not aware of work applying reduction to any of these languages.

In [19], Steve Hodges and Cliff Jones present an operational semantics for the language $\pi o\beta\lambda$, a concurrent object-based modeling language. They use this semantics to prove the validity of optimizations allowing increased concurrency. The main optimization is *early return*. In this a call to procedure π' from procedure π returns as soon as the values of the return parameters are available. The remainder of the executions of the calls to π' and π are executed concurrently.

The semantics for $\pi o\beta\lambda$ given by Hodges and Jones is based on the Structured Operational Semantics outlined by Gordon Plotkin in [29]. The semantics we give for Seuss is strongly influenced by that given in [19].

1.5 Notation

We use the operator $\triangleq$ to mean "is equal by definition". We use the following general format for quantification.

$$\langle \circ\, x \;:\; R.x \;:\; T.x \;\rangle$$

Here $\circ$ is a commutative associative binary operator. We call x the *dummy*, $R.x$ the *range*, and $T.x$ the *term*. The range is a boolean expression, and the term is an expression of the correct type to be a operand of $\circ$. The meaning of the quantification is the result of applying operator $\circ$ to the set of values $T.x$ for all x satisfying $R.x$.

We use the following instances of this notation.

$$\begin{aligned}
\langle \forall\, x \;:\; R.x \;:\; T.x \,\rangle &\triangleq \text{for all } x\text{, if } R.x \text{ then } T.x \\
\langle \exists\, x \;:\; R.x \;:\; T.x \,\rangle &\triangleq \text{there exists an } x\text{, such that } R.x \text{ and } T.x \\
\langle \exists !\, x \;:\; R.x \;:\; T.x \,\rangle &\triangleq \text{there exists a unique } x\text{, such that } R.x \text{ and } T.x \\
\langle \Sigma\, x \;:\; R.x \;:\; T.x \,\rangle &\triangleq \text{the sum of } T.x\text{, for } x \text{ satisfying } R.x, \\
\langle \#\, x \;:\; R.x \;:\; T.x \,\rangle &\triangleq \text{the number of } x \text{ satisfying } R.x \text{ and } T.x \\
\langle \min x \;:\; R.x \;:\; T.x \,\rangle &\triangleq \text{the minimum of } T.x\text{, for } x \text{ satisfying } R.x, \\
\langle \cup\, x \;:\; R.x \;:\; T.x \,\rangle &\triangleq \text{the union of } T.x\text{, for } x \text{ satisfying } R.x,
\end{aligned}$$

When the range is *true* or understood from the context, we omit it, and write $\langle \forall\, x \;::\; T.x \,\rangle$. For long formulae, we write the quantification over several lines as follows.

$$\begin{array}{l}
\langle \forall \;\; x \\
\;\; : R_0.x \;\wedge\; R_1.x \;\wedge\; R_2.x \;\wedge\; R_3.x \;\wedge\; R_4.x \;\wedge\; R_5.x \;\wedge\; R_6.x \\
\;\; : T_0.x \;\wedge\; T_1.x \;\wedge\; T_2.x \;\wedge\; T_3.x \;\wedge\; T_4.x \;\wedge\; T_5.x \;\wedge\; T_6.x \\
\;\; \rangle
\end{array}$$

For sets, we use the following form of quantification

$$\{\, x \;:\; R.x \;:\; T.x \,\} \;\triangleq\; \text{the set of all } T.x\text{, for } x \text{ satisfing } R.x$$

For set quantification where the term is x, we use the comprehension notation.

$$\{\, x \mid R.x \,\} \;\triangleq\; \{\, x \;:\; R.x \;:\; x \,\}$$

We use the following operator to express the fact that two sets are disjoint. For sets

A and B,

$$A \operatorname{disj} B \triangleq A \cap B = \emptyset$$

For a set A,

$$\#A \triangleq \text{the number of elements in } A$$

$$A_\perp \triangleq A \cup \{\perp\}$$

Here, $\perp$ is an element not in set A. A common use of a set $A_\perp$ is to define a function that returns either an element of A, or $\perp$ indicating "no such item".

For sets A and B, $A \to B$ is the set of total functions from A to B, and $A \hookrightarrow B$ is the set of partial functions from A to B. For a function f, we write $\operatorname{dom}(f)$ and $\operatorname{rng}(f)$ for the domain and range of f.

For a set A, A^* is the set of finite sequences over A, and A^∞ is the set of finite and infinite sequences over A. For $s \in A^\infty$, we write $|s|$ for the length of s. If $s \notin A^*$, then $|s| = \infty$. We use the indexing operator $s[i]$ for the ith element of s, for $0 \leq i < |s|$. Element indices start at 0. We use the segment operator $s[i \ldots j]$ for the segment of s containing elements $s[i]$ through $s[j]$ inclusive, for $0 \leq i \leq j < |s|$. For an infinite sequence, we allow segments $s[i \ldots \infty]$, meaning the segment from element $s[i]$ on. With these, we define

$$\operatorname{first}(s) \triangleq s[0]$$

$$\operatorname{rest}(s) \triangleq s[1 \ldots (|s| - 1)]$$

$$\operatorname{last}(s) \triangleq s[|s| - 1] \quad \text{if } |s| < \infty$$

For $s, t \in A^\infty$, the concatenation of s and t is written $s \circ t$. If $|s| = \infty$, then $s \circ t = s$.

Sequences are ordered by the prefix ordering.

$$s \sqsubseteq t \;\triangleq\; \langle \exists\, s' \;:\; s' \in A^\infty \;:\; s \circ s' = t \;\rangle$$

Under this order, the set A^∞ is a *complete partial order* (CPO) (see [8]). By definition, the least upper bound of any chain is an element of A^∞. That is, if $s_0, s_1, s_2, \ldots$ are such that for all i, $s_i \in A^\infty$, and $s_i \sqsubseteq s_j$ for all i, j, such that $i < j$, then the least upper bound of the s_i, written $\langle \sqcup\, i \;:\; 0 \leq i \;:\; s_i \;\rangle$, is an element of A^∞. This means we can define elements in A^∞ as the limit of a chain of finite sequences. We use $\perp$ for the empty list. This is the bottom element in the prefix order.

We write proofs in the style of [9]. The following proof fragment shows the essential features of this format.

$$\begin{array}{ll}
& \neg(a = b \wedge b = c) \;\wedge\; (b = c \vee c = d) \\
\equiv & \quad \{ \text{ predicate calculus } \} \\
& (a = b \Rightarrow b \neq c) \;\wedge\; (b \neq c \Rightarrow c = d) \\
\Rightarrow & \quad \{ \text{ transitivity of } \Rightarrow \} \\
& a = b \Rightarrow c = d \\
\Rightarrow & \quad \{ \text{ assumption } a = b \text{ } \} \\
& c = d
\end{array}$$

The proof shows derivation of $c = d$, from the formula in the first line, and the assumption $a = b$. The proof consists of alternating lines of terms and hints. A hint line starts with a connective. The second line of the proof (the first hint) says that the term in the first line is equivalent to the term in the third line. The hint justifies this claim with a reference to the appropriate mathematical law, theorem, definition, or assumption. The proof as a whole says that the first term implies the last term.

We define tuples with typed components as in the following example.

$$T \triangleq \textbf{record} \quad r : integer \quad b : boolean \quad \textbf{end}$$

This defines the type T as a tuple consisting of an integer r and a boolean b. If $t \in T$, we write $t.r$ for the first component of t, and $t.b$ for the second component. We write an arbitrary element of T as (r, b).

Chapter 2

The definition of *TCB*

2.1 Introduction

All Seuss languages have the same mechanisms for communication between boxes, and for evaluation of guards, so the major difference between them is the sequential language used to write procedure bodies. *TCB* is a Seuss language that uses a simple sequential language, called *TCBloc*, for procedure bodies. We define a syntax and operational semantics for both languages.

TCBloc is a simple language, and it has a simple semantics with a single thread of control. In the semantics of *TCB*, we must represent threads executing concurrently at different boxes.

We define the semantics for *TCBtot*, a restricted form of *TCB* in which all procedures are total. In this semantics, we represent the execution of each box in the program separately, with communication between boxes for method call and return. The semantics implements procedure call and return, but does not implement guard evaluation or rejecting procedure calls, since there are no partial procedures.

We extend the semantics for *TCBtot* to a semantics for the full *TCB* language. The extension implements guard evaluation and rejecting procedure calls.

We give two semantic definitions for *TCB*, using different models for procedure call. The differences are discussed below.

2.2 Syntax

The syntax of *TCB* is essentially the same as that used for the example programs in Chapter 1. We define a simple sequential language, called *TCBloc*. This has some of the standard features of an imperative language: assignment, alternation, and iteration. We define *TCB* as a Seuss language that uses *TCBloc*, augmented with a procedure call statement, as the language for the bodies of procedures.

2.2.1 Variables, states, values and expressions

The set *Id* contains the legal identifiers. We use x, y and z as typical variables, D and E as typical boxes, a as a typical action, m and n as typical methods, and p as a typical procedure.

We are deliberately vague about the language used for expressions in this language. We assume that we can represent values of the following types.

$$\begin{aligned} \mathit{boolean} &= \{\mathit{true}, \mathit{false}\} \\ \mathit{integer} &= \{\ldots, -2, -1, 0, 1, 2, \ldots\} \end{aligned}$$

We define

$$\begin{aligned} \mathit{Val} &\triangleq \mathit{boolean} \oplus \mathit{integer} \\ \mathit{Type} &\triangleq \{\mathit{boolean}, \mathit{integer}\} \end{aligned}$$

Here $\oplus$ is the disjoint union operator. The set *Val* contains all values that can be assigned to a variable. We use v for a typical member of *Val*. The set *Type* contains the all the types in *TCB*.

We use standard operators for the boolean and integer types, including equality, logical operators such $\wedge$ and $\neg$, and integer operators such as $+$ and $*$. Set *Exp* contains all legal expressions. We use e for a typical expression, and b for a typical boolean expression.

We use a *state* to give the values of the variables in a program. A state is a map from variable identifiers to values.

Definition 2.1

$$State \;\triangleq\; Id \hookrightarrow Val_{\perp}$$

A *State* is a partial function, since only a subset of *Id* are defined as variables in any given program. We use σ for a typical member of *State*. For any $x \in VarId$, if $x \in \mathrm{dom}(\sigma)$, $\sigma.x$ is the value of x in σ. If $\sigma.x = \perp$, then x is defined in σ, but uninitialized.

The evaluation operator $[\![\cdot]\!]$ is defined so that for any expression e, and any state σ such that every variable in e is defined in σ, with a value of the appropriate type, the value of e in σ is $[\![e]\!]\sigma$.

The type *VarTypeList* is used to represent a list of variable ids and their associated types (for example, for a parameter list). The elements of *VarTypeList* are sequences over $Id \times Type$, such that all the *Id* entries in the sequence are distinct. A typical element of *VarTypeList* is V. For $L \in VarTypeList$, $VarList(L)$ is the sequence of ids in L.

2.2.2 Syntax for *TCBloc*

A *TCBloc* program is shown in Figure 2.1. This example shows all the constructs of *TCBloc*, a minimal language of the Algol-Pascal family. A program consists of a variable declaration block, started by **var**, followed by the code of the program

$$
\begin{array}{l}
\textbf{var}\ x : integer \\
\quad d : integer\ \textbf{init}\ 4 \\
\quad q : integer \\
\quad r : integer \\
\quad b : boolean\ \textbf{init}\ false \\
\textbf{begin} \\
\quad q := -23 \\
\quad \textbf{if}\ x < 0\ \textbf{then}\ x := -x\ \textbf{else}\ b := true \\
\quad r := x\ ;\ q := 0 \\
\quad \textbf{while}\ r \geq d\ \textbf{do}\ |[\ r := r - d\ ;\ q := q + 1\ ]| \\
\textbf{end}
\end{array}
$$

Figure 2.1: *TCBloc* code to divide x by d

between **begin** and **end**. The program shown implements integer division using repeated subtraction. Four *integer* variables and one *boolean* variable are declared. Variable d is initialized to 4, b to *false*, and x, q, and r are uninitialized. The first statement of the code assigns a value to x. The second statement checks if x is negative, and if it is, negates it. The next line contains two assignment statements in sequence. The final line is a loop statement that checks if r is at least d, and if it is, updates r and q and repeats. The two assignments in the loop are enclosed in brackets indicating that they form a compound statement, which is treated as a single statement.

The syntax for *TCBloc* is shown in Figure 2.2. It is given as a set of Backus Normal Form (BNF) production rules. Terminals and nonterminals of the grammar are enclosed in angle brackets. On the right-hand side of the production rules, we use quotation marks around elements of the concrete syntax, and we use

$\langle locprog \rangle$	::=	"**var**" $\langle decl \rangle^*$ "**begin**" $\langle stmt \rangle^*$ "**end**"
$\langle decl \rangle$	::=	$\langle id \rangle$ ":" $\langle type \rangle$ ["**init**" $\langle value \rangle$]
$\langle stmt \rangle$	::=	$\langle assign \rangle$ | $\langle ifelse \rangle$ | $\langle whiledo \rangle$ | $\langle compound \rangle$
$\langle assign \rangle$	::=	$\langle id \rangle$ ":=" $\langle exp \rangle$
$\langle ifelse \rangle$	::=	"**if**" $\langle exp \rangle$ "**then**" $\langle stmt \rangle$ "**else**" $\langle stmt \rangle$
$\langle whiledo \rangle$	::=	"**while**" $\langle exp \rangle$ "**do**" $\langle stmt \rangle$
$\langle compound \rangle$	::=	"|[" $\langle stmt \rangle^*$ "]|"

Figure 2.2: Syntax for *TCBloc*

the following operators.

$$
\begin{aligned}
S^* &\triangleq S \text{ repeated 0 or more times} \\
S^+ &\triangleq S \text{ repeated 1 or more times} \\
S\, S' &\triangleq S \text{ followed by } S' \\
S \mid S' &\triangleq S \text{ or } S' \\
[\ S\ ] &\triangleq S \text{ is optional}
\end{aligned}
$$

The operators are given in order of precedence, from highest to lowest. We use parentheses for grouping when necessary.

Thus, the first production rule in Figure 2.2 says that a program $\langle locprog \rangle$ is the keyword **var**, followed by zero or more $\langle decl \rangle$s, followed by the keyword **begin**, followed by zero or more $\langle stmt \rangle$s, followed by the keyword **end**. The second rule says that a declaration $\langle decl \rangle$ is an $\langle id \rangle$, followed by the separator :, followed by a $\langle type \rangle$, optionally followed by the keyword **init** and a $\langle value \rangle$. The third rule says that a statement $\langle stmt \rangle$ is one of an $\langle assign \rangle$, an $\langle ifelse \rangle$, a $\langle whiledo \rangle$, or a $\langle compound \rangle$.

$$
\begin{array}{l}
\textbf{box } \mathit{Sem} \\
\quad \textbf{var } n : \mathit{integer} \textbf{ init } 1 \\
\quad \textbf{method } P \quad :: \quad n > 0 \longrightarrow n := n - 1 \\
\quad \textbf{method } V \quad :: \quad n := n + 1 \\
\textbf{end} \\
\\
\textbf{box } D \\
\quad \textbf{var } x : \mathit{integer} \textbf{ init } 0 \\
\quad \textbf{action } \mathit{act} \quad :: \quad \mathit{true} \ \& \ \mathit{Sem.P} \longrightarrow x := x + 1 \ ; \ \mathit{Sem.V} \\
\textbf{end} \\
\\
\textbf{box } E \\
\quad \textbf{var } y : \mathit{integer} \textbf{ init } 0 \\
\qquad\quad b : \mathit{boolean} \textbf{ init } \mathit{false} \\
\quad \textbf{action } \mathit{acq} \quad :: \quad \neg b \ \& \ \mathit{Sem.P} \longrightarrow b := \mathit{true} \\
\quad \textbf{action } \mathit{rel} \quad :: \quad b \qquad\qquad\quad \longrightarrow y := y + 1 \ ; \ b := \mathit{false} \ ; \ \mathit{Sem.V} \\
\textbf{end}
\end{array}
$$

Figure 2.3: A *TCB* program with three boxes

We assume that the terminals $\langle id \rangle$ (for identifiers), $\langle type \rangle$ (for typenames), and $\langle exp \rangle$ (for expressions) are defined elsewhere.

2.3 Syntax for *TCB*

An example *TCB* program with three boxes is shown in Figure 2.3. The first box, *Sem*, implements a general semaphore. The box has a local *integer* variable n, which is initially 1, and two methods, P and V. Method P is partial; a call to this method accepts only if $n > 0$. If a call to $Sem.P$ accepts, then the semaphore has been granted to the caller. The method V releases the semaphore. This method is total, so it always accepts.

Boxes D and E show two different styles for using the semaphore. Box D contains a single action *act*. This is a partial action. The condition is *true*. The test

$$
\begin{array}{lcl}
\langle program \rangle & ::= & \langle box \rangle^{+} \\
\langle box \rangle & ::= & \text{“}\textbf{box}\text{”}\ \langle id \rangle\ \text{“}\textbf{var}\text{”}\ \langle decl \rangle^{*}\ \langle proc \rangle^{+}\ \text{“}\textbf{end}\text{”} \\
\langle proc \rangle & ::= & (\ \text{“}\textbf{action}\text{”}\ \mid\ \text{“}\textbf{method}\text{”}\ )\ \langle header \rangle\ \text{“::”} \\
 & & \quad (\ \langle partial \rangle\ \mid\ \langle total \rangle\ ) \\
\langle header \rangle & ::= & \langle id \rangle\ \text{“(”}\ [\ \text{“}\textbf{in}\text{”}\ \langle decl \rangle^{+}\ ]\ \text{“;”}\ [\ \text{“}\textbf{out}\text{”}\ \langle decl \rangle^{+}\ ]\ \text{“)”} \\
\langle partial \rangle & ::= & \langle altern \rangle^{+} \\
\langle altern \rangle & ::= & \langle boolexp \rangle\ [\ \text{“\&”}\ \langle call \rangle\ ]\ \text{“}\longrightarrow\text{”}\ \langle stmt \rangle^{*} \\
\langle total \rangle & ::= & \langle stmt \rangle^{*} \\
\langle stmt \rangle & ::= & \langle call \rangle\ \mid\ \langle assign \rangle\ \mid\ \langle ifelse \rangle\ \mid\ \langle whiledo \rangle\ \mid\ \langle comp \rangle \\
\langle call \rangle & ::= & \langle id \rangle\ \text{“.”}\ \langle id \rangle\ [\ \text{“(”}\ \langle exp \rangle^{*}\ \text{“;”}\ \langle id \rangle^{*}\ \text{“)”}\ ] \\
\langle assign \rangle & ::= & \langle id \rangle\ \text{“:=”}\ \langle exp \rangle \\
\langle ifelse \rangle & ::= & \text{“}\textbf{if}\text{”}\ \langle boolexp \rangle\ \text{“}\textbf{then}\text{”}\ \langle stmt \rangle\ \text{“}\textbf{else}\text{”}\ \langle stmt \rangle \\
\langle whiledo \rangle & ::= & \text{“}\textbf{while}\text{”}\ \langle boolexp \rangle\ \text{“}\textbf{do}\text{”}\ \langle stmt \rangle \\
\langle comp \rangle & ::= & \text{“|[”}\ \langle stmt \rangle^{*}\ \text{“]|”} \\
\langle decl \rangle & ::= & \langle id \rangle\ \text{“:”}\ \langle type \rangle\ [\ \text{“}\textbf{init}\text{”}\ \langle value \rangle\ ]
\end{array}
$$

Figure 2.4: Syntax for *TCB*

is a call to $Sem.P$ to obtain the semaphore. If the test accepts, box D increments x, and then calls $Sem.V$ to release the semaphore. Box E has two actions. The local *boolean* variable b indicates if the box holds the semaphore. The first action, *acq*, attempts to acquire the semaphore, if it does not already hold it. If it succeeds, it sets flag b to *true*. This enables action *rel*, which increments y, and then releases the semaphore, and resets b.

The syntax for *TCB* is shown in Figure 2.4. This syntax corresponds to the syntax we have been using for the examples so far. There is one slight extension: a procedure header, and a procedure call, has a semicolon between the input and output parameters. Thus we write a call to a method $D.m$ with input and output

parameters as $D.m(\tilde{e};\tilde{x})$, where $\tilde{e}$ is a sequence of expressions giving values for the input parameters, and $\tilde{x}$ is a sequence of variable ids to receive the output parameter values. We omit the semicolon on a header or a call if there is only one type of parameter. We omit the parenthesis if there are no parameters.

Note that the production for $\langle stmt \rangle$ has one alternative more than the equivalent production in the syntax of *TCBloc*. The extra alternative is the method call statement.

2.4 Well-formed programs

There are syntactically legal *TCBloc* and *TCB* programs that cannot be implemented. The syntax given ignores issues of variable scope, and type correctness, since these cannot easily to expressed in a BNF grammar.

For the remainder of this work, we consider only *well-formed* programs, that is, programs that respect scoping rules and are type correct. We do not give an exact definition of a well-formed program. Rather, we list the assumptions that we make in defining the semantics for well-formed programs.

We define the *scope* of a variable declaration in a *TCBloc* program to be all the statements. In *TCB*, the scope of a box variable is all the procedures in the box, and the scope of a procedure parameter is the procedure where it is declared.

We assume that each operator in the expression language has a *type signature*, giving the type of its operands and result. We assume that the expression language defines the set of *well-typed* expressions, and assigns a unique type to each. These are the expressions where every operator is used with expressions of the correct type for its operands, as given by the type signature. The type of a well-typed expression is the result type of the top-level operator. If e is an expression containing variables, then the type of e is defined relative to a *type assignment* a partial function from variable ids to types. If e contains a variable whose type is undefined, then it is not

well-typed.

For example, the expression "$\neg b \wedge (x = y)$" is well-typed, with respect to a type assignment that gives b type *boolean*, and x and y type *integer*, assuming the operators have the customary type signatures. The expression "$true + 17$" is not well-typed.

For each statement in a program, we define a type assignment mapping x to T for every declaration "$x : T$" whose scope includes the statement. Any expression in the statement is typed relative to this type assignment.

Assumption 2.2 *For a well-formed program,*

1. *Every variable is used in the scope of a declaration.*

2. *Every expression is well-typed.*

3. *For every assignment statement, the expression of the right-hand side has the same type as the variable on the left-hand side.*

4. *The expression if every if-else and while-do statement is of type boolean.*

5. *Every method call statement in TCB is to a method declared in another box, and the types of expressions for the input parameters, and of the variables for the output parameters, matches the declared parameter types for the method.*

6. *Actions have no parameters.*

For $e \in Exp$, and $x \in Id$, we say that e *matches* x if the type of e matches the type of variable x. We use this only in a context where the types of e and x is understood, that is, in the scope of the necessary declarations. We extend this to sequences in the obvious way. For $\tilde{e} \in Exp^*$, and $\tilde{x} \in Id^*$, we say that $\tilde{e}$ *matches* $\tilde{x}$ if $\tilde{e}$ and $\tilde{x}$ are the same length, and each expression in $\tilde{e}$ matches the corresponding variable in $\tilde{x}$.

2.5 Run-time errors

Run-time errors describe steps of a program caused by the failure to evaluate an expression in the current state. Often, a run-time error is defined for division by zero, or accessing an array outside its bounds. We choose not to define any run-time errors for *TCB*, since we thereby avoid adding to the complexity of the semantics. That is, we make the following assumption.

Assumption 2.3 *A TCB expression e is defined in every state σ that assigns a value of the correct type to every variable in e.*

We indicate extensions to the semantics and to the complete execution theorem that are required if we do not make this assumption.

2.6 Operational semantics

We give semantic definitions in the *Structured Operational Semantics* style introduced by Plotkin [29]. We choose this style because it allows us to present the semantics in a way that closely models the informal explanation of the execution of *TCB* programs given in earlier chapters.

For each semantic definition, we first define a *configuration* that includes (at least) a state, and a sequence of program statements (the *code*). A configuration represents a "snapshot" of a system. The state gives values to the program variables, and the code is the part of the program code left to execute.

The execution of a program steps from configuration to configuration as statements are executed, and the state is updated. We use function update to describe the changes to the state. For a state σ, the state

$$\sigma' \quad = \quad \sigma\,[x \mapsto v]$$

is defined by

$$\sigma'.x \quad = \quad v$$

and, for y any identifier other than x,

$$\sigma'.y \quad = \quad \sigma.y$$

For $\sigma \in State$, and $\theta \in Statement^*$, let $\mathbf{C} = (\sigma, \theta)$ be a configuration. The semantic rules define a transition relation on configurations. Configuration $\mathbf{C}$ is related to $\mathbf{C}' = (\sigma', \theta')$ if the semantics allows a step from $\mathbf{C}$ to $\mathbf{C}'$. We call $\mathbf{C}'$ a *successor* of $\mathbf{C}$. The rules defining the relation generally depend on the first statement in θ, and on the state σ. Component θ' is the updated code after the first statement has been executed, and the σ' is σ updated with the effect of the statement.

An execution of a program starts from a configuration $\mathbf{C}_0 = (\sigma_0, \theta_0)$ in which σ_0 maps each variable to its initial value, and θ_0 is the whole program. Execution proceeds by taking a step from $\mathbf{C}_0$ to $\mathbf{C}_1$, where $\mathbf{C}_1$ is a successor of $\mathbf{C}_0$, and then taking a step from $\mathbf{C}_1$ to $\mathbf{C}_2$, one of its successors, and continuing in this way. If an execution reaches a configuration with no successors in the transition relation, the execution stops. There are no successors for configurations with an empty code component. In such a configuration, there is no more code to execute. An execution that reaches such a configuration has *terminated*.

The sequential language *TCBloc* is very simple, and its operational semantics is corresponding simple. The semantic definition serves mainly as an introduction to the Structured Operational Semantics style before we tackle concurrent execution.

The semantics for *TCBtot* and *TCB* represents an executing program a system comprised of separately executing nodes, one for each box. Each node executes calls to procedures from its associated box. We identify each box in the program

with its node, and use the name "box" to mean either, depending on context. A box that is not currently executing a procedure call is called *quiescent*. Initially, all boxes are quiescent.

The execution of a thread for action $D.a$ begins with box D starting to execute a's code. If the execution of $D.a$ reaches a call to method $E.m$, this is executed by suspending execution at D, and starting execution at E. When the execution of $E.m$'s code is complete, D becomes active again, and E becomes quiescent.

In a *TCBtot* program, every procedure is total, so every procedure call during an execution is accepting. The semantics is simplified by not having to deal with rejecting procedure calls.

We implement a simple model of procedure call for *TCBtot*. In this model, a procedure call step occurs when a box D has a call to $E.m$ as the first statement in its code, and E is quiescent. If E is executing a procedure call, D is blocked until this call completes. When E is quiescent, D is able to make the call. The configurations of D and E are updated in the same step: D enters a waiting phase, and E is updated with the call from D. Box E is then committed to executing this call. When it completes the call, the configurations of D and E are again updated in a single step: D is updated with the values of the output parameters from E, and continues executing the rest of its code, and E becomes quiescent again. We call this model of procedure call a *rendezvous* between the source and the agent; the step is enabled only if both boxes are in the right configuration. We call the semantics for *TCBtot* a *rendezvous semantics.*

The first semantics we define for the full *TCB* language uses the rendezvous model for procedure call. This extends the ideas from the *TCBtot* semantics, and implements guard evaluation and accepting and rejecting calls to partial procedures.

The rendezvous model for procedure call is inadequate for cases when there is contention for access to a box. Suppose procedure calls are executing concurrently

at boxes D and D', and both are ready to call a method on box E. Box D cannot proceed until E executes its method call, and similarly for D'. When E is idle, it can start executing only one of the calls, say the one from D'. Box D waits while E executes the call from D'. When E is again idle, it may start D's call, or it may start a call from yet another box. If there are enough boxes trying to call methods on E, there may be contention each time E is quiescent, and box D may always lose this contention.

An execution with a nonterminating thread cannot be represented by a sequential execution. To get an execution in which all threads terminate, we must ensure that every procedure call is eventually executed by its agent. The above example shows that the rendezvous semantics does not guarantee this. We introduce a different model for procedure call, one that uses a *call queue* for each box.

In the new model, a box D reaches a call to a procedure on box E, an entry is placed at the back of E's call queue, regardless of E's configuration. Box D enters a waiting state. When E is idle and there is an entry in its call queue, it starts executing the front entry. When it completes the call it returns any output parameters to the call's source, and the source continues executing. The front entry is removed from the queue, and E becomes idle, so it again checks to see if the queue is empty. Assuming each procedure call terminates, the call from D eventually reaches the front of the queue, and E executes it.

The queue model ensures that no box is permanently prevented from accessing a box to execute a method. We call a semantics using this model a *queue semantics*.

Note that the rendezvous form for procedure return does not cause similar contention problems, since we restrict our attention to *well-formed* program configurations, one of whose characteristics is that if an agent is ready to return from a method call, then the source of the call is waiting, ready to execute the return.

Our final semantics is a queue semantics for the full *TCB* language. We extend the program configurations from the rendezvous semantics to include the call queues, and we define a transition relation on these extended configurations. This semantics is the one which we use as the semantics of *TCB* in later chapters.

2.7 Operational semantics for *TCBloc*

We define the semantics of a *TCBloc* program, using the Structured Operational Semantics style.

2.7.1 Program configuration for *TCBloc*

The configuration of a *TCBloc* program is given by the type *LocConfig*.

Definition 2.4

$$
\begin{array}{lcl}
\mathit{LocConfig} & \triangleq & \textbf{record} \\
 & & \quad \sigma : \mathit{State} \\
 & & \quad \theta : \mathit{Statement}^* \\
 & & \textbf{end}
\end{array}
$$

Component σ is the value of the program variables, and component θ is the sequence of statements — the code — remaining to be executed.

2.7.2 Semantic rules for *TCBloc*

The semantics for this language is given as a set of inference rules which together define a relation $\longrightarrow$ over *LocConfig*. These inference rules are shown in Figure 2.5. The intended operational meaning of these rules is as follows.

(assign) If $\text{first}(\theta)$ is an assignment statement, evaluate the right-hand side of the assignment in the current state, and update the state to give the variable of

(assign)
$$\frac{\theta = (x := e);\theta' \qquad \sigma' = \sigma\,[x \mapsto [\![e]\!]\sigma]}{(\sigma,\theta) \longrightarrow (\sigma',\theta')}$$

(block)
$$\frac{\theta = |[\ \theta_0\ ]|;\theta_1 \qquad \theta' = \theta_0;\theta_1}{(\sigma,\theta) \longrightarrow (\sigma,\theta')}$$

(if-true)
$$\frac{\theta = (\textbf{if}\ \ b\ \ \textbf{then}\ \ S_0\ \ \textbf{else}\ S_1);\hat{\theta} \qquad [\![b]\!]\sigma = \mathit{true} \qquad \theta' = S_0;\hat{\theta}}{(\sigma,\theta) \longrightarrow (\sigma,\theta')}$$

(if-false)
$$\frac{\theta = (\textbf{if}\ \ b\ \ \textbf{then}\ \ S_0\ \ \textbf{else}\ S_1);\hat{\theta} \qquad [\![b]\!]\sigma = \mathit{false} \qquad \theta' = S_1;\hat{\theta}}{(\sigma,\theta) \longrightarrow (\sigma,\theta')}$$

(while-true)
$$\frac{\theta = (\textbf{while}\ \ b\ \ \textbf{do}\ \ S);\hat{\theta} \qquad [\![b]\!]\sigma = \mathit{true} \qquad \theta' = S;(\textbf{while}\ \ b\ \ \textbf{do}\ \ S);\hat{\theta}}{(\sigma,\theta) \longrightarrow (\sigma,\theta')}$$

(while-false)
$$\frac{\theta = (\textbf{while}\ \ b\ \ \textbf{do}\ \ S);\theta' \qquad [\![b]\!]\sigma = \mathit{false}}{(\sigma,\theta) \longrightarrow (\sigma,\theta')}$$

Figure 2.5: Semantics for *TCBloc*

the left-hand side of the assignment the value computed; continue executing the rest of the program after the assignment.

(block) If first(θ) is a block statement, prepend the statements in the block to the code, without changing the state.

(if-true)

(if-false) If first(θ) is an if-else statement, first evaulate the condition expression in the current state; if it is *true*, execute the then-statement, if it is *false*, execute the else-statement; afterwards, continue with the rest of the program.

(while-true)

(while-false) If first(θ) is a while-do statement, first evaluate the condition expression in the current state; if it is *true*, execute the body of the loop, and then execute the loop again, if it is *false*, continue with the rest of the program after the loop.

An execution of a *TCBloc* program starts from a configuration (σ_0, θ_0), where σ_0 is the initial state, and θ_0 is the whole code of the program. State σ assigns values to all the variables defined for the program, and to no others.

From the initial configuration (σ_0, θ_0), the execution takes a step to configuration (σ_1, θ_1), where $(\sigma_0, \theta_0) \longrightarrow (\sigma_1, \theta_1)$, and then it takes a step to (σ_2, θ_2), where $(\sigma_1, \theta_1) \longrightarrow (\sigma_2, \theta_2)$. The execution continues in this manner. If execution reaches a configuration (σ_n, θ_n) where $\theta_n = \perp$, no further transitions are possible, and we say that the execution has *terminated*, and σ_n is the *final state*.

Note that none of the semantic rules extend the domain of the state component of the configuration. In this simple programming language all variables are global, and last the lifetime of the program.

The semantics given is standard, and we do not investigate it further since it is not the focus of our work. The semantics of the *TCBloc* have little effect on

results that we prove for *TCB*. The main feature of this semantics we use in the remainder of our work is that it that for every (σ, θ), where $\theta \neq \perp$, there is a (σ', θ') such that $(\sigma, \theta) \longrightarrow (\sigma', \theta')$.

The results that we show can be extended to a Seuss language using *any* sequential language for the bodies of the procedures, not just *TCB*. We chose *TCBloc* as a simple representative sequential language.

See [29] for examples of Structured Operational Semantics definitions of languages with local variables, subroutines, and dynamic allocation. For our purposes, a richer sequential language is unnecessary.

2.8 Defining the semantics for *TCB*

We discuss below a number of the design decisions that were made in defining the semantics for *TCB*.

2.8.1 Modelling boxes

As noted in the introduction, the configuration of a *TCB* program consists of the configuration of each box in the program. The box configuration has several components. As with *TCBloc*, it includes the local state, which gives the values of all the variables declared in the box, and some code, which, in this case, is the statements remaining to be executed for the current procedure call. The other components are the *phase*, and the *call information*.

Execution for each box consists of a sequence of complete procedure calls. The *phase* for a box records where the box stands in the cycle of waiting to start execution of a procedure call, deciding to accept or reject the call, executing the procedure body, and so on. The phase is the state for a finite state automaton controlling this aspect of the box's execution.

The *call information* records information about the currently executing procedure call. This is needed for, among other things, determining the values to return to the source on return from a procedure call.

During the execution of a *TCB* program, the box configurations are interrelated. For example, if box D is executing a method call with source E, then we expect E to be waiting for control to return from the method call. In Chapter 3, we define the set of *well-formed* box configuration and program configurations. These are the only configurations we expect to see during the execution of a program. We show that an execution starting from a well-formed configuration never reaches a configuration that is not well-formed.

The program configuration we define allows for independent activity at each box. The model allows an action call to be started when other action calls are running. Each action call is a separate thread. The model is one in which there can be multiple threads active at one time.

Note The three semantics that we introduce share many similarities, and we define similar but different types and inference rules for different semantics. We adopt a naming convention (with the aid of hindsight) for these entities, as follows.

- If an entity is used in the semantics of *TCBtot*, but not in the rendezvous semantics of *TCB*, it is named with the suffix *-tot*.
- If an entity is used in the rendezvous semantics of *TCB*, but not in the queue semantics of *TCB*, it is named with the suffix *-rdv*.
- If an entity is used in the queue semantics of *TCB*, it is named with neither of these suffixes.

So, for example, *ProgConfig-tot* is the program configuration for the *TCBtot* semantics; *ProgConfig-rdv* is that for the rendezvous semantics of *TCB*; and *ProgConfig* is

that for the queue semantics of *TCB*. In the *TCBtot* semantics, we use names with both suffixes, and with none; in the rendezvous semantics for *TCB* we use names with the suffix *-rdv*, and names with no suffix; in the queue semantics for *TCB* we use only names with no suffix. (End of note)

2.9 Rendezvous semantics for *TCBtot*

We present a rendezvous semantics for *TCBtot*. In the semantics, we need to refer to information about boxes and procedures from the program. When a procedure call is initialized, for example, the semantics uses the names of input parameters declared for that procedure to correctly update the state with the input parameter values. We define some types for this static information.

2.9.1 Static information for a *TCBtot* box

We define a type for the information about procedures defined in a box.

Definition 2.5

$$
\begin{aligned}
\mathit{TotalDef} \triangleq\ & \textbf{record} \\
& \quad I : \mathit{VarTypeList} \\
& \quad O : \mathit{VarTypeList} \\
& \quad C : \mathit{Statement}^* \\
& \textbf{end}
\end{aligned}
$$

Here I represents the input parameters and their types, O represents the output parameters and their types, and C represents the code for the body of the method. For an action, both I and O are empty lists. We use *VarTypeList* rather than a *State* for the input and output parameters because we use positional matching to pass multiple actual parameter values to multiple formal parameters. That is, if we

have the parameter list $\langle(x, \mathit{Integer}), (y, \mathit{Integer})\rangle$, and we have the parameter value list $\langle 5, 9\rangle$, then we give x the value 5, and y the value 9. The translation of the code of a well-formed *TCBtot* procedure to a *TotalDef* is straightforward.

The type *BoxInfo-tot* records the static information for a box.

Definition 2.6

$$
\begin{array}{lcl}
\mathit{BoxInfo\text{-}tot} & \triangleq & \textbf{record} \\
& & \quad \sigma_0 : \mathit{State} \\
& & \quad T : \mathit{Id} \hookrightarrow \mathit{TotalDef} \\
& & \quad A : \mathrm{dom}(T) \to \mathit{Bool} \\
& & \textbf{end}
\end{array}
$$

Component σ_0 represents the initial state, which assigns initial values of the appropriate type to the box variables, T represents the total procedures defined in the box (for the present case, all procedures are total), and A indicates which of the procedures in $\mathrm{dom}(T)$ are actions; $A.p$ holds if and only if procedure p is an action.

Again, translating the code of a *TCBtot* box into a *BoxInfo-tot* is a straightforward process. Translating the well-formedness conditions on *TCBtot* programs to the equivalent conditions on the generated *BoxInfo-tot*, we assume the following about any *BoxInfo-tot*.

Assumption 2.7 *If* (σ_0, T, A) *is a BoxInfo-tot generated from the code of a well-*

formed TCBtot box,

$$
\begin{array}{l}
\langle \forall\ p \\
\quad :\ p \in \mathrm{dom}(T) \\
\quad :\ (\ A.p \ \Rightarrow\ T.p = (\bot, \bot, C)\ ) \quad \wedge \\
\qquad \mathrm{dom}(\sigma_0) \quad \mathrm{disj} \quad \mathrm{dom}(T.p.I) \quad \wedge \\
\qquad \mathrm{dom}(\sigma_0) \quad \mathrm{disj} \quad \mathrm{dom}(T.p.O) \quad \wedge \\
\qquad \mathrm{dom}(T.p.I) \quad \mathrm{disj} \quad \mathrm{dom}(T.p.O) \\
\quad \rangle
\end{array}
$$

The first conjunct in the term states that actions have no input or output parameters. The remaining conjuncts require that the names of the input parameters, and the output parameters be distinct from each other and from the names of the box variables. This restriction means that we do not have to deal with the case when the name of a parameter hides the name of a box variable. This simplifies the definition of the semantic rules.

2.9.2 Static infomation for a *TCBtot* program

Let **B** be the set of boxes in the program. The static information associated with a *TCB* program is represented by a map **I** that maps each box to its associated *BoxInfo-tot*.

$$\mathbf{I} : \mathbf{B} \to \mathit{BoxInfo\text{-}tot}$$

This map is generated from the program text. When we use the following functions, which use **I** implicitly, we do so in a context in which there is an identified program under consideration, so the value of **I** is given.

Definition 2.8 *For* $D \in \mathbf{B}$,

$$\begin{aligned}
\mathit{InitState}(D) &\triangleq \mathbf{I}.D.\sigma_0 \\
\mathit{BoxVars}(D) &\triangleq \mathrm{dom}(\mathit{InitState}(D)) \\
\mathit{TotActs}(D) &\triangleq \{\, p \mid p \in \mathrm{dom}(\mathbf{I}.D.T) \wedge \mathbf{I}.D.A.p \,\} \\
\mathit{TotMeths}(D) &\triangleq \{\, p \mid p \in \mathrm{dom}(\mathbf{I}.D.T) \wedge \neg \mathbf{I}.D.A.p \,\}
\end{aligned}$$

For $p \in \mathrm{dom}(\mathbf{I}.D.T)$,

$$\begin{aligned}
\mathit{InParam}(D.p) &\triangleq \mathit{VarList}(\mathbf{I}.D.T.p.I) \\
\mathit{OutParam}(D.p) &\triangleq \mathit{VarList}(\mathbf{I}.D.T.p.O) \\
\mathit{Code}(D.p) &\triangleq \mathbf{I}.D.T.p.C
\end{aligned}$$

2.9.3 Box configuration for *TCBtot*

The type *BoxConfig-tot* is used to represent the configuration of a *TCBtot* box during the execution of a program. To define this we first define a type to represent the *phase* of an executing box. At each point during the execution of a program, a box is in one of the following phases.

- Not currently executing a procedure call.
- Executing the local code of a procedure call.
- Waiting for a call on a method in another box to return.
- Ready to return control and output parameter values to the source.

The following type represents the phase.

Definition 2.9

$$\mathit{BoxPhase\text{-}tot} \triangleq \{\text{IDLE}, \text{ACCEPT}, \text{WAIT}, \text{RETURN}\}$$

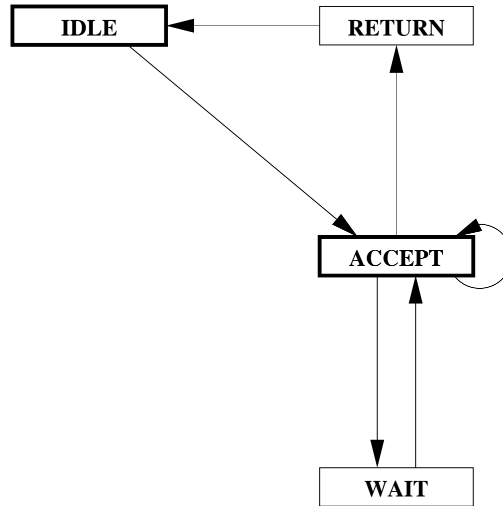

Figure 2.6: Transition diagram for phases of a *TCBtot* box.

These values correspond, in order, to the phases listed above. The diagram in Figure 2.6 shows the possible phase changes for a single box.

The type *CallInfo* is used to record information regarding a procedure call.

Definition 2.10

$$
\begin{aligned}
\mathit{CallInfo} \;\triangleq\; & \textbf{record} \\
& \quad p : \mathit{Id} \\
& \quad Q : \mathbf{B}_{\bot} \\
& \quad \tilde{v} : \mathit{Val}^{*} \\
& \textbf{end}
\end{aligned}
$$

Here p is the name of the procedure, and Q is the source. We use the $\bot$ for the source of an action. The sequence $\tilde{v}$ is used to hold values for output parameters (and, in other semantic definitions, values for input parameters). During the execution of a procedure call it is always $\bot$, until the box enters phase RETURN, at which point

it is set to the output parameter values. Since actions have caller $\perp$, and have no parameters, a *CallInfo* for an action a is always of the form $(a, \perp, \perp)$. We drop the trailing $\perp$ values and write this as (a). We also write (p, D) for $(p, D, \perp)$. The following functions are accessors for components of a *CallInfo*.

Definition 2.11 *For $\gamma \in$ CallInfo, where $\gamma = (p, Q, \tilde{v})$,*

$$\begin{aligned} Proc(\gamma) &\triangleq p \\ Source(\gamma) &\triangleq Q \end{aligned}$$

Now we define the type that represents the configuration of a box during program execution.

Definition 2.12

$$\begin{aligned} \mathit{BoxConfig\text{-}tot} \triangleq\ & \textbf{record} \\ & \quad \phi : \mathit{BoxPhase\text{-}tot} \\ & \quad \sigma : \mathit{State} \\ & \quad \gamma : \mathit{CallInfo}_{\perp} \\ & \quad \theta : \mathit{Statement}^{*} \\ & \textbf{end} \end{aligned}$$

In our intended model of program execution, when a box is idle, or ready to return, it has no code to execute, that is, $\theta = \perp$. So, for

$$\phi \in \{\text{IDLE}, \text{RETURN}\}$$

we let

$$(\phi, \sigma, \gamma) = (\phi, \sigma, \gamma, \perp)$$

2.9.4 Program configuration for *TCBtot*

The configuration of a program is given by a *BoxConfig-tot* for each box in it.

Definition 2.13

$$ProgConfig\text{-}tot \triangleq \mathbf{B} \rightarrow BoxConfig\text{-}tot$$

The program starts in a quiescent configuration where every box has its initial state.

Definition 2.14 *The* initial configuration *for the program is* $\mathbf{C}_{init} \in ProgConfig\text{-}tot$ *such that, for all* $D \in \mathbf{B}$

$$\mathbf{C}_{init}.D = (\textsc{idle}, InitState(D), \perp)$$

2.9.5 Semantic rules for *TCBtot*

The transition relation for *TCBtot* is written $\Longrightarrow$. As with *TCBloc*, we define this relation using a set of inference rules.

Action start, method call, and procedure initialization

The inference rules in Figure 2.7 concern procedure calls. An explanation of these rules follows.

(t-action-start-rdv) A box in phase IDLE, with an empty call information, can start a call of any of its actions. It sets its call information for the action, its code to the code for the body of the action, and enters phase ACCEPT.

(total-call-rdv) A box in phase ACCEPT that has a method call as the first statement of its code can execute the call when the agent is in phase IDLE, and its call information is empty. The source evaluates the actual values for the input parameters, leaves the method call at the head of its code, and enters phase

(t-action-start-rdv)

$$\frac{\begin{array}{r@{\;}l} \mathbf{C}.D &= (\textsc{idle}, \sigma, \bot) \\ a &\in \mathit{TotActs}(D) \\ \gamma' &= (a) \\ \theta' &= \mathit{Code}(D.a) \end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma, \gamma', \theta')]}$$

(total-call-rdv)

$$\frac{\begin{array}{r@{\;}l} \mathbf{C}.D &= (\textsc{accept}, \sigma_0, \gamma_0, \theta_0) \\ \theta_0 &= E.m(\tilde{e}; \tilde{x}); \hat{\theta} \\ \mathbf{C}.E &= (\textsc{idle}, \sigma_1, \bot) \\ \tilde{v} &= [\![\tilde{e}]\!]\sigma_0 \\ \tilde{y} &= \mathit{InParam}(E.m) \\ \tilde{z} &= \mathit{OutParam}(E.m) \\ \sigma_1' &= \sigma\,[\tilde{z} \mapsto \tilde{\bot}]\,[\tilde{y} \mapsto \tilde{v}] \\ \gamma_1' &= (m, D) \\ \theta_1' &= \mathit{Code}(E.m) \end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\textsc{wait}, \sigma_0, \gamma_0, \theta_0) \\ E \mapsto (\textsc{accept}, \sigma_1', \gamma_1', \theta_1')) \end{bmatrix}}$$

Figure 2.7: Semantics for *TCBtot*: action start and method call

$$\textbf{(local-step)}\quad \frac{\mathbf{C}.D = (\textsc{accept}, \sigma, \gamma, \theta) \qquad (\sigma, \theta) \longrightarrow (\sigma', \theta')}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma', \gamma, \theta')]}$$

$$\textbf{(proc-term-rdv)}\quad \frac{\begin{array}{c}\mathbf{C}.D = (\textsc{accept}, \sigma, \gamma, \bot) \\ \gamma = (p, Q) \\ \tilde{z} = \mathit{OutParam}(D.p) \\ \sigma' = \sigma \restriction \mathit{BoxVars}(D) \\ \tilde{v} = [\![\tilde{z}]\!]\sigma \\ \gamma' = (p, Q, \tilde{v})\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{return}, \sigma', \gamma')]}$$

Figure 2.8: Semantics for *TCBtot*: procedure body execution.

WAIT. The agent extends its state with the input and output parameters, initializes the input parameters according to the values from the source, saves the name of the source and the method in its call information, sets its code to the code for the body of the method, and enters phase ACCEPT.

Note that the rule **(t-action-start-rdv)** says nothing about scheduling of actions: it does not require, for example, that there is only one action active at a time, nor does it require that all actions in a box are given a chance to run at some point during an execution. As we noted in the introduction, these issues are handled separately from the semantics.

Procedure body execution

The inference rules in Figure 2.8 concern the execution of a procedure body. An explanation of these rules follows.

(local-step) A box in phase ACCEPT takes a local step if there is a *TCBloc* transition defined for its values of σ and θ. The step updates σ and θ as in the

(action-end-rdv)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\text{RETURN}, \sigma, \gamma) \\ \gamma = (a)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\text{IDLE}, \sigma, \bot)]}$$

(total-return-rdv)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\text{WAIT}, \sigma_0, \gamma_0, \theta_0) \\ \theta_0 = E.m(\tilde{e}; \tilde{x}); \theta'_0 \\ \mathbf{C}.E = (\text{RETURN}, \sigma_1, \gamma_1) \\ \gamma_1 = (m, D, \tilde{v}) \\ \sigma'_0 = \sigma_0\,[\tilde{x} \mapsto \tilde{v}]\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\text{ACCEPT}, \sigma'_0, \gamma_0, \theta'_0) \\ E \mapsto (\text{IDLE}, \sigma_1, \bot) \end{bmatrix}}$$

Figure 2.9: Semantics for *TCBtot*: procedure return

TCBloc transition, and leaves the other components of the box's configuration unchanged. Note that a *TCBloc* transition is defined only if the first statement of θ is a statement of *TCBloc* (that is, it is not a method call).

(proc-term-rdv) A box in phase ACCEPT that has no code left to execute (so it has completed the execution of the current procedure call) evaluates the values of the output parameters in its current state, puts these in the call information, removes from its state any values for variables other than the box variables, and enters phase RETURN.

Return from a procedure call

The inference rules in Figure 2.9 concern the return of control to the source at the end of a procedure call. An explanation of these rules follows.

(action-end-rdv) A box in phase RETURN that has an action call in its call information ends the action. The box clears its call information and enters phase IDLE.

(total-return-rdv) A box in phase RETURN that has a method call in its call information ends the call, returning the output parameter values to the source. The agent clears its call information and enters phase IDLE. The source updates its state with the output parameter values and reënters phase ACCEPT.

2.10 Rendezvous semantics for *TCB*

In this section, we extend the rendezvous semantics for *TCBtot* to a rendezvous semantics for the full *TCB* language. Since the difference between these languages is the absence of partial procedures in *TCBtot*, the extensions all concern the evaluation of guards, and handling rejected procedure calls.

2.10.1 Static infomation for a *TCB* box

A partial procedure consists of a set of alternatives, each of which has a guard, consisting of a *condition*, which is a boolean expression on the local state, including any procedure parameters, and, optionally, a *test*, which is a call to a partial method. We require that the conditions on the alternatives in a single procedure be disjoint, so any value for the local state satisfies the condition for at most one alternative. To model a partial procedure, we use a function with the type *Alternative*.

Definition 2.15

$$Alternative \triangleq State \hookrightarrow (ProcCall_{\perp} \times Statement^{*})_{\perp}$$

Note that a function in *Alternative* is partial, since it is only defined for states that give values to all variables needed to evaluate the conditions and the tests. As an

example, consider the following partial procedure with two alternatives.

$$\begin{array}{lcll} \pi & :: & b_0 \ \& \ E.m(\tilde{x};\tilde{v}) & \longrightarrow \theta_0 \\ & | & b_1 & \longrightarrow \theta_1 \end{array}$$

The *Alternative* function for this procedure is the function G, where, for all σ such that b_0 and b_1 are defined in σ,

$$\begin{array}{lcll} G.\sigma & \triangleq & (E.m(\tilde{x};\tilde{v}), \theta_0) & \text{if } [\![b_0]\!]\sigma \\ & & (\bot, \theta_1) & \text{if } [\![b_1]\!]\sigma \\ & & \bot & \text{if } \neg[\![b_0]\!]\sigma \wedge \neg[\![b_1]\!]\sigma \end{array}$$

When condition b_0 holds, G returns the test call and the body code from the first alternative; when b_1 holds, G returns an empty test call and the body code from the second alternative; and when neither condition is true, G returns $\bot$. To execute a call on this procedure, we check the value of $G.\sigma$. If it is $\bot$, the call rejects, and control returns to the source; if it is $(\bot, \theta)$, the call accepts, and θ is executed before control returns to the source; and if it is $(E.m(\tilde{x};\tilde{v}), \theta)$, control is passed to box E to execute the test call, and when that call ends, if it rejects, then the current call rejects, and control returns to the source, and if it accepts, then the current call accepts, and θ is executed before control returns to the source.

We now define a type to represent a partial procedure.

Definition 2.16

$$\begin{array}{lcl} \mathit{PartialDef} & \triangleq & \textbf{record} \\ & & \quad I : \mathit{VarTypeList} \\ & & \quad O : \mathit{VarTypeList} \\ & & \quad G : \mathit{Alternative} \\ & & \textbf{end} \end{array}$$

Here I and O are as in *TotalDef*, and G is a function used for evaluating the guards and selecting the appropriate test and body code, as described above.

We next define the type for the static information for a *TCB* box.

Definition 2.17

$$
\begin{aligned}
\mathit{BoxInfo} \;\triangleq\; & \textbf{record} \\
& \quad \sigma_0 : \mathit{State} \\
& \quad P : \mathit{Id} \hookrightarrow \mathit{PartialDef} \\
& \quad T : \mathit{Id} \hookrightarrow \mathit{TotalDef} \\
& \quad A : (\mathrm{dom}(P) \cup \mathrm{dom}(T)) \rightarrow \mathit{Bool} \\
& \textbf{end}
\end{aligned}
$$

Components σ_0 and T are as in the definition of *BoxInfo-tot*. Component P represents the partial procedures defined in the box, and predicate A is extended to $\mathrm{dom}(P)$.

Each box in a program is translated into a *BoxInfo*, and from the well-formedness conditions on *TCB* programs, we get the following results.

Assumption 2.18 *If* (σ_0, P, T, A) *is a BoxInfo generated from the code of a well-formed TCB box, then*

$$
\begin{array}{l}
\mathrm{dom}(P) \;\; \mathrm{disj} \;\; \mathrm{dom}(T) \;\wedge \\
\langle \forall \;\; X, p \\
\quad : \; X \in \{P, T\} \;\; \wedge \;\; p \in \mathrm{dom}(X) \\
\quad : \; (\;\; A.p \;\; \Rightarrow \;\; X.p.I = \bot \;\; \wedge \;\; X.p.O = \bot \;\;) \;\; \wedge \\
\qquad \mathrm{dom}(\sigma_0) \quad\quad \mathrm{disj} \;\; \mathrm{dom}(X.p.I) \quad\quad \wedge \\
\qquad \mathrm{dom}(\sigma_0) \quad\quad \mathrm{disj} \;\; \mathrm{dom}(X.p.O) \quad\quad \wedge \\
\qquad \mathrm{dom}(X.p.I) \;\; \mathrm{disj} \;\; \mathrm{dom}(X.p.O) \\
\quad \rangle
\end{array}
$$

2.10.2 Static information for a *TCB* program

As with *TCBtot*, the static information for a *TCB* program is a function that maps each box to a *BoxInfo*.

$$\mathbf{I} : \mathbf{B} \rightarrow \mathit{BoxInfo}$$

We use **I** implicitly via the functions in the following definition.

Definition 2.19 *For $D \in \mathbf{B}$,*

$$\begin{array}{rcll}
\mathit{InitState}(D) & \triangleq & \mathbf{I}.D.\sigma_0 & \\
\mathit{PartActs}(D) & \triangleq & \{\ p \mid p \in \mathrm{dom}(\mathbf{I}.D.P) \wedge \mathbf{I}.D.A.p\ \} & \\
\mathit{PartMeths}(D) & \triangleq & \{\ p \mid p \in \mathrm{dom}(\mathbf{I}.D.P) \wedge \neg\mathbf{I}.D.A.p\ \} & \\
\mathit{TotActs}(D) & \triangleq & \{\ p \mid p \in \mathrm{dom}(\mathbf{I}.D.T) \wedge \mathbf{I}.D.A.p\ \} & \\
\mathit{TotMeths}(D) & \triangleq & \{\ p \mid p \in \mathrm{dom}(\mathbf{I}.D.T) \wedge \neg\mathbf{I}.D.A.p\ \} & \\
\mathit{Actions}(D) & \triangleq & \mathit{PartActs}(D) \cup \mathit{TotActs}(D) & \\
\mathit{Methods}(D) & \triangleq & \mathit{PartMeths}(D) \cup \mathit{TotMeths}(D) & \\
\mathit{Partials}(D) & \triangleq & \mathit{PartActs}(D) \cup \mathit{PartMeths}(D) & \\
\mathit{Totals}(D) & \triangleq & \mathit{TotActs}(D) \cup \mathit{TotMeths}(D) & \\
\mathit{Procs}(D) & \triangleq & \mathit{Actions}(D) \cup \mathit{Methods}(D) & \\
\mathit{InParam}(D.p) & \triangleq & \mathit{VarList}(\mathbf{I}.D.P.p.I) & \text{if } p \in \mathrm{dom}(\mathbf{I}.D.P) \\
 & & \mathit{VarList}(\mathbf{I}.D.T.p.I) & \text{if } p \in \mathrm{dom}(\mathbf{I}.D.T) \\
\mathit{OutParam}(D.p) & \triangleq & \mathit{VarList}(\mathbf{I}.D.P.p.O) & \text{if } p \in \mathrm{dom}(\mathbf{I}.D.P) \\
 & & \mathit{VarList}(\mathbf{I}.D.T.p.O) & \text{if } p \in \mathrm{dom}(\mathbf{I}.D.T) \\
\mathit{Alt}(D.p,\sigma) & \triangleq & \mathbf{I}.D.P.p.G.\sigma & \text{if } p \in \mathrm{dom}(\mathbf{I}.D.P) \\
\mathit{Code}(D.p) & \triangleq & \mathbf{I}.D.T.p.C & \text{if } p \in \mathrm{dom}(\mathbf{I}.D.T)
\end{array}$$

Note that $\mathit{Alt}(D.p,\sigma)$ is defined only for partial $D.p$, and $\mathit{Code}(D.p)$ is defined only for total $D.p$.

2.10.3 Box configuration for *TCB*

To implement the evaluation of the guard on partial procedures, we need to extend the set of phases. The type *BoxPhase* extends the type *BoxPhase-tot*.

Definition 2.20

$$\mathit{BoxPhase} \triangleq \mathit{BoxPhase\text{-}tot} \cup \{\textsc{guard}, \textsc{pwait}, \textsc{reject}\}$$

The phases *BoxPhase-tot* as used as before. A box is in phase GUARD when it has committed to executing a partial procedure, but has yet to decide whether to accept or to reject. A box in phase PWAIT has called a test, and is waiting for the result of the call. A box in phase REJECT has failed to successfully evaluate a guard (either no alternative has a condition that holds in the current state, or an alternative was selected, but the test call rejected) and the box is ready to return control to the source. A procedure call that ends in phase REJECT is a rejecting call. Conversely, one that ends in phase RETURN is accepting (so every call in *TCBtot* accepts, as expected).

The diagram in Figure 2.10 shows the phase changes for a single box. Some of the transitions in the diagram are only possible when executing a partial procedure, and some only when executing a total procedure. The diagram is isomorphic to that in Figure 2.6 on the phases in *BoxPhase-tot*.

The phases IDLE, ACCEPT, and REJECT are emphasized in the transition diagram, because these phases play an important rôle in the semantics. A box in phase IDLE is between the execution of two procedure calls. We expect each box to be in this phase repeatedly during an execution.

From the diagram, we can see that once a box enters phase ACCEPT while executing a procedure, it can only return to phase IDLE via phase RETURN, which indicates that the call was accepted. There is no way for the box to reject once it has

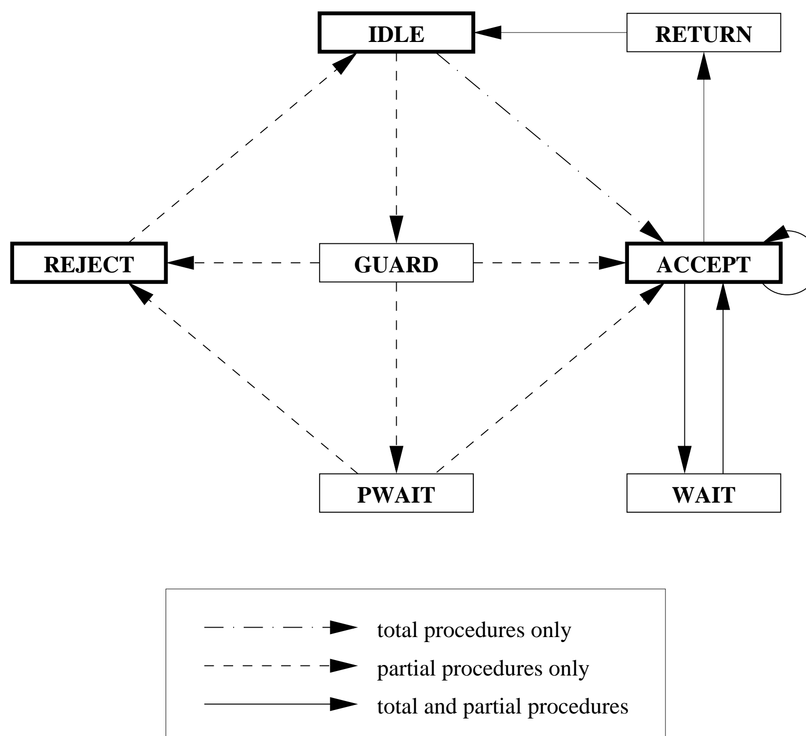

Figure 2.10: Transition diagram for phases of a *TCB* box.

entered phase ACCEPT. Similarly, once a box that enters phase REJECT, the decision has been made to reject, and this cannot be revoked. Note that every nonempty path from IDLE back to itself passes through either phase ACCEPT or REJECT. So we see that the first step during the execution of a procedure call which leaves a box in phase ACCEPT or REJECT (and there is always such a step) determines the overall outcome of the call from the point of view of acceptance or rejection.

Again from the diagram, we can see that there is a possibility for a box to execute an unbounded number of steps a procedure, yet never return to the idle state. There is a self-loop on ACCEPT, and a loop containing ACCEPT and WAIT, so a path in the diagram through either of these phases can be extended without limit. We want to avoid such executions, and ensure that every box returns to phase IDLE at some point after starting to execute a procedure, or, in other words, that each procedure call terminates. We do not address this issue of termination in the semantic definition. In Chapter 4, we develop conditions that ensure that every procedure call terminates.

The type *BoxConfig-rdv* is used to record the dynamic information for a *TCB* box in the rendezvous semantics. Apart from component ϕ, the entries in this record are the same as for *BoxConfig-tot*.

Definition 2.21

$$
\begin{array}{lcl}
\mathit{BoxConfig\text{-}rdv} & \triangleq & \textbf{record} \\
 & & \quad \phi : \mathit{BoxPhase} \\
 & & \quad \sigma : \mathit{State} \\
 & & \quad \gamma : \mathit{CallInfo}_{\perp} \\
 & & \quad \theta : \mathit{Statement}^{*} \\
 & & \textbf{end}
\end{array}
$$

As with *BoxConfig-tot*, we omit the final component of a *BoxConfig-rdv* value when the phase is such that it is necessarily $\bot$. In the current case, if

$$\phi \quad \in \quad \{\text{IDLE}, \text{GUARD}, \text{RETURN}, \text{REJECT}\}$$

then we let

$$(\phi, \sigma, \gamma) \quad = \quad (\phi, \sigma, \gamma, \bot)$$

2.10.4 Program configuration for *TCB*

We define a program configuration for a *TCB* program under the rendezvous semantics as a *BoxConfig-rdv* for each box.

Definition 2.22

$$\textit{ProgConfig-rdv} \quad \triangleq \quad \mathbf{B} \rightarrow \textit{BoxConfig-rdv}$$

2.10.5 Rendezvous semantic rules for *TCB*

All six semantic rules for *TCBtot* carry over as rules for the rendezvous semantics of *TCB*. When we carry the rules over, we apply them to program configurations from *ProgConfig-rdv* rather than from *ProgConfig-tot*. The *TCBtot* rules are the only ones needed to execute total procedure calls. We add to these seven rules for executing partial procedure calls.

The additional rules deal with the issues of guard evaluation, and rejection, since these are particular to partial procedures. The evaluation of total procedures, and the evaluation of the bodies of partial procedures during any accepting call, are handled by the *TCBtot* rules.

The first couple of added rules deal with starting a partial action, and ending a partial action that has rejected. These are shown in Figure 2.11. The rules are

$$\textbf{(p-action-start-rdv)}\quad \frac{\mathbf{C}.D = (\text{IDLE}, \sigma, \bot) \quad a \in \mathit{PartActs}(D) \quad \gamma' = (a)}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\text{GUARD}, \sigma, \gamma')]}$$

$$\textbf{(action-reject-rdv)}\quad \frac{\mathbf{C}.D = (\text{REJECT}, \sigma, \gamma) \quad \gamma = (a)}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\text{IDLE}, \sigma, \bot)]}$$

Figure 2.11: Rendezvous semantics for *TCB*: starting and rejecting a partial action call

explained below.

(p-action-start-rdv) A box in phase IDLE, with an empty call information, can start a call of any of its partial actions. It sets it call information for the action, and enters phase GUARD. Unlike rule **(t-action-start-rdv)**, it does not set the code, since this is determined during guard evaluation.

(action-reject-rdv) This rule is exactly the same as rule **(action-end-rdv)**, except that, before the step, the box is in phase REJECT.

We define separate rules for ending an action that accepts and ending an action that reject, even though they are implemented similarly, because we use the name of the rule for a transition as a label for that transition. For some of our later purposes, having a simple way to distinguish these steps is useful.

The remainder of the added rules concern guard evaluation. These rules describe the transitions on the left-hand side of the diagram in Figure 2.10. Three rules are shown in Figure 2.12.

(guard-accept-rdv) If the value of the alternative function is a pair with no test, the current call accepts, and the box sets its code to the code component of

(guard-accept)

$$\frac{\begin{array}{r@{\;}l}\mathbf{C}.D &= (\textsc{guard}, \sigma, \gamma)\\ p &= Proc(\gamma)\\ Alt(D.p, \sigma) &= (\bot, \theta')\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma, \gamma, \theta')]}$$

(guard-reject)

$$\frac{\begin{array}{r@{\;}l}\mathbf{C}.D &= (\textsc{guard}, \sigma, \gamma)\\ p &= Proc(\gamma)\\ Alt(D.p, \sigma) &= \bot\\ \sigma' &= \sigma \restriction BoxVars(D)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{reject}, \sigma', \gamma)]}$$

(guard-test-rdv)

$$\frac{\begin{array}{r@{\;}l}\mathbf{C}.D &= (\textsc{guard}, \sigma_0, \gamma_0)\\ p &= Proc(\gamma)\\ Alt(D.p, \sigma_0) &= (E.m(\tilde{e}; \tilde{x}), \theta)\\ \theta'_0 &= E.m(\tilde{e}; \tilde{x}); \hat{\theta}\\ \mathbf{C}.E &= (\textsc{idle}, \sigma_1, \bot)\\ \tilde{v} &= [\![\tilde{e}]\!]\sigma_0\\ \tilde{y} &= InParam(E.m)\\ \tilde{z} &= OutParam(E.m)\\ \sigma'_1 &= \sigma_1\,[\tilde{z} \mapsto \tilde{\bot}]\,[\tilde{y} \mapsto \tilde{v}]\\ \gamma'_1 &= (m, D)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix}D \mapsto (\textsc{pwait}, \sigma_0, \gamma_0, \theta'_0)\\ E \mapsto (\textsc{guard}, \sigma'_1, \gamma'_1)\end{bmatrix}}$$

Figure 2.12: Rendezvous semantics for *TCB*: guard evaluation

the pair, and enters phase ACCEPT.

(guard-reject-rdv) If the value of the alternative function is $\perp$, the current call rejects, and the box restricts its state to the box variables, and enters phase REJECT.

(guard-test-rdv) If the value of the alternative function is a pair containing a test call, and the target box of the call is in phase IDLE and its call information is empty, the call can be initiated, in a way similar to rule **(total-call-rdv)**. The source box sets its code to the code component of the pair, with the test call prepended (since the output parameters from the call are required for the return from the procedure), and enters phase PWAIT; the agent box adds a call information, extends its state with the parameters from the call and enters phase GUARD.

The remaining two rules deal with the return from the test call. These rules are shown in Figure 2.13.

(test-accept-rdv) If a box is in phase PWAIT, and the agent is in phase RETURN, the procedure at the source accepts. The returning of output parameter values is handled as in rule **(total-return-rdv)**.

(test-reject-rdv) If a box is in phase PWAIT, and the agent is in phase REJECT, the procedure at the source rejects. The source cleans up its state, clears its code, and enters phase REJECT, and the agent clears its call information, and enters phase IDLE.

2.11 Queue semantics for *TCB*

As noted in the introduction, the rendezvous semantics that we have presented for *TCBtot* and full *TCB* fail to deal adequately with contention between threads for

(test-accept-rdv)

$$\frac{\begin{array}{r}\mathbf{C}.D = (\textsc{pwait}, \sigma_0, \gamma_0, \theta_0)\\ \theta_0 = E.m(\tilde{e};\tilde{x});\theta_0'\\ \mathbf{C}.E = (\textsc{return}, \sigma_1, \gamma_1)\\ \gamma_1 = (m, D, \tilde{v})\\ \sigma_0' = \sigma_0\,[\tilde{x} \mapsto \tilde{v}]\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\textsc{accept}, \sigma_0', \gamma_0, \theta_0')\\ E \mapsto (\textsc{idle}, \sigma_1, \bot)\end{bmatrix}}$$

(test-reject-rdv)

$$\frac{\begin{array}{r}\mathbf{C}.D = (\textsc{pwait}, \sigma_0, \gamma_0, \theta_0)\\ \theta_0 = E.m(\tilde{e};\tilde{x});\hat{\theta}\\ \mathbf{C}.E = (\textsc{reject}, \sigma_1, \gamma_1)\\ \gamma_1 = (m, D)\\ \sigma_0' = \sigma_0 \upharpoonright \mathit{BoxVars}(D)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\textsc{reject}, \sigma_0', \gamma_0)\\ E \mapsto (\textsc{idle}, \sigma_1, \bot)\end{bmatrix}}$$

Figure 2.13: Rendezvous semantics for *TCB*: test return

access to a box. The procedure call rules **(action-start-rdv)**, **(total-call-rdv)**, and **(guard-test)**, all require that the call information for the agent is empty before the step, and full after the step. We show in Chapter 3 that all the rules, except these three, have the property that from a configuration in which a step is enabled according to the rule, no step by a box other than D can reach a configuration where this step is not enabled. This is an important stability property, since it allows us to reason successfully about the termination of procedure calls. Certainly this stability seems plausible. To decide if rule **(local-step)**, for example, can be applied to take a step for box D, we need only look at the configuration of box D. A step for any other box does not change D's configuration, so the local step for D remains enabled until it is taken.

For the three procedure call rules, we do not have this stability property. That is, from a configuration where a procedure call from box D to box E is enabled, a step involving E and another box can reach a configuration where the procedure call step for D is no longer enabled.

We now define a queue semantics that addresses this shortcoming. In the next chapter, we show that every rule in the queue semantics has the above stability property.

We use the same source language as for the last semantics, and we require no extra static information, so we use the function $\mathbf{I}$ for the static program information, as described in Section 2.10.1. We define a program configuration for the new semantics.

2.11.1 Call queues

For the queue semantics, we replace the single *CallInfo* component of the box configuration with a *call queue*, which we represent by an element of $\mathit{CallInfo}^*$. A procedure call on a box is started by appending a *CallInfo* to the end of this se-

quence, and a box that is idle can initialize the call at the head of the queue. Thus we split the rendezvous procedure call into two steps. In the first step the source adds an entry to the agent's call queue, and in the second step, which occurs when this entry has reached the front of the queue, and the agent is idle, the agent starts the procedure call. When the the first step is taken, the input parameter values from the source are included with the call information placed in the queue, since, unlike the rendezvous semantics, the agent does not immediately initialize the parameters for the procedure call.

In this new model, a nonidle box is executing the call at the head of its call queue. We redefine $\mathit{Proc}(\gamma)$, and $\mathit{Source}(\gamma)$ for $\gamma \in \mathit{CallInfo}^*$.

Definition 2.23 *For* $\gamma \in \mathit{CallInfo}^*$, *where* $\gamma = (p, Q, \tilde{v}) \triangleright \hat{\gamma}$,

$$\begin{aligned} \mathit{Proc}(\gamma) &\triangleq p \\ \mathit{Source}(\gamma) &\triangleq Q \end{aligned}$$

2.11.2 Box and program configurations for the queue semantics

The type *BoxConfig* is used to represent the configuration of a box in the queue semantics. The only change from the type *BoxConfig-rdv* is the type of component γ, which is a sequence of *CallInfo* entries, rather than a single *CallInfo*.

Definition 2.24

$$\begin{aligned} \mathit{BoxConfig} \triangleq\ & \textbf{record} \\ & \quad \phi : \mathit{BoxPhase} \\ & \quad \sigma : \mathit{State} \\ & \quad \gamma : \mathit{CallInfo}^* \\ & \quad \theta : \mathit{Statement}^* \\ & \textbf{end} \end{aligned}$$

We omit an empty θ for the same phases as with *BoxConfig-rdv*.

A program configuration is, as may be expected, a map mapping each box in the program to a *BoxConfig*.

Definition 2.25

$$ProgConfig \quad \triangleq \quad \mathbf{B} \rightarrow BoxConfig$$

2.11.3 Queue semantic rules for *TCB*

The major change we make to the rendezvous semantics to give the queue semantics is the splitting of procedure calls into two independent steps, one under the control of the source, and the other under the control of the agent. We replace the rendezvous procedure call rules

(p-action-start-rdv)
(t-action-start-rdv)
(total-call-rdv)
(guard-test-rdv)

with seven rules implementing the two-step procedure call. We also replace each of the following rules

(proc-term-rdv)
(action-end-rdv)
(action-reject-rdv)
(test-accept-rdv)
(test-reject-rdv)
(total-return-rdv)

with a rule named without the **-rdv** suffix. The changes for these rules are minor changes needed because of the change from a single call information to a call queue. The remaining rules,

> **(guard-accept)**
> **(guard-reject)**
> **(local-step)**

are used as is. We show only the replacements for the first group of rules here. The full set of queue semantic rules for *TCB* is given in Appendix A.

The first three new rules are shown in Figure 2.14. These rules implement the first step of the two-step procedure call outlined above. In each case, the procedure call is made by placing an entry containing the name of the procedure, the source, and the values for the input parameters (if any) at the back of the agent's call queue, regardless of the agent's configuration. Note that rule **(action-start)** is used for both partial and total actions.

The other four rules, which implement the second of the two steps, are shown in Figure 2.15. There are separate rules for all four combinations of partial/total and action/method. These rules involve only the agent of the call. When a box is idle, and there is an entry in its call queue, a call can be initialized for the first entry. The value list from this entry is used to initialize the input parameters, and these values are then removed from the entry.

For partial procedures, the box enters phase GUARD after the initialization step. For total procedures, the enters box phase ACCEPT after the step, and its code is initialized, as in the rendezvous semantics.

The rules for actions are a special case of those for methods, since actions have no parameters. However, as with **(action-end)** and **(action-reject)**, we use separate rules to give separate labels to the corresponding step in the semantics.

(action-start)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\phi, \sigma, \gamma, \theta)\\ a \in Actions(D)\\ \gamma' = \gamma \triangleleft (a)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\left[D \mapsto (\phi, \sigma, \gamma', \theta)\right]}$$

(total-call)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\textsc{accept}, \sigma_0, \gamma_0, \theta_0)\\ \theta_0 = E.m(\tilde{e}; \tilde{x}); \hat{\theta}\\ \mathbf{C}.E = (\phi_1, \sigma_1, \gamma_1, \theta_1)\\ \tilde{v} = [\![\tilde{e}]\!]\sigma_0\\ \gamma_1' = \gamma_1 \triangleleft (m, D, \tilde{v})\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix}D \mapsto (\textsc{wait}, \sigma_0, \gamma_0, \theta_0)\\ E \mapsto (\phi_1, \sigma_1, \gamma_1', \theta_1)\end{bmatrix}}$$

(guard-test)

$$\frac{\begin{array}{r l}\mathbf{C}.D &= (\textsc{guard}, \sigma_0, \gamma_0)\\ p &= Proc(\gamma)\\ Alt(D.p, \sigma_0) &= (E.m(\tilde{e}; \tilde{x}), \theta)\\ \theta_0' &= E.m(\tilde{e}; \tilde{x}); \hat{\theta}\\ \mathbf{C}.E &= (\phi_1, \sigma_1, \gamma_1, \theta_1)\\ \tilde{v} &= [\![\tilde{e}]\!]\sigma_0\\ \gamma_1' &= \gamma_1 \triangleleft (m, D, \tilde{v})\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix}D \mapsto (\textsc{pwait}, \sigma_0, \gamma_0, \theta_0')\\ E \mapsto (\phi_1, \sigma_1, \gamma_1', \theta_1)\end{bmatrix}}$$

Figure 2.14: Queue semantic rules for *TCB*: procedure call

(p-action-init)

$$\frac{\mathbf{C}.D = (\textsc{idle}, \sigma, \gamma) \quad p = \mathit{Proc}(\gamma) \quad p \in \mathit{PartActs}(D)}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{guard}, \sigma, \gamma)]}$$

(p-method-init)

$$\frac{\mathbf{C}.D = (\textsc{idle}, \sigma, \gamma) \quad \gamma = (p, Q, \tilde{v}) \triangleright \hat{\gamma} \quad p \in \mathit{PartMeths}(D) \quad \tilde{y} = \mathit{InParam}(D.p) \quad \tilde{z} = \mathit{OutParam}(D.p) \quad \sigma' = \sigma\,[\tilde{z} \mapsto \tilde{\bot}]\,[\tilde{y} \mapsto \tilde{v}] \quad \gamma' = (p, Q) \triangleright \hat{\gamma}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{guard}, \sigma', \gamma')]}$$

(t-action-init)

$$\frac{\mathbf{C}.D = (\textsc{idle}, \sigma, \gamma) \quad p = \mathit{Proc}(\gamma) \quad p \in \mathit{TotActs}(D) \quad \theta' = \mathit{Code}(D.p)}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma, \gamma, \theta')]}$$

(t-method-init)

$$\frac{\mathbf{C}.D = (\textsc{idle}, \sigma, \gamma) \quad \gamma = (p, Q, \tilde{v}) \triangleright \hat{\gamma} \quad p \in \mathit{TotMeths}(D) \quad \tilde{y} = \mathit{InParam}(D.p) \quad \tilde{z} = \mathit{OutParam}(D.p) \quad \sigma' = \sigma\,[\tilde{z} \mapsto \tilde{\bot}]\,[\tilde{y} \mapsto \tilde{v}] \quad \gamma' = (p, Q) \triangleright \hat{\gamma} \quad \theta' = \mathit{Code}(D.p)}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma', \gamma', \theta')]}$$

Figure 2.15: Queue semantic rules for *TCB*: procedure initialization

2.12 Summary

We have given two semantic definitions for *TCB*, a rendezvous semantics and a queue semantics. We regard the queue semantics as the true description of the execution of *TCB* programs. In the next chapter, we explore some of the properties of program configurations and executions for this semantics.

In the remaining chapters, we explore the properties of the queue semantics for *TCB*. In Chapter 5, we use the execution generated by the rendezvous semantics as an intermediate form as we rearrange the steps of a concurrent execution to give a sequential execution.

2.12.1 Run-time errors

As noted in Section 2.5, we avoid the issue of run-time errors with Assumption 2.3. If we choose not to make this assumption, we can extend the semantics of *TCB* to handle run-time errors as follows.

We define a new phase FAIL, and, for each rule that involves evaluation of an expression, either directly, as in rules **(local-step)** and **(total-call)**, or indirectly, as in rule **(guard-accept)**, we add a rule that defines a transition to FAIL for a box where evaluation of an expression causes a run-time error. We define no rules that take a step for a box in phase FAIL. This means that a run-time error puts a box in a failed configuration, and it remains in this configuration for the remainder of the execution.

Chapter 3

Execution of *TCB* programs

3.1 Introduction

This chapter discusses the elements, and some of the consequences, of the queue semantics for *TCB* defined in Chapter 2.

We define subsets of *BoxConfig* and *ProgConfig* containing the *well-formed* elements of these types. These subsets contain only the elements that we expect to arise during an execution. They exclude, for example, box configurations in which the phase is ACCEPT, but the call queue is empty, and program configurations in which box D is executing a method call for box E, but box E is in phase IDLE.

We next consider the transition relation defined by the semantic rules. We define a way to assign *labels* to steps between configurations allowed by the rules. A step is *enabled* in a configuration if there is any step allowed by the rules starting from that configuration; a step is enabled for box D if the step is enabled, and taking it changes D's configuration. We show that, in any configuration, all steps enabled for D have the same label.

We show that a step from a well-formed configuration cannot reach a configuration that is not well-formed. By induction, we have that any sequence of steps,

starting from a well-formed configuration, passes through only well-formed configurations. Thus we can confine our attention to well-formed program configurations for the remainder of this work.

Executions are paths made up of labelled steps between program configurations, where each step is allowed by some semantic rule. We give a definition for constructing a set of executions from a set of configurations and a set of labels, and we use this to define the executions for a *TCB* program using the well-formed program configurations and the step labels.

3.2 Box configurations

The type *BoxConfig* is used for represent the dynamic information associated with a box (its *configuration*) during program execution under the queue semantics. First we define some mild abuse of notation for components of the box configuration.

Definition 3.1 *For* $\gamma \in \mathit{CallInfo}^*$,

$$(p, Q) \in \gamma \quad \triangleq \quad \langle \exists\, \tilde{v} \;\; :: \;\; (p, Q, \tilde{v}) \in \gamma \;\rangle$$

Definition 3.2 *For* $S \in \mathit{Statement}$,

$$S = E.m(-) \quad \triangleq \quad \langle \exists\, \tilde{x}, \tilde{v} \;\; :: \;\; S = E.m(\tilde{x}; \tilde{v}) \;\rangle$$

We define the following terminology for box configurations.

Definition 3.3 *For* $bc \in \mathit{BoxConfig}$,

$$qt(bc) \quad \triangleq \quad bc.\gamma = \bot$$

If $qt(bc)$, *then we say that* bc *is* quiescent.

Definition 3.4 *For $D \in \mathbf{B}$, and $bc \in BoxConfig$, where bc is the configuration associated with D, a call of procedure p is* active *in D if*

$$\langle \exists\ Q \ :\ Q \in \mathbf{B}_\perp \ :\ (p, Q) \in bc.\gamma \ \rangle$$

If $(p, Q, \tilde{v}) \in bc.\gamma$, Q is the source, *and D is the* agent *for this call.*

As noted in Chapter 2 not all possible combinations of component values for a box configuration make sense in terms of the intended interpretation. We define the set of *well-formed* configurations.

Definition 3.5 *For $bc \in BoxConfig$, bc is a* well-formed *configuration for D if all the following hold.*

1. $bc.\phi \neq \text{IDLE} \ \Rightarrow \ bc.\gamma \neq \perp$
2. $bc.\phi \in \{\text{IDLE}, \text{GUARD}, \text{RETURN}, \text{REJECT}\} \ \Rightarrow \ bc.\theta = \perp$
3. $bc.\phi \in \{\text{WAIT}, \text{PWAIT}\} \ \Rightarrow \ \langle \exists\ E, m \ :: \ \text{first}(bc.\theta) = E.m(-) \ \rangle$
4. $bc.\phi \in \{\text{GUARD}, \text{PWAIT}, \text{REJECT}\} \ \Rightarrow \ Proc(bc.\gamma) \in Partials(D)$
5. $bc.\phi \in \{\text{IDLE}, \text{RETURN}, \text{REJECT}\} \ \Rightarrow \ \text{dom}(bc.\sigma) = BoxVars(D)$
6. $\text{dom}(bc.\sigma) \ \supseteq \ BoxVars(D)$
7. $\langle \forall\ (p, Q)$
   $:\ (p, Q) \in bc.\gamma$
   $:\ p \in Procs(D) \ \wedge \ Q \neq D \ \wedge \ (p \in Actions(D) \ \equiv \ Q = \perp)$
   $\rangle$
8. $\langle \forall\ E \ :\ E \in \mathbf{B} \ :\ \langle \#\ (m, Q, \tilde{v}) \ :\ (m, Q, \tilde{v}) \in bc.\gamma \ :\ Q = E \ \rangle \ \leq \ 1 \ \rangle$

9. $\langle \forall \; (p, Q, \tilde{v})$

$: (bc.\phi = \text{IDLE} \;\wedge\; (p, Q, \tilde{v}) \in bc.\gamma) \;\vee$

$(bc.\phi \neq \text{IDLE} \;\wedge\; (p, Q, \tilde{v}) \in \text{rest}(bc.\gamma))$

$: \tilde{v} \textit{ matches } \mathit{InParam}(D.p)$

$\rangle$

10. $bc.\phi = \text{RETURN} \wedge \text{first}(bc.\gamma) = (p, Q, \tilde{v}) \;\Rightarrow\; \tilde{v} \textit{ matches } \mathit{OutParam}(D.p)$

For $D \in \mathbf{B}$,

$$BC(D) \;\triangleq\; \{\, bc \mid bc \textit{ is a well-formed configuration for } D \,\}$$

Condition 1 ensures that a box that is executing a call has an entry in the call queue that identifies the call. Note that this condition means that if $bc \in BC(D)$ and $qt(bc)$, then $bc.\phi = \text{IDLE}$. Condition 2 justifies the abbreviated form used for box configurations for the given phases. Condition 3 ensures that the any waiting box has the procedure call for which it is waiting at the front of its code. Condition 4 ensures that the phases associated with guard evaluation and procedure rejection are only reached while executing a partial procedure. Condition 5 ensures that the state of each box between procedure calls gives values only to the box variables. Condition 6 ensures that the state gives values to the box variables, at least. Condition 7 ensures that a box never calls a method on itself, that the procedure names in the call queue are valid, and that there is a box as source in each method call, and none in each action call. Condition 8 says that at any time, a box has at most one active method call for any particular source. Condition 9 ensures that all calls that have not yet started have the right types in their input parameter lists. Condition 10 ensures that a box that is ready to return to its source has the correct types in its output parameter list.

For *TCB*, the state of each box satisfies stronger conditions than Conditions 5 and 6 in Definition 3.5. During the evaluation of a guard, and the execution of the

body of a procedure, a box's state is extended by the input and output parameters of the procedure in the current call. This stronger condition is an artifact of the simple definition of *TCBloc*, and the conditions given allow for a local language that extends the local state in more complex ways during procedure execution.

For a well-formed box configuration that is not in phase IDLE, we call the front entry in the call queue the *current call.* This is the call that the box is currently executing. If the phase is IDLE, the current call is not defined.

We define a partial order on box configurations using to the prefix order on the call queue. A pair of configurations in this order have the same value for the other components.

Definition 3.6 *For $D \in \mathbf{B}$, and $bc, bc' \in BC(D)$,*

$$bc \sqsubseteq bc' \quad \triangleq \quad bc.\phi = bc'.\phi \;\wedge\; bc.\gamma \sqsubseteq bc'.\gamma \;\wedge\; bc.\sigma = bc'.\sigma \;\wedge\; bc.\theta = bc'.\theta$$

3.3 Program Configurations

A program configuration maps each box name to its associated box configuration. We extend the terminology for box configurations to program configurations in the natural way.

Definition 3.7 *For $\mathbf{C} \in ProgConfig$,*

$$qt(\mathbf{C}) \quad \triangleq \quad \langle \forall\, D \;:\; D \in \mathbf{B} \;:\; qt(\mathbf{C}.D)\ \rangle$$

If $qt(\mathbf{C})$, we say $\mathbf{C}$ is quiescent*.*

Definition 3.8 *A call of procedure $D.p$ is* active *in $\mathbf{C}$ if an call of p is active in D.*

$$Actives(\mathbf{C}) \quad \triangleq \quad \{\, D.a \mid a \in Actions(D) \;\wedge\; D.a \textit{ is active in } \mathbf{C} \,\}$$

The set $Actives(\mathbf{C})$ contains all actions that have active threads in $\mathbf{C}$.

As with box configurations, we define *well-formed* program configurations. The first requirement is that each box has a well-formed box configuration.

Definition 3.9 *For* $\mathbf{C} \in ProgConfig$,

$$\mathbf{C} \textit{ is } \text{locally well-formed} \;\triangleq\; \langle \forall\, D \;:\; D \in \mathbf{B} \;:\; \mathbf{C}.D \in BC(D)\; \rangle$$

For a program configuration to be well-formed, we also need to ensure that the configuration respects the mutual dependencies between the values of the configurations for different boxes. For example, if a box is executing a method call for a source, then we expect the source to be waiting for the results of the call. Conversely, if a box is waiting for a method call to return, we expect the method call to be in the call queue for the agent. We define conditions on a program configuration to capture this restriction. We use the following function, which returns the appropriate phase for the source of a method call.

Definition 3.10 *For* $D \in \mathbf{B}$, *and* $m \in Methods(D)$,

$$\begin{array}{rcll} WaitPhase(D.m) & \triangleq & \text{PWAIT} & \text{if } \; m \in PartMeths(D) \\ & & \text{WAIT} & \text{if } \; m \in TotMeths(D) \end{array}$$

Definition 3.11 *For* $\mathbf{C} \in ProgConfig$,

$$\begin{array}{l} \mathbf{C} \textit{ is } \text{call correct} \;\triangleq \\ \quad \langle \forall\; D, E, m \\ \quad\;\; : D, E \in \mathbf{B} \;\wedge\; m \in Methods(E) \\ \quad\;\; : (m, D) \in \mathbf{C}.E.\gamma \;\equiv \\ \quad\;\;\;\; \mathbf{C}.D.\phi = WaitPhase(E.m) \;\wedge\; \text{first}(\mathbf{C}.D.\theta) = E.m(-) \\ \quad\;\; \rangle \end{array}$$

Note that the term of the quantification in Definition 3.11 is an equivalence, so it excludes configurations in which there is a method call with source D in a call queue, but D is not waiting, and the case when D is waiting for E but there is no method call with source D in E's call queue.

The following theorem is a direct consequence of Definitions 3.9 and 3.11.

Theorem 3.12 *If* $\mathbf{C}$ *is locally well-formed and call correct program configuration, and if* D *is waiting in* $\mathbf{C}$*, then there is exactly one call queue entry with source* D *in* $\mathbf{C}$.

Proof

Suppose $\mathbf{C}$ is a locally well-formed and call correct program configuration, and that $\mathbf{C}.D.\phi = \textsc{pwait}$.

$$\begin{array}{ll}
 & \mathbf{C}.D.\phi = \textsc{pwait} \\
\equiv & \qquad \{\ \mathbf{C} \text{ is locally well-formed, so } \mathbf{C}.D \in BC(D)\ \} \\
 & \mathbf{C}.D.\phi = \textsc{pwait} \\
 & \wedge\ \langle \exists\ E, m\ \ :\ \ E \in \mathbf{B}\ \wedge\ m \in \mathit{PartMeths}(E)\ \ :\ \ \text{first}(\mathbf{C}.D.\theta) = E.m(-)\ \rangle \\
\equiv & \qquad \{\ \text{predicate calculus}\ \} \\
 & \langle \exists\ E, m \\
 & \quad :\ m \in \mathit{PartMeths}(E) \\
 & \quad :\ \mathbf{C}.D.\phi = \textsc{pwait} \wedge \text{first}(\mathbf{C}.D.\theta) = E.m(-) \\
 & \quad \rangle \\
\equiv & \qquad \{\ \mathbf{C} \text{ is call correct}\ \} \\
 & \langle \exists\ E, m\ \ :\ \ m \in \mathit{PartMeths}(E)\ \ :\ \ (m, D) \in \mathbf{C}.E.\gamma\ \rangle
\end{array}$$

Thus, we have that there is at least one box that has a call queue entry with source D, when $\mathbf{C}.D.\phi = \textsc{pwait}$. A similar proof shows the same result when $\mathbf{C}.D.\phi = \textsc{wait}$.

We have

$$(m, D) \in \mathbf{C}.E.\gamma\ \wedge\ (m', D) \in \mathbf{C}.E'.\gamma$$

$$\Rightarrow \qquad \{ \ \mathbf{C} \text{ is call correct } \}$$

$$\text{first}(\mathbf{C}.D.\theta) = E.m(-) \ \wedge \ \text{first}(\mathbf{C}.D.\theta) = E'.m'(-)$$

$$\Rightarrow \qquad \{ \text{ transitivity of } =, \text{ equality of procedure calls } \}$$

$$E = E'$$

This shows that, if D is waiting, there is a unique box E with a call queue entry with source D.

Since $\mathbf{C}$ is locally well-formed, $\mathbf{C}.E$ is well-formed, so there is exactly one entry in its call queue with source D. Thus there is exactly one such entry in the whole of $\mathbf{C}$.
(End of proof)

3.3.1 Relations induced by calls

Let $\mathbf{C}$ be a locally well-formed and call correct program configuration, and suppose that, for some $D_0 \in \mathbf{B}$, D_0's current call in $\mathbf{C}$ is for an action, and that D_0 is in a waiting phase. Then, from Theorem 3.12, there is a unique D_1 with a call queue entry for D_0. We say that D_0 is *waiting for* D_1. If the call for D_0 is the current call in D_1, then we say that D_1 is *executing for* D_0. If D_1 is in a waiting phase, then there is a unique D_2 for which D_1 is waiting. Continuing in this way, we construct the longest possible sequence of boxes,

$$\langle D_0, D_1, D_2, \dots, D_{n-1} \rangle$$

where D_i is waiting for D_{i+1}, and D_{i+1} is executing for D_i, for $0 \leq i < n-1$. Each of $D_0, \dots, D_{n-2}$ is in a waiting phase, but D_{n-1} may not be. If D_{n-1} is waiting for some box D_n, D_n is not executing for D_{n-1} (since the sequence is maximal), and so the call for D_{n-1} is not the current call in D_n.

In a well-formed configuration, we expect every box that is not idle to be

executing as part of a sequence like the one above. We define some functions on program configurations so we can express these conditions. The first set of functions express the "is executing for" relation. Since each call has a unique source, we express the relations as partial functions.

Definition 3.13 *For* $\mathbf{C} \in \mathit{ProgConfig}$, *and* $D \in \mathbf{B}$,

$$\begin{array}{rcll} \mathit{Kp}.\mathbf{C}.D & \triangleq & \mathit{Source}(\mathbf{C}.D.\gamma) & \text{if } \mathbf{C}.D.\phi \neq \text{IDLE} \\ & & & \quad \wedge\ \mathit{Proc}(\mathbf{C}.D.\gamma) \in \mathit{Partials}(D) \\ \mathit{Kt}.\mathbf{C}.D & \triangleq & \mathit{Source}(\mathbf{C}.D.\gamma) & \text{if } \mathbf{C}.D.\phi \neq \text{IDLE} \\ & & & \quad \wedge\ \mathit{Proc}(\mathbf{C}.D.\gamma) \in \mathit{Totals}(D) \\ K.\mathbf{C}.D & \triangleq & \mathit{Kp}.\mathbf{C}.D & \text{if } D \in \text{dom}(\mathit{Kp}.\mathbf{C}) \\ & & \mathit{Kt}.\mathbf{C}.D & \text{if } D \in \text{dom}(\mathit{Kt}.\mathbf{C}) \end{array}$$

We have

$$\mathit{Kp}, \mathit{Kt}, K : \mathit{ProgConfig} \to \mathbf{B} \hookrightarrow \mathbf{B}_\perp$$

Thus $\mathit{Kp}.\mathbf{C}$ is a partial function from $\mathbf{B}$ to $\mathbf{B}_\perp$. If $\mathit{Kp}.\mathbf{C}.D = E$, then D is executing a partial method for source E, and if $\mathit{Kp}.\mathbf{C}.D = \perp$, then D is executing a partial action, Functions $\mathit{Kt}.\mathbf{C}$ and $K.\mathbf{C}$ have similar interpretations, the first for total procedure calls, and the second for procedure calls of either kind. If $K.\mathbf{C}$ is not defined at D, then D is not currently executing a procedure in $\mathbf{C}$.

The following theorems state some useful properties of these functions. The proofs are given in Appendix B. The first theorem states that there is at most one box executing for any given box. That is, in any well-formed configuration $\mathbf{C}$, there cannot be two boxes mapped to the same box by $K.\mathbf{C}$. The function $K.\mathbf{C}$ is one-to-one, except that multiple boxes can be mapped to $\perp$ by $K.\mathbf{C}$.

Theorem 3.14

$$\begin{array}{l} \langle\forall\ \mathbf{C}, D, E \\ \quad : \mathbf{C} \in \mathit{ProgConfig} \ \wedge\ \mathbf{C} \textit{ is locally well-formed and call correct} \ \wedge \\ \quad\quad D, E \in \mathrm{dom}(K.\mathbf{C}) \ \wedge\ K.\mathbf{C}.D = K.\mathbf{C}.E \\ \quad : K.\mathbf{C}.D = \bot \ \vee\ D = E \\ \quad \rangle \end{array}$$

The next theorem states that if box D is executing for box E, then E is executing a procedure.

Theorem 3.15

$$\begin{array}{l} \langle\forall\ \mathbf{C}, D \\ \quad : \mathbf{C} \in \mathit{ProgConfig} \ \wedge\ \mathbf{C} \textit{ is locally well-formed and call correct} \ \wedge \\ \quad\quad D \in \mathrm{dom}(K.\mathbf{C}) \\ \quad : K.\mathbf{C}.D = \bot \ \vee\ K.\mathbf{C}.D \in \mathrm{dom}(K.\mathbf{C}) \\ \quad \rangle \end{array}$$

The final theorem says that the source of a partial method call is never executing a total procedure, which is precisely the restriction we impose.

Theorem 3.16

$$\begin{array}{l} \langle\forall\ \mathbf{C} \\ \quad : \mathbf{C} \in \mathit{ProgConfig} \ \wedge\ \mathbf{C} \textit{ is locally well-formed and call correct} \\ \quad : \mathrm{rng}(Kp.\mathbf{C}) \ \ \mathrm{disj} \ \ \mathrm{dom}(Kt.\mathbf{C}) \\ \quad \rangle \end{array}$$

To follow the chain of calls given by $K.\mathbf{C}$, we apply the function repeatedly to an argument. We define iteration for a partial function.

Definition 3.17 *Let $f : S \hookrightarrow S$, be a partial function on some set S. For $x \in S$,*

$$f^0.x \triangleq x$$

and, for any $n \geq 0$

$$f^{n+1}.x \triangleq f(f^n.x) \quad \text{if} \quad f^n.x \textit{ is defined} \wedge f^n.x \in \text{dom}(f)$$

The following is a useful property of this iteration.

Theorem 3.18

$$f^{n+1}.x = f^n(f.x)$$

Proof
A simple induction, using the associativity of function composition.
(End of proof)

So, for any $D \in \text{dom}(K.\mathbf{C})$, $(K.\mathbf{C})^0.D$ is D, and $(K.\mathbf{C})^1.D$ is the box for which D is executing. If $(K.\mathbf{C})^1.D \in \text{dom}(K.\mathbf{C})$, then $(K.\mathbf{C})^2.D$ is the box for which $(K.\mathbf{C})^1.D$ is executing, and so on. For any $D \in \text{dom}(K.\mathbf{C})$, we can construct the sequence of boxes

$$\begin{array}{c}(K.\mathbf{C})^0.D \\ (K.\mathbf{C})^1.D \\ (K.\mathbf{C})^2.D \\ \vdots\end{array}$$

If $(K.\mathbf{C})^k.D = \perp$, for some $k > 0$, then the sequence is finite. In this case, D is (transitively) executing for a box which is executing an action call. This what we expect to happen during program execution. Otherwise, if no such k exists, then by

Theorem 3.15, the sequence is infinite. In this case, since $\mathbf{B}$ is finite, the sequence contains repeated elements. We can use Theorem 3.14 to show that in this case there is a k such that $(K.\mathbf{C})^k.D = D$. So there is a cycle of boxes, where each box in the cycle is executing a method for the next, so D is transitively executing for itself. We define a condition to exclude such cases from consideration.

Definition 3.19 *For* $\mathbf{C} \in \mathit{ProgConfig}$,

$$
\begin{array}{l}
\mathbf{C} \textit{ is } \text{well-founded} \;\triangleq \\
\quad \langle \forall\, D \;:\; D \in \mathrm{dom}(K.\mathbf{C}) \;:\; \langle \exists\, k \;:\; k > 0 \;:\; (K.\mathbf{C})^k.D = \bot \;\rangle\;\rangle
\end{array}
$$

3.3.2 Well-formed program configurations

We combine all the conditions on program configurations defined so far to give the definition of a well-formed program configuration.

Definition 3.20 *For* $\mathbf{C} \in \mathit{ProgConfig}$,

$$
\begin{array}{rcl}
\mathbf{C} \textit{ is } \text{well-formed} & \triangleq & \mathbf{C} \textit{ is locally well-formed, call correct, and well-founded} \\
\mathit{PC} & \triangleq & \{\, \mathbf{C} \mid \mathbf{C} \in \mathit{ProgConfig} \wedge \mathbf{C} \textit{ is well-formed} \,\}
\end{array}
$$

3.3.3 Call stacks

We define a *call stack* for a well-formed program configuration as a maximal sequence generated by $K.\mathbf{C}$.

Definition 3.21 *For* $\mathbf{C} \in \mathit{PC}$*, the sequence* $\Omega \in \mathbf{B}^+$ *is a* call stack *if*

$$
\begin{array}{l}
\langle \forall\, i \;:\; 0 \le i < |\Omega| - 1 \;:\; K.\mathbf{C}.(\Omega[i]) = \Omega[i+1] \;\rangle \;\wedge \\
\mathrm{first}(\Omega) \notin \mathrm{rng}(K.\mathbf{C}) \;\wedge\; K.\mathbf{C}.(\mathrm{last}(\Omega)) = \bot
\end{array}
$$

Definition 3.22 *For* $\mathbf{C} \in PC$,

$$Stacks(\mathbf{C}) \triangleq \{\, \Omega \mid \Omega \textit{ is a call stack for } \mathbf{C} \,\}$$

From Theorem 3.16, a call stack Ω can be divided into two parts, a first part in which all boxes are executing a total procedure, and a second part in which all boxes are executing a partial procedure.

The following theorem follows from the definitions and theorems above.

Theorem 3.23

$$\begin{array}{l} \langle \forall\ \mathbf{C}, D \\ \quad : \mathbf{C} \in PC \ \wedge \ D \in \mathbf{B} \ \wedge \ \mathbf{C}.D.\phi \neq \text{IDLE} \\ \quad : \langle \exists!\, \Omega \ : \ \Omega \in Stacks(\mathbf{C}) \ : \ D \in \Omega \ \rangle \\ \quad \rangle \end{array}$$

Theorem 3.23 shows that the set of active boxes can be partitioned among the call stacks. The theorem justifies the following definition.

Definition 3.24 *For* $\mathbf{C} \in PC$, *and* $D \in \mathbf{B}$, *where* D *is not idle in* $\mathbf{C}$,

$$\begin{aligned} CallStack(\mathbf{C}, D) &\triangleq \Omega \quad \textit{where } \Omega \in Stacks(\mathbf{C}) \wedge D \in \Omega \\ RootBox(\mathbf{C}, D) &\triangleq \text{last}(CallStack(\mathbf{C}, D)) \\ Root(\mathbf{C}, D) &\triangleq E.a \quad \textit{where } E = RootBox(\mathbf{C}, D) \ \wedge \ a = Proc(\mathbf{C}.E.\gamma) \end{aligned}$$

We call $Root(\mathbf{C}, D)$ *the* root action *for* D *in* $\mathbf{C}$.

Action $Root(\mathbf{C}, D)$ is the action whose execution is the ultimate cause of box D's current execution in $\mathbf{C}$.

3.3.4 Wait lines

We define some partial functions for the "is waiting for" relation between boxes. These functions are almost the inverse of the call stack function, and it has similar properties. However, the differences are significant. First, we define a function that returns the box from a method call statement at the front of a sequence of statements.

Definition 3.25 *For $\theta \in Statement^*$,*

$$Agent(\theta) \;\triangleq\; E \quad \text{if } \langle \exists\, m \;::\; \text{first}(\theta) = E.m(-) \;\rangle$$

The following theorem states that in a well-formed configuration, every box that is in a waiting phase has an agent, by the above function. This is an immediate consequence of the definitions.

Theorem 3.26

$$\begin{array}{l} \langle \forall\; \mathbf{C}, D \\ \;\; : \mathbf{C} \in PC \;\;\wedge\;\; D \in \mathbf{B} \\ \;\; : \mathbf{C}.D.\phi \in \{\textsc{pwait}, \textsc{wait}\} \;\;\Rightarrow\;\; \mathbf{C}.D.\theta \in \text{dom}(Agent) \\ \;\; \rangle \end{array}$$

Now we define the functions for the "is waiting for" relation.

Definition 3.27 *For $\mathbf{C} \in PC$, and $D \in \mathbf{B}$,*

$$\begin{array}{rcll} Wp.\mathbf{C}.D & \triangleq & Agent(\mathbf{C}.D.\theta) & \text{if } \;\mathbf{C}.D.\phi = \textsc{pwait} \\ Wt.\mathbf{C}.D & \triangleq & Agent(\mathbf{C}.D.\theta) & \text{if } \;\mathbf{C}.D.\phi = \textsc{wait} \\ W.\mathbf{C}.D & \triangleq & Wp.\mathbf{C}.D & \text{if } \;D \in \text{dom}(Wp.\mathbf{C}) \\ & & Wt.\mathbf{C}.D & \text{if } \;D \in \text{dom}(Wt.\mathbf{C}) \end{array}$$

The following theorem relates these functions to the "is executing for" function.

Theorem 3.28

$$\langle \forall\ \mathbf{C}, D, E$$
$$:\ \mathbf{C} \in PC\ \ \wedge\ \ D, E \in \mathbf{B}$$
$$:\ (Kp.\mathbf{C}.D = E\ \Rightarrow\ Wp.\mathbf{C}.E = D)\ \ \wedge\ \ (Kt.\mathbf{C}.D = E\ \Rightarrow\ Wt.\mathbf{C}.E = D)$$
$$\rangle$$

Proof

From Definitions 3.13, 3.20 and 3.27.

(End of proof)

From Theorem 3.28, we have, for any $\Omega \in \mathit{Stacks}(\mathbf{C})$,

$$\langle \forall\, i\ :\ 0 < i < |\Omega|\ :\ W.\mathbf{C}.(\Omega[i]) = \Omega[i-1]\ \rangle$$

Thus every box in a call stack, except the first, is waiting for the previous box. The first box may be in a waiting state, in which case there is some D such that $W.\mathbf{C}.(\Omega[0]) = D$. In this case, a call with source $\Omega[0]$ is in D's call queue, but it is not D's current call. If box D is waiting, then $\Omega[0]$ is also waiting for $W.\mathbf{C}.D$. Box D may not be executing for the same thread as box $\Omega[0]$. Unlike $K.\mathbf{C}$, sequences formed by iterating $W.\mathbf{C}$ can include boxes executing for different threads.

As with $K.\mathbf{C}$, we can form a sequence by iterating $W.\mathbf{C}$. For any box D that is waiting in configuration $\mathbf{C}$, we construct a maximal sequence.

$$\langle\ D\ ,\ W.\mathbf{C}.D\ ,\ (W.\mathbf{C})^2.D\ ,\ \ldots\ \rangle$$

Box D is waiting for every box in this sequence. Before D can next take a step, every other box in the sequence must return control to the preceding box. We call

these sequences *wait lines.* Unlike $K.\mathbf{C}$, function $W.\mathbf{C}$ is not well-founded. We can reach a well-formed configuration $\mathbf{C}$ in which the wait line for box D is infinite.

Definition 3.29 *For $D \in \mathbf{B}$, and $\mathbf{C} \in PC$, the* wait line *for D in $\mathbf{C}$ is a sequence $\Omega \in \mathbf{B}^+$ such that*

$$\Omega[0] \;=\; D \;\wedge\; (|\Omega| \;=\; \infty \;\vee\; \text{last}(\Omega) \notin \text{dom}(W.\mathbf{C})) \;\wedge$$
$$\langle \forall\, i \;:\; 0 \leq i < |\Omega| - 1 \;:\; W.\mathbf{C}.(\Omega[i]) = \Omega[i+1] \;\rangle$$

If the wait line for D in $\mathbf{C}$ is finite, then the last box is not waiting, and it is not quiescent, so it can take a conditional step. If the wait line for D in $\mathbf{C}$ is infinite, then it contains repeated elements, since $\mathbf{B}$ is finite. In this case, $W.\mathbf{C}$ is cyclic. The wait line has a suffix consisting of the boxes in the cycle repeated infinitely. Each box in the cycle is in a waiting phase, and is waiting for a box in a waiting phase, so no box can take a step. The boxes are *deadlocked*, as are all boxes whose wait line includes them.

Deadlock is a permanent configuration. If box D is deadlocked at configuration i in execution ε, then, in the execution after $\varepsilon[i]$, D is deadlocked in every configuration, and there are no conditional steps for D. The next chapter discusses deadlock, and defines conditions that ensure deadlock-free executions.

3.3.5 Persistent states

When a box finishes executing a procedure, it restrict the domain of its state to the box variables. Any variables added to the state during execution of the procedure are discarded. The values of the box variables is the box's *persistent state.* This is the state that is carried from one procedure call to the next.

We define the persistent state for a box, and an equivalence on program configurations based on the persistent states.

Definition 3.30 *For* $\mathbf{C}, \mathbf{C}' \in PC$, *and* $D \in \mathbf{B}$,

$$\begin{aligned} \mathit{Persist}(\mathbf{C}, D) &\triangleq (\mathbf{C}.D.\sigma) \restriction \mathit{BoxVars}(D) \\ \mathit{PersistEq}(\mathbf{C}, \mathbf{C}') &\triangleq \langle \forall\, D \;:\; D \in \mathbf{B} \;:\; \mathit{Persist}(\mathbf{C}, D) = \mathit{Persist}(\mathbf{C}', D)\ \rangle \end{aligned}$$

3.4 Program steps

We now consider single steps of a program. We show that any program configuration reachable by a single step from a well-formed program configuration is itself well-formed. This allows us to confine our attention to well-formed configurations for the rest of this work.

We can consider the transition relation $\Longrightarrow$ in two ways. One way is as a static relation between pairs of program configurations. The other way is to think of the relation in in operational terms, where $\mathbf{C} \Longrightarrow \mathbf{C}'$ means that an executing *TCB* program in configuration $\mathbf{C}$ can change to configuration $\mathbf{C}'$ in one indivisible step. In the former view we consider conditions on pairs of configurations so that they are in the transition relation. In the latter view, we consider conditions on a configuration so that it has a successor in the transition relation.

We prove some properties of the transition relation, or, equivalently, the steps allowed in the execution of a *TCB* program.

3.4.1 Steps and configurations

We define some terminology for the boxes where a rule is applied.

Definition 3.31 *For* $\mathbf{C}, \mathbf{C}' \in PC$, *such that there is a step* $\mathbf{C} \Longrightarrow \mathbf{C}'$, *the* loci *of this step are the boxes which satisfy the conditions of the semantic rule used to admit pair* $(\mathbf{C}, \mathbf{C}')$.

If $\mathbf{C} \Longrightarrow \mathbf{C}'$, then $\mathbf{C}$ and $\mathbf{C}'$ are mostly the same, as the following theorem shows.

Theorem 3.32

$$\begin{array}{l}\langle\forall\ \mathbf{C}, \mathbf{C}', D \\ \ :\ \mathbf{C} \Longrightarrow \mathbf{C}' \ \wedge\ D \in \mathbf{B} \ \wedge\ D \textit{ is not a locus for } \mathbf{C} \Longrightarrow \mathbf{C}' \\ \ :\ \mathbf{C}.D = \mathbf{C}'.D \\ \ \rangle\end{array}$$

Proof

We note that each semantic rule requires that the configurations for the boxes not mentioned in the rule be the same before and after the step. By Definition 3.31, the boxes mentioned in the rule are the loci for any step admitted using the rule. Thus, the boxes other than the loci are unchanged by the step.
(End of proof)

We show that any program configuration reachable in a single step from a well-formed configuration is well-formed. The proof is in Appendix B.

Theorem 3.33

$$\langle\forall\ \mathbf{C}, \mathbf{C}' \ :\ \mathbf{C} \in PC \ \wedge\ \mathbf{C} \Longrightarrow \mathbf{C}' \ :\ \mathbf{C}' \in PC\ \rangle$$

3.4.2 Step labels

By definition, $\mathbf{C} \Longrightarrow \mathbf{C}'$ means that there is some inference rule satisfied by $\mathbf{C}$ and $\mathbf{C}'$. There is never more than one rule satisfied, as the following theorem shows.

Theorem 3.34 *If $\mathbf{C} \Longrightarrow \mathbf{C}'$, then there is a unique rule satisfied by $(\mathbf{C}, \mathbf{C}')$.*

Proof

Assume $\mathbf{C} \Longrightarrow \mathbf{C}'$. If $(\mathbf{C}, \mathbf{C}')$ satisfies a two-locus rule, then, by examining the rules we can see that $\mathbf{C}$ and $\mathbf{C}'$ differ on the configuration of both loci. If, on the other hand, $(\mathbf{C}, \mathbf{C}')$ satisfies the conditions of a one-locus rule, then, by Theorem 3.32, $\mathbf{C}$

Rule	$\mathbf{C}.D.\phi$	$\mathbf{C}'.D.\phi$
(action-start)	$\hat{\phi}$	$\hat{\phi}$
(p-action-init)	IDLE	GUARD
(p-method-init)	IDLE	GUARD
(t-action-init)	IDLE	ACCEPT
(t-method-init)	IDLE	ACCEPT
(guard-accept)	GUARD	ACCEPT
(guard-reject)	GUARD	REJECT
(local-step)	ACCEPT	ACCEPT
(proc-term)	ACCEPT	RETURN
(action-end)	RETURN	IDLE
(action-reject)	REJECT	IDLE

Table 3.1: Conditions on the box phase for one-locus rules.

and $\mathbf{C}'$ agree on the configurations of all boxes except one, the locus.[1] Thus, if rules l and l' are both satisfied by $(\mathbf{C}, \mathbf{C}')$, then l and l' have the same number of loci.

Table 3.1 shows the required values of the phase component of D for $\mathbf{C}$ and $\mathbf{C}'$ according to each one-locus rule. All the rules in the table require specific values for the phase of the locus in $\mathbf{C}$ and $\mathbf{C}'$, except for **(action-start)**. All the rules require that $\mathbf{C}$ and $\mathbf{C}'$ differ on this value, except for **(action-start)** and **(local-step)**, which require that the phase be the same before and after.

By comparing rows of the table, we observe that the only pairs of rules that can possibly both be satisfied by a pair $(\mathbf{C}, \mathbf{C}')$ are

(action-start)
(local-step)

[1]We discuss below whether a pair of configurations satisfying a one-locus rule differ on the configuration of the locus. For our present purposes, we can ignore this question.

and

(t-action-init)

(t-method-init)

For the first pair, we observe from the rules that

$$(\mathbf{C}, \mathbf{C}') \text{ satisfies } \textbf{(action-start)} \quad \Rightarrow \quad \mathbf{C}.D.\gamma \neq \mathbf{C}'.D.\gamma$$

$$(\mathbf{C}, \mathbf{C}') \text{ satisfies } \textbf{(local-step)} \quad \Rightarrow \quad \mathbf{C}.D.\gamma = \mathbf{C}'.D.\gamma$$

Thus there is no pair $(\mathbf{C}, \mathbf{C}')$ that satisfies both of these rules.

For the second pair, we have that

$$(\mathbf{C}, \mathbf{C}') \text{ satisfies } \textbf{(t-action-init)} \quad \Rightarrow \quad \mathit{Proc}(\mathbf{C}.D.\gamma) \in \mathit{TotActs}(D)$$

$$(\mathbf{C}, \mathbf{C}') \text{ satisfies } \textbf{(t-method-init)} \quad \Rightarrow \quad \mathit{Proc}(\mathbf{C}.D.\gamma) \in \mathit{TotMeths}(D)$$

Since $\mathit{TotActs}(D)$ and $\mathit{TotMeths}(D)$ are disjoint, the conditions are exclusive. This completes the proof for the one-locus rules.

For the two-locus rules, we show the phase for D (the source) and E (the agent). The configurations agree on the configuration of all boxes other than these. Table 3.2 shows the required values of the phase component of D and E for $\mathbf{C}$ and $\mathbf{C}'$ according to each two-locus rule. The phase for E is not determined for the rules **(total-call)** and **(guard-test)**, but it is the same in $\mathbf{C}$ and $\mathbf{C}'$, as with **(action-start)**. Comparing the table rows, we see that there is no pair of two-locus rules that are both satisfied by $(\mathbf{C}, \mathbf{C}')$. This completes the proof for the two-locus rules.
(End of proof)

Each rule in the semantics defines a relation, containing all pairs $(\mathbf{C}, \mathbf{C}')$ that satisfy the rule. Clearly $\Longrightarrow$ is the union of all these relations, and we have from Theorem 3.34 that all of these single-rule relations are disjoint.

Rule	$\mathbf{C}.D.\phi$ $\mathbf{C}.E.\phi$	$\mathbf{C}'.D.\phi$ $\mathbf{C}'.E.\phi$
(total-call)	ACCEPT $\hat{\phi}$	WAIT $\hat{\phi}$
(guard-test)	GUARD $\hat{\phi}$	PWAIT $\hat{\phi}$
(total-return)	WAIT RETURN	ACCEPT IDLE
(test-accept)	PWAIT RETURN	ACCEPT IDLE
(test-reject)	PWAIT REJECT	REJECT IDLE

Table 3.2: Conditions on the box phase for two-locus rules.

Definition 3.35 *For an inference rule* **(***l***)***, we define a relation with the same name by the following.*

$$l \;=\; \{\, (\mathbf{C}, \mathbf{C}') \mid \mathbf{C} \Longrightarrow \mathbf{C}' \textit{ satisfies } \mathbf{(}l\mathbf{)} \,\}$$

The following Theorems summarize the properties of the relations for the semantic rules for *TCB*.

Theorem 3.36 *If* **(***l***)** *and* **(***l'***)** *are semantic rules, and* $l \neq l'$*, then*

$$l \;\; \text{disj} \;\; l'$$

Theorem 3.37 *If* $\{\, l_i \mid 0 \leq i < 16 \,\}$ *be the set of relations for the rules in the queue semantics of TCB, then*

$$\Longrightarrow \;=\; \langle \cup\, i \;:\; 0 \leq i < 16 \;:\; l_i \,\rangle$$

We can further partition the relation for each rule by considering the loci in the application of the rule.

Definition 3.38 *For* **(**l**)** *a one-locus rule, and* $D \in \mathbf{B}$*,*

$$l(D) \quad = \quad \{\, (\mathbf{C}, \mathbf{C}') \mid \mathbf{C}, \mathbf{C}' \in PC \;\wedge\; (\mathbf{C}, \mathbf{C}') \textit{ satisfies } \mathbf{(}l\mathbf{)} \textit{ with locus } D \,\}$$

For **(**l**)** *a two-locus rule, and* $D, E \in \mathbf{B}$*,*

$$\begin{aligned} l(D, E) \quad = \quad \{\; & (\mathbf{C}, \mathbf{C}') \\ \mid\; & \mathbf{C}, \mathbf{C}' \in PC \;\wedge \\ & (\mathbf{C}, \mathbf{C}') \textit{ satisfies } \mathbf{(}l\mathbf{)} \textit{ with source } D \textit{ and agent } E \\ \}\; & \end{aligned}$$

We write $\mathbf{L}$ *for a typical label, either a one-locus label* $l(D)$*, or a two locus label* $l(D, E)$*. We write* $\mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}'$ *for* $(\mathbf{C}, \mathbf{C}') \in \mathbf{L}$*. We call* $\mathbf{L}$ *a* label *or* step*.*

We use the fact each label is a subrelation of the relation for one of the rules, and we write, for example

$$\mathbf{L} \;\subseteq\; \textsf{\textbf{local-step}}$$

to mean

$$\langle \exists\, D \;::\; \mathbf{L} = \textsf{\textbf{local-step}}(D) \;\rangle$$

and

$$\mathbf{L} \;\subseteq\; \textsf{\textbf{total-call}}$$

to mean

$$\langle \exists\, D, E \;::\; \mathbf{L} = \textsf{\textbf{total-call}}(D, E) \;\rangle$$

We define the set containing all the labels for the queue semantics.

Definition 3.39

$$
\begin{array}{lll}
Lab & \triangleq & \{ \ \mathbf{L} \\
& & \mid \ \mathbf{L} \ \textit{is a label generated by one of} \\
& & \quad \textbf{(action-start)} \quad \textbf{(guard-reject)} \\
& & \quad \textbf{(guard-test)} \quad \textbf{(local-step)} \\
& & \quad \textbf{(total-call)} \quad \textbf{(proc-term)} \\
& & \quad \textbf{(p-action-init)} \quad \textbf{(action-end)} \\
& & \quad \textbf{(p-method-init)} \quad \textbf{(action-reject)} \\
& & \quad \textbf{(t-action-init)} \quad \textbf{(test-accept)} \\
& & \quad \textbf{(t-method-init)} \quad \textbf{(test-reject)} \\
& & \quad \textbf{(guard-accept)} \quad \textbf{(total-return)} \\
& & \}
\end{array}
$$

We use the following function to refer to the loci of a label.

Definition 3.40 *For* $\mathbf{L} \in Lab$,

$$
\begin{array}{llll}
Loci(\mathbf{L}) & \triangleq & \{D\} & \text{if} \quad \mathbf{L} = l(D) \\
& & \{D, E\} & \text{if} \quad \mathbf{L} = l(D, E)
\end{array}
$$

We have the following result.

Theorem 3.41

$$
\begin{array}{l}
\langle \forall \ \mathbf{C}, \mathbf{C}' \\
\quad : \ \mathbf{C}, \mathbf{C}' \in PC \ \wedge \ \mathbf{C} \Longrightarrow \mathbf{C}' \ \wedge \ \mathbf{C} \neq \mathbf{C}' \\
\quad : \ \langle \exists! \, \mathbf{L} \ : \ \mathbf{L} \in Lab \ : \ \mathbf{C} \langle \mathbf{L} \rangle \mathbf{C}' \ \rangle \\
\quad \rangle
\end{array}
$$

Proof

We already have, from Theorem 3.34, that there is a unique rule satisfied by any $(\mathbf{C}, \mathbf{C}') \in \Longrightarrow$. Thus, we need only show for a one-locus rule (l_1), that $l_1(D)$ and $l_1(D')$ are disjoint for $D \neq D'$, and for a two-locus rule (l_2), that $l_2(D, E)$ and $l_2(D', E')$ are disjoint for $(D, E) \neq (D', E')$.

For (l_1), assume that we have $D, D' \in \mathbf{B}$, and $(\mathbf{C}, \mathbf{C}')$, such that $\mathbf{C} \neq \mathbf{C}'$, $(\mathbf{C}, \mathbf{C}') \in l_1(D)$, and $(\mathbf{C}, \mathbf{C}') \in l_1(D')$. First we have

$$
\begin{array}{ll}
 & (\mathbf{C}, \mathbf{C}') \in l_1(D) \\
\Rightarrow & \quad \{ \text{ Theorem 3.32 } \} \\
 & \langle \forall\, E \ : \ E \neq D \ : \ \mathbf{C}.E = \mathbf{C}'.E \ \rangle \\
\equiv & \quad \{ \text{ trading; } \mathbf{C} \neq \mathbf{C}' \ \} \\
 & \langle \forall\, E \ : \ \mathbf{C}.E \neq \mathbf{C}'.E \ : \ E = D \ \rangle \ \wedge \ \langle \exists\, E \ :: \ \mathbf{C}.E \neq \mathbf{C}'.E \ \rangle \\
\equiv & \quad \{ \ \wedge \text{ over } \exists \ \} \\
 & \langle \exists\, E \ :: \ \langle \forall\, E \ : \ \mathbf{C}.E \neq \mathbf{C}'.E \ : \ E = D \ \rangle \ \wedge \ \mathbf{C}.E \neq \mathbf{C}'.E \ \rangle \\
\Rightarrow & \quad \{ \text{ instantiation } \} \\
 & \langle \exists\, E \ :: \ E = D \ \wedge \ \mathbf{C}.E \neq \mathbf{C}'.E \ \rangle \\
\equiv & \quad \{ \text{ one-point rule } \} \\
 & \mathbf{C}.D \neq \mathbf{C}'.D
\end{array}
$$

Now we have

$$
\begin{array}{ll}
 & (\mathbf{C}, \mathbf{C}') \in l_1(D) \ \wedge \ (\mathbf{C}, \mathbf{C}') \in l_1(D') \\
\Rightarrow & \quad \{ \text{ Theorem 3.32, above result } \} \\
 & \langle \forall\, E \ : \ E \neq D \ : \ \mathbf{C}.E = \mathbf{C}'.E \ \rangle \ \wedge \ \mathbf{C}.D' \neq \mathbf{C}'.D' \\
\Rightarrow & \quad \{ \text{ instantiate for } D' \ \} \\
 & (D' \neq D \Rightarrow \mathbf{C}.D' = \mathbf{C}'.D') \ \wedge \ \mathbf{C}.D' \neq \mathbf{C}'.D' \\
\equiv & \quad \{ \text{ contraposition } \} \\
 & (\mathbf{C}.D' \neq \mathbf{C}'.D' \Rightarrow D' = D) \ \wedge \ \mathbf{C}.D' \neq \mathbf{C}'.D' \\
\Rightarrow & \quad \{ \text{ modus ponens } \} \\
 & D' = D
\end{array}
$$

For (l_2), assume that we have $D, E, D', E' \in \mathbf{B}$, and $(\mathbf{C}, \mathbf{C}')$ such that $(\mathbf{C}, \mathbf{C}') \in l_2(D, E)$, and $(\mathbf{C}, \mathbf{C}') \in l_2(D', E')$. We first note that $D \neq E$, and $D' \neq E'$, since no such methods calls appear in a well-formed program. Also, as noted in the proof of Theorem 3.34,

$$(\mathbf{C}, \mathbf{C}') \in l(D, E) \;\; \Rightarrow \;\; \mathbf{C}.D \neq \mathbf{C}'.D \;\; \wedge \;\; \mathbf{C}.E \neq \mathbf{C}'.E$$

We have

$$\begin{array}{ll}
 & (\mathbf{C}, \mathbf{C}') \in l_2(D, E) \;\; \wedge \;\; (\mathbf{C}, \mathbf{C}') \in l_2(D', E') \\
\Rightarrow & \qquad \{ \text{ Theorem 3.42, above observation } \} \\
 & \langle \forall\, D'' \;\; : \;\; D'' \notin \{D, E\} \;\; : \;\; \mathbf{C}.D'' = \mathbf{C}'.D'' \;\rangle \;\; \wedge \;\; \mathbf{C}.D' \neq \mathbf{C}'.D' \\
\Rightarrow & \qquad \{ \text{ instantiate for } D' \; \} \\
 & (D' \notin \{D, E\} \Rightarrow \mathbf{C}.D' = \mathbf{C}'.D') \;\; \wedge \;\; \mathbf{C}.D' \neq \mathbf{C}'.D' \\
\equiv & \qquad \{ \text{ contraposition } \} \\
 & (\mathbf{C}.D' \neq \mathbf{C}'.D' \Rightarrow D' \in \{D, E\}) \;\; \wedge \;\; \mathbf{C}.D' \neq \mathbf{C}'.D' \\
\Rightarrow & \qquad \{ \text{ modus ponens } \} \\
 & D' \in \{D, E\}
\end{array}$$

Similarly, we can show that $E' \in \{D, E\}$, and since $D' \neq E'$, we have $\{D', E'\} = \{D, E\}$. To complete the proof, we need to show that $l_2(D, E)$ and $l_2(E, D)$ are disjoint. From Table 3.2 we observe that there is no two-locus rule where the source and the agent have equal phases before the step, and equal phases after the step. Thus the relations are disjoint.

(End of proof)

As noted in the above proof, for $(\mathbf{C}, \mathbf{C}') \in l(D, E)$, we have $\mathbf{C}.D \neq \mathbf{C}'.D$, and $\mathbf{C}.E \neq \mathbf{C}'.E$. For $(\mathbf{C}, \mathbf{C}') \in l(D)$, for almost all cases, we similarly have $\mathbf{C}.D \neq \mathbf{C}'.D$. The one exception is $(\mathbf{C}, \mathbf{C}') \in$ **local-step**(D). Suppose

$$\mathbf{C}.D = (\text{ACCEPT}, \sigma, \gamma, \theta)$$

and there is a *TCBloc* transition

$$(\sigma, \theta) \longrightarrow (\sigma, \theta)$$

then

$$(\mathbf{C}, \mathbf{C}) \in \textsf{\textbf{local-step}}(D)$$

Such a step is called a *stuttering step.* Stuttering steps are problematic because, in operational terms, the configuration after the step is the same as that before, so the step can be repeated indefinitely, and thus a procedure call can execute an infinite number of steps without terminating.

For a step that is not stuttering, we can rewrite Theorem 3.32 in a stronger form, as follows.

Theorem 3.42

$$\begin{array}{l} \langle \forall\ \mathbf{L}, \mathbf{C}, \mathbf{C}' \\ \quad : \mathbf{L} \in \mathit{Lab} \ \wedge\ \mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}' \ \wedge\ \mathbf{L} \textit{ is not a stuttering step} \\ \quad : \langle \forall\ D \ :\ D \in \mathbf{B} \ :\ D \in \mathit{Loci}(\mathbf{L}) \ \not\equiv\ \mathbf{C}.D = \mathbf{C}'.D \ \rangle \\ \quad \rangle \end{array}$$

Proof

Assume we have $\mathbf{C}, \mathbf{C}' \in PC$, and $\mathbf{L} \in \mathit{Lab}$, such that $\mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}'$ and $\mathbf{L}$ is not a stuttering step, and $D \in \mathbf{B}$. If $D \notin \mathit{Loci}(\mathbf{L})$, then, by Theorem 3.32, $\mathbf{C}.D = \mathbf{C}'.D$. If $D \in \mathit{Loci}(\mathbf{L})$, then, since $\mathbf{L}$ is not stuttering, we have from the above discussion, $\mathbf{C}.D \neq \mathbf{C}'.D$.

(End of proof)

The final result in this section shows that the changes a step makes to its loci are independent of the configurations of the other boxes in the program.

Theorem 3.43

$$
\begin{array}{l}
\langle \forall\ \mathbf{L}, \mathbf{C}_0, \mathbf{C}_0', \mathbf{C}_1 \\
\quad : \mathbf{L} \in \mathit{Lab} \ \wedge\ \mathbf{C}_0 \langle \mathbf{L} \rangle \mathbf{C}_0' \ \wedge\ \mathbf{C}_1 \in \mathit{PC} \ \wedge \\
\quad\quad \langle \forall\ D \ :\ D \in \mathit{Loci}(\mathbf{L}) \ :\ \mathbf{C}_0.D = \mathbf{C}_1.D\ \rangle \\
\quad : \langle \exists\ \mathbf{C}_1' \ :\ \mathbf{C}_1 \langle \mathbf{L} \rangle \mathbf{C}_1' \ :\ \langle \forall\ D \ :\ D \in \mathit{Loci}(\mathbf{L}) \ :\ \mathbf{C}_0'.D = \mathbf{C}_1'.D\ \rangle\ \rangle \\
\rangle
\end{array}
$$

Proof
By examination of the rules.
(End of proof)

3.4.3 Enabled steps

In the last section we showed that, for a given pair of configurations $(\mathbf{C}, \mathbf{C}')$, there is at most one step $\mathbf{L}$ such that $\mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}'$. We now turn to the operational view of a program and consider, for a given program configuration $\mathbf{C}$, how many steps there are where $\mathbf{C}$ is the configuration before the step.

Definition 3.44 *For* $\mathbf{C} \in PC$, *and* $\mathbf{L} \in Lab$

$$\mathbf{L} \text{ is enabled in } \mathbf{C} \ \triangleq\ \langle \exists\ \mathbf{C}' \ ::\ \mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}'\ \rangle$$

Examining the rules, we see that there are three cases where there is no restriction on the configuration of a locus before a step: an **action-start**(D) step is enabled for any configuration of D; and a **total-call**(E, D), or **guard-test**(E, D) step is enabled in any configuration where E has the right configuration, regardless of D's configuration. We define some notation to distinguish these steps.

Definition 3.45 *For $D \in \mathbf{B}$,*

$$
\begin{aligned}
Uncond(D) &\triangleq \{\textsf{action-start}(D)\} \cup \\
&\quad \{\ l, E \\
&\quad :\ l \in \{\textsf{total-call}, \textsf{guard-test}\}\ \wedge\ E \in \mathbf{B} \setminus \{D\} \\
&\quad :\ l(E, D) \\
&\quad \} \\
Cond(D) &\triangleq \{\ \mathbf{L} \mid D \in Loci(\mathbf{L})\ \wedge\ \mathbf{L} \notin Uncond(D)\ \}
\end{aligned}
$$

For $\mathbf{L} \in Lab$,

$$
\begin{aligned}
ULoci(\mathbf{L}) &\triangleq \{\ D \mid \mathbf{L} \in Uncond(D)\ \} \\
CLoci(\mathbf{L}) &\triangleq \{\ D \mid \mathbf{L} \in Cond(D)\ \}
\end{aligned}
$$

We call $Uncond(D)$ the unconditional steps*, and $Cond(D)$ the* conditional steps*, for D. We call $ULoci(\mathbf{L})$ the* unconditional loci*, and $CLoci(\mathbf{L})$ the* conditional loci *for $\mathbf{L}$.*

The unconditional steps for D are those which have D as a locus, but which are enabled regardless of D's configuration. The conditional steps for D are the remaining steps with D as a locus. By definition, $Uncond(D)$ and $Cond(D)$ partition the set of steps with D as a locus, and so $ULoci(\mathbf{L})$ and $CLoci(\mathbf{L})$ partition $Loci(\mathbf{L})$.

We note that **action-start**(D) is an unconditional step for its only locus, and that **total-call**(E, D) **guard-test**(E, D) are unconditional steps for D, and conditional steps for E. Thus **action-start**(D) is the only step that is unconditional for all of its loci. In a quiescent configuration, there is no box with a conditional step enabled, and only **action-start** steps are enabled.

The following theorem shows that the configurations of the conditional loci determine whether or not a step is enabled. If $D \notin CLoci(\mathbf{L})$, in particular, if

$D \in ULoci(\mathbf{L})$, then $\mathbf{L}$ is enabled regardless of D's configuration.

Theorem 3.46

$$\begin{array}{l}\langle\forall\ \mathbf{L}, \mathbf{C}, \mathbf{C}' \\ \quad :\ \mathbf{L} \in Lab\ \ \wedge\ \ \mathbf{C}, \mathbf{C}' \in PC\ \ \wedge\ \ \langle\forall\ D\ \ :\ \ D \in CLoci(\mathbf{L})\ \ :\ \ \mathbf{C}.D = \mathbf{C}'.D\ \rangle \\ \quad :\ \mathbf{L}\ \mathit{is\ enabled\ in}\ \mathbf{C}\ \ \equiv\ \ \mathbf{L}\ \mathit{is\ enabled\ in}\ \mathbf{C}' \\ \quad \rangle\end{array}$$

Proof

Use Definition 3.45 and the semantic rules.

(End of proof)

Table 3.3 shows the conditions on a program configuration $\mathbf{C}$ such that a conditional step for D is enabled. In the case of the two-locus rules that are conditional for both loci, the conditions on the other locus, box E, are also given. In each rule, E is $K.\mathbf{C}.D$ or $W.\mathbf{C}.D$, depending on the rule.

The table represents a partition of the configurations, from the point of view of box D. The primary partition is on the phase of D in $\mathbf{C}$. Each line to the right of a phase represents a condition on the phase disjoint from all others for that phase. The case analysis is exhaustive for each phase. For phases RETURN and REJECT, this relies on the fact that $\mathbf{C}$ is locally well-formed and call correct.

The following theorem shows an important inference we can draw from the table, that there is at most one conditional step enabled for a box in any well-formed configuration.

$\mathbf{C}.D.\phi$	Other components		Enabled step
IDLE	$\mathbf{C}.D.\gamma = \perp$		none
	$Proc(\mathbf{C}.D.\gamma) \in PartActs(D)$		**p-action-init**(D)
	$Proc(\mathbf{C}.D.\gamma) \in PartMeths(D)$		**p-method-init**(D)
	$Proc(\mathbf{C}.D.\gamma) \in TotActs(D)$		**t-action-init**(D)
	$Proc(\mathbf{C}.D.\gamma) \in TotMeths(D)$		**t-method-init**(D)
GUARD	$Proc(\mathbf{C}.D.\gamma) = p$	$Alt(D.p, \mathbf{C}.D.\sigma) = \perp$	**guard-reject**(D)
		$Alt(D.p, \mathbf{C}.D.\sigma) = (\perp, \theta)$	**guard-accept**(D)
		$Alt(D.p, \mathbf{C}.D.\sigma) = (E.m(-), \theta)$	**guard-test**(D, E)
PWAIT	$Wp.\mathbf{C}.D = E$	$\mathbf{C}.E.\phi =$ RETURN	**test-accept**(D, E)
		$\mathbf{C}.E.\phi =$ REJECT	**test-reject**(D, E)
		otherwise	none
	otherwise		none
ACCEPT	$\mathbf{C}.D.\theta = \perp$		**proc-term**(D)
	$\text{first}(\mathbf{C}.D.\theta) = E.m(-)$		**total-call**(D, E)
	$\langle \exists\, \sigma', \theta' \ :: \ (\, \mathbf{C}.D.\sigma \, , \, \mathbf{C}.D.\theta \,) \longrightarrow (\sigma', \theta') \ \rangle$		**local-step**(D)
	otherwise		none
WAIT	$Wt.\mathbf{C}.D = E$	$\mathbf{C}.E.\phi =$ RETURN	**total-return**(D, E)
		otherwise	none
	otherwise		none
RETURN	$K.\mathbf{C}.D = \perp$		**action-end**(D)
	$Kp.\mathbf{C}.D = E$		**test-accept**(E, D)
	$Kt.\mathbf{C}.D = E$		**total-return**(E, D)
REJECT	$Kp.\mathbf{C}.D = \perp$		**action-reject**(D)
	$Kp.\mathbf{C}.D = E$		**test-reject**(E, D)

Table 3.3: Conditions on $\mathbf{C}$ to enable a conditional step for D.

Theorem 3.47

$$
\begin{array}{l}
\langle \forall\ \mathbf{C}, D \\
\quad : \mathbf{C} \in PC \ \wedge\ D \in \mathbf{B} \\
\quad : \langle \#\, \mathbf{L}\ :\ \mathbf{L} \in \mathit{Cond}(D)\ :\ \mathbf{L}\ \textit{is enabled in}\ \mathbf{C}\ \rangle \leq 1 \\
\quad \rangle
\end{array}
$$

Proof

We check that each pair of rows in Table 3.3 describes disjoint conditions on $\mathbf{C}.D$, and thus there is no configuration in which two conditional steps are enabled.
(End of proof)

Theorem 3.47 allows the definition of the following function.

Definition 3.48 *For* $\mathbf{C} \in PC$, *and* $D \in \mathbf{B}$,

$$
\begin{array}{lcll}
\mathit{Enabled}(\mathbf{C}, D) & \triangleq & \mathbf{L} & \text{if } \mathbf{L} \in \mathit{Cond}(D) \wedge \mathbf{L}\ \textit{enabled in}\ \mathbf{C} \\
 & & \bot & \text{if } \textit{no such}\ \mathbf{L}\ \textit{exists}
\end{array}
$$

We call $\mathit{Enabled}(\mathbf{C}, D)$ *the* enabled step *for* D *in* $\mathbf{C}$.

Suppose $\mathbf{C}.D.\phi = \textsc{accept}$, then, according to Table 3.3, no step is enabled for box D if $\mathbf{C}.D.\theta \neq \bot$, $\mathbf{C}.D.\theta \neq E.m(-)$, and there is no local step $(\ \mathbf{C}.D.\sigma\ ,\ \mathbf{C}.D.\theta\ ) \longrightarrow (\sigma', \theta')$. We note from the rules defining *TCBloc* that there is a step defined for any (σ, θ) provided $\theta \neq \bot$, and $\mathrm{first}(\theta) \neq E.m(-)$. Thus the "otherwise" condition of this line for this phase never applies, and $\mathit{Enabled}(\mathbf{C}, D) \neq \bot$. This is not a general property of sequential language semantics, so, in the interests of generality, we make it an explicit assumption.

Assumption 3.49

$$\begin{array}{l}\langle\forall\ \mathbf{C}, D \\ \quad : \mathbf{C} \in PC \ \wedge\ D \in \mathbf{B} \ \wedge\ \mathbf{C}.D.\phi = \textsc{accept} \\ \quad : \mathit{Enabled}(\mathbf{C}, D) \neq \bot \\ \quad \rangle\end{array}$$

For Seuss languages where Assumption 3.49 does not hold, the following theorem, and some of the results that follow from it — in particular the complete execution theorem — require additional conditions.

The following theorem gives the conditions under which no conditional step is enabled for a box.

Theorem 3.50

$$\begin{array}{lll}\langle\forall\ \mathbf{C}, D & & \\ \quad : \mathbf{C} \in PC \ \wedge\ D \in \mathbf{B} & & \\ \quad : \mathit{Enabled}(\mathbf{C}, D) = \bot & \equiv & (\ \mathbf{C}.D.\phi = \textsc{idle} \ \wedge\ \mathbf{C}.D.\gamma = \bot\ ) \qquad \vee \\ & & (\ \mathbf{C}.D.\phi = \textsc{pwait} \ \wedge \\ & & \quad \mathbf{C}.(\ Wp.\mathbf{C}.D).\phi \notin \{\textsc{return}, \textsc{reject}\}\ )\ \vee \\ & & (\ \mathbf{C}.D.\phi = \textsc{wait} \ \wedge \\ & & \quad \mathbf{C}.(\ Wt.\mathbf{C}.D).\phi \neq \textsc{return}\ ) \\ \quad \rangle & & \end{array}$$

Proof

We observe that the conditions given are exactly those in which there is an entry "none" in the final column of Table 3.3. As noted above, Assumption 3.49 excludes the case where $\mathbf{C}.D.\phi = \textsc{wait}$.

(End of proof)

The following theorem shows an important stability property of $\mathit{Enabled}(\mathbf{C}, D)$. The proof is in Appendix B.

Theorem 3.51

$$
\begin{array}{l}
\langle\forall\ \mathbf{L}, \mathbf{C}, \mathbf{C}', D \\
\quad :\ \mathbf{L} \in \mathit{Lab}\ \wedge\ \mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}'\ \wedge\ D \in \mathbf{B}\ \wedge\ \mathit{Enabled}(\mathbf{C}, D) \notin \{\bot, \mathbf{L}\} \\
\quad :\ \mathit{Enabled}(\mathbf{C}, D) = \mathit{Enabled}(\mathbf{C}', D) \\
\quad \rangle
\end{array}
$$

3.4.4 Enabled steps for a call stack

We showed in Section 3.3.3 that for each action call active in $\mathbf{C} \in PC$ there is a call stack Ω which contains all boxes that are executing on behalf of that action call. Every nonquiescent box in the configuration is in exactly one call stack, and all the boxes in each call stack, except possibly the first, are in a waiting phase. We consider what conditional steps are enabled for the boxes in a call stack Γ. For $0 < i < |\Omega|$, we have

$$\mathbf{C}.\Omega[i].\phi \quad \in \quad \{\text{PWAIT}, \text{WAIT}\}$$

So, by Theorem 3.50, there is no conditional step enabled for $\Omega[i]$, for $1 < i < |\Omega|$. Thus only $\Omega[0]$ and $\Omega[1]$ may have enabled steps. A step is enabled for $\Omega[1]$ only if $\mathbf{C}.\Omega[0].\phi \in \{\text{RETURN}, \text{REJECT}\}$, and the enabled step, which is a **(test-accept)**, **(test-reject)**, or **(total-return)** step, is also a conditional step for $\Omega[0]$. Therefore, if a conditional step is enabled for any box in the stack, it is enabled for $\Omega[0]$.

Box $\Omega[0]$ is not quiescent, so by Theorem 3.50 it has a conditional step enabled unless it is waiting for box $D = W.\mathbf{C}.(\Omega[0])$, and D is not ready to return. If there is no conditional step enabled for $\Omega[0]$, then D is not executing for $\Omega[0]$, since $\Omega[0] \notin \text{rng}(K.\mathbf{C})$, by Definition 3.21. In this case, the next step for the thread

is a method initialization step at D. In the configuration resulting from this step, D is added to the front of the call stack.

Thus we see that there is at most one step enabled for a thread in any well-formed configuration.

3.5 Executions

We consider the execution of a program under the queue semantics from Chapter 2. We define an execution as a sequence of configurations and a parallel sequence of steps. Each pair of adjacent configurations in the configuration sequence are related by the corresponding step in the step sequence. The execution records all the configurations seen during the program, and the steps taken to reach these configurations.

We define executions relative to a set of well-formed configurations, and a step of labels.

Definition 3.52 *For $P \subseteq PC$, and $L \subseteq P \times P$,*

$$
\begin{aligned}
\mathit{Executions}(P, L) \triangleq \{ & \ (\Gamma, \Delta) \\
| & \ \Gamma \in P^{\infty} \ \wedge \ \Delta \in L^{\infty} \ \wedge \ |\Gamma| = |\Delta| + 1 \ \wedge \\
& \ \langle \forall\, i \ : \ 0 \leq i < |\Delta| \ : \ (\Gamma[i], \Gamma[i+1]) \in \Delta[i] \ \rangle \\
\} &
\end{aligned}
$$

An execution is, as described above, a pair of sequences, one containing program configurations, and the other containing program steps. The first sequence contains the configurations attained during program execution, in order, and the second sequence contains the steps from one configuration to the next. The sequences can be finite or infinite, and if finite, the configuration sequence is one element longer than the step sequence. The configuration sequence cannot be empty. We define

a set of executions using the whole of PC for the configurations, and Lab for the labels.

Definition 3.53

$$Z \triangleq \mathit{Executions}(PC, Lab)$$

We define some functions and operators for executions.

Definition 3.54 *For* $\varepsilon \in Z$*, where* $\varepsilon = (\Gamma, \Delta)$*,*

$$\begin{array}{rcll}
\mathit{CSeq}(\varepsilon) & \triangleq & \Gamma & \\
\mathit{SSeq}(\varepsilon) & \triangleq & \Delta & \\
|\varepsilon| & \triangleq & |\Delta| & \\
\mathit{Start}(\varepsilon) & \triangleq & \text{first}(\Gamma) & \\
\mathit{Final}(\varepsilon) & \triangleq & \text{last}(\Gamma) & \text{if } \; |\varepsilon| < \infty \\
\mathit{First}(\varepsilon) & \triangleq & \text{first}(\Delta) & \\
\mathit{Last}(\varepsilon) & \triangleq & \text{last}(\Delta) & \text{if } \; |\varepsilon| < \infty \\
\varepsilon[i] & \triangleq & \Gamma[i] & \text{if } \; 0 \leq i \leq |\varepsilon| \\
\varepsilon\langle i \rangle & \triangleq & \Delta[i] & \text{if } \; 0 \leq i < |\varepsilon| \\
\varepsilon\langle i \ldots j \rangle & \triangleq & (\, \Gamma[i \ldots (j+1)] \, , \, \Delta[i \ldots j] \,) & \\
 & & & \text{if } \; 0 \leq i \leq j < |\varepsilon| \\
\varepsilon & \mathit{is} & \text{finite} & \text{if } \; |\varepsilon| < \infty \\
\varepsilon & \mathit{is} & \text{infinite} & \text{if } \; |\varepsilon| = \infty
\end{array}$$

Thus an execution ε takes $|\varepsilon|$ steps, starting from configuration $\mathit{Start}(\varepsilon)$. If the number of steps is finite, the execution terminates in configuration $\mathit{Final}(\varepsilon)$. Configuration $\varepsilon[i]$ is before step $\varepsilon\langle i \rangle$, and configuration $\varepsilon[i+1]$ is after. The segment $\varepsilon\langle i \ldots j \rangle$ is steps i through j of ε, with the corresponding configurations. This is an execution, since ε is.

If we have executions ε and ε', where ε is finite, and $Final(\varepsilon) = Start(\varepsilon')$, then we can compose ε and ε'. We also define composition for the case that ε is infinite.

Definition 3.55 *For $\varepsilon, \varepsilon' \in Z$ where $s = |\varepsilon|$, $s' = |\varepsilon'|$, and*

$$s = \infty \;\vee\; (s < \infty \;\wedge\; Final(\varepsilon) = Start(\varepsilon'))$$

the sequential composition *of ε and ε', written $\varepsilon;\varepsilon'$, is $(\Gamma, \Delta) \in PC^{\infty} \times Lab^{\infty}$, satisfying the following.*

$$|\Gamma| \;=\; s + s' + 1 \;\wedge\; |\Delta| \;=\; s + s' \;\wedge$$
$$\langle \forall\, i \;:\; 0 \le i < s \;:\; \Gamma[i] = \varepsilon[i] \;\wedge\; \Delta[i] = \varepsilon\langle i \rangle \;\rangle$$

and, if $s < \infty$,

$$\langle \forall\, i \;:\; s \le i \le s + s' \;:\; \Gamma[i] = \varepsilon'[i - s] \;\rangle \;\wedge$$
$$\langle \forall\, i \;:\; s \le i < s + s' \;:\; \Delta[i] = \varepsilon'\langle i - s \rangle \;\rangle$$

If ε is finite, and $\varepsilon;\varepsilon'$ is defined, then the matching configuration appears only once in $\varepsilon;\varepsilon'$. If, for example,

$$CSeq(\varepsilon) \;=\; \langle \mathbf{C}_0, \mathbf{C}_1, \mathbf{C}_2 \rangle$$
$$CSeq(\varepsilon') \;=\; \langle \mathbf{C}_2, \mathbf{C}_3 \rangle$$

then the configuration sequence for $\varepsilon;\varepsilon'$ is

$$\langle \mathbf{C}_0, \mathbf{C}_1, \mathbf{C}_2, \mathbf{C}_3 \rangle$$

The following results about sequential composition are simple to prove from the definition.

Theorem 3.56

$$\langle \forall\, \varepsilon, \varepsilon' \;:\; \varepsilon, \varepsilon' \in Z \;\wedge\; \varepsilon;\varepsilon' \textit{ is defined} \;:\; \varepsilon;\varepsilon' \in Z \;\rangle$$

Theorem 3.57

$$\begin{aligned}
&\langle \forall\; \varepsilon, \varepsilon', \varepsilon'' \\
&\quad : \varepsilon, \varepsilon', \varepsilon'' \in Z \;\wedge\; (\varepsilon;\varepsilon');\varepsilon'' \textit{ is defined} \\
&\quad : \varepsilon;(\varepsilon';\varepsilon'') \textit{ is defined} \;\wedge\; (\varepsilon;\varepsilon');\varepsilon'' = \varepsilon;(\varepsilon';\varepsilon'') \\
&\quad \rangle
\end{aligned}$$

The last theorem shows that sequential composition is associative, so we omit parentheses, and write $\varepsilon;\varepsilon';\varepsilon''$ for the sequential composition of ε, ε' and ε''.

We define an order on Z, using the order on the underlying sequences.

Definition 3.58 *For* $\varepsilon, \varepsilon' \in Z$,

$$\varepsilon \sqsubseteq \varepsilon' \;\triangleq\; \mathit{CSeq}(\varepsilon) \sqsubseteq \mathit{CSeq}(\varepsilon') \;\wedge\; \mathit{SSeq}(\varepsilon) \sqsubseteq \mathit{SSeq}(\varepsilon')$$

Theorem 3.59 $(Z, \sqsubseteq)$ *is a complete partial order.*

Since $(Z, \sqsubseteq)$ is a CPO, we can define infinite executions as the limit of a chain of finite executions.

3.6 Discussion

The conditions we define for well-formed box and program configurations exclude many problematic or illogical configurations from consideration. We showed that our definition is sound in that a step from a well-formed configuration reaches another well-formed configuration.

The set of well-formed configurations is larger than the set of configurations that we expect to see during the execution of a program. There is a restricted set of values that the code component for a box can take during the execution of a program. For *TCB*, we do not expect to encounter the statement "$x := 2$" at the front of the code component unless this statement appears somewhere in the body code for the currently executing procedure. Definition 3.5 places only two restrictions on the code component: that it be empty when the phase is IDLE, GUARD, RETURN or REJECT, and that its first element be a procedure call when the phase is PWAIT or WAIT. Other than these restrictions, a well-formed box configuration can have anything in its code component. In particular, we can form a well-formed box configuration in which box D is waiting for the return of a method call to box E, and E is executing the call, but there is no call statement for a method in E anywhere in the program code of box D.

When we require stronger conditions on the code component than those given by Definitions 3.5 and 3.20, as we do when we discuss deadlock in the next chapter, we restrict our attention to configurations that can appear in an execution starting from a quiescent state. This ensures that every statement that appears in the code component for a box appears in the program code for the box. Restricting the set of configurations in this way limits our attention to the *reachable* configurations.

3.6.1 Deterministic and nondeterministic steps

The queue semantics has the pleasant properties that in any configuration, there is at most one conditional step enabled for a box and, once enabled, a conditional step remains enabled until it is taken.

It can be shown that every step in the semantics, with two exceptions, is *deterministic*. That is, for any program configuration $\mathbf{C}$ and label $\mathbf{L}$, if $\mathbf{L}$ is enabled in $\mathbf{C}$, there is exactly one $\mathbf{C}'$ such that $\mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}'$. If $\mathbf{L}$ is deterministic, the transition

relation is a function. The two potentially nondeterministic steps are **action-start** and **local-step**.

For nondeterministic **action-start** steps, consider a box D with actions $a0$ and $a1$. Choose $\mathbf{C}, \mathbf{C}_0, \mathbf{C}_1 \in PC$, such that

$$\begin{aligned} \mathbf{C}.D &= (\phi, \sigma, \gamma, \theta) \\ \mathbf{C}_0 &= \mathbf{C}\,[D \mapsto (\phi, \sigma, \gamma \triangleleft (a0), \theta)] \\ \mathbf{C}_1 &= \mathbf{C}\,[D \mapsto (\phi, \sigma, \gamma \triangleleft (a1), \theta)] \end{aligned}$$

We have $\mathbf{C}_0 \neq \mathbf{C}_1$, and $(\mathbf{C}, \mathbf{C}_0), (\mathbf{C}, \mathbf{C}_1) \in$ **action-start**(D).

A **local-step** is defined by a transition for the local language *TCBloc*. All steps for this language are deterministic. But it is not difficult to define a Seuss language with a nondeterministic local language. For such a language, there are local configurations (σ, θ), (σ_0, θ_0), and (σ_1, θ_1), where

$$\begin{aligned} (\sigma, \theta) &\longrightarrow (\sigma_0, \theta_0) \\ (\sigma, \theta) &\longrightarrow (\sigma_1, \theta_1) \\ (\sigma_0, \theta_0) &\neq (\sigma_1, \theta_1) \end{aligned}$$

Suppose we can find $\mathbf{C} \in PC$, and $D \in \mathbf{B}$ such that

$$\mathbf{C}.D = (\phi, \sigma, \gamma, \theta)$$

Then we define $\mathbf{C}_0, \mathbf{C}_1 \in PC$ by

$$\begin{aligned} \mathbf{C}_0 &\triangleq \mathbf{C}\,[D \mapsto (\phi, \sigma_0, \gamma, \theta_0)] \\ \mathbf{C}_1 &\triangleq \mathbf{C}\,[D \mapsto (\phi, \sigma_1, \gamma, \theta_1)] \end{aligned}$$

We have $\mathbf{C}_0 \neq \mathbf{C}_1$, and $(\mathbf{C}, \mathbf{C}_0), (\mathbf{C}, \mathbf{C}_1) \in$ **local-step**(D).

3.6.2 Threads

We chose, in defining the semantics, to represent each box independently. The box-centric view of an execution models a simple implementation of *TCB* on a network of processors. The execution of a box is a sequence of complete procedure calls. If there is a method call during the execution of a procedure call, the box suspends execution until the method call is executed.

A single thread may execute at many boxes. We defined functions $K.\mathbf{C}$ and $W.\mathbf{C}$ for program configuration $\mathbf{C}$ so we can discuss executions in terms of threads.

We use the "is executing for" function $K.\mathbf{C}$ to identify the root action of an execution step. We use this to determine if two steps are from the same or different threads. We use the "is waiting for" function $W.\mathbf{C}$ to find the box that can take the next step for a thread.

For well-founded $\mathbf{C}$, there are no cycles in $K.\mathbf{C}$. This ensures that every nonidle box is executing on behalf of an action. The function $W.\mathbf{C}$ is not necessarily acyclic. In a configuration $\mathbf{C}$ in which $W.\mathbf{C}$ contains a cycle there is deadlock. A thread that is active at a deadlocked box cannot complete its execution. In the next chapter we discuss deadlock further, and consider ways to avoid it.

Chapter 4

Complete executions

4.1 Introduction

In this chapter, we define the *proper* executions and the *complete* executions, and we define the necessary and sufficient conditions for a proper execution to be complete. In the remaining chapters, we give reduction theorems for complete executions.

A proper execution is one that starts with no active threads, and, if it is finite, no conditional step is enabled in the final configuration. A finite proper execution can only be extended by starting a new thread.

The steps for every thread in a proper execution start with the first step — an **action-start** step — for the thread. However, it is not necessarily the case that the last step of the thread is in the execution. If a thread has an infinite number of steps, then every step has a successor, and so none of the steps can be the last step. Also, a thread may take a finite number of steps and reach a configuration where no step for the thread is enabled, and no step is enabled for any other thread.

The complete executions are those that contain the first and last step — and thus all the steps — for every thread.

We consider ways that a thread may fail to terminate. The first is *deadlock*,

where the thread is permanently disabled, and can take no more steps. The second is *nontermination*, where the thread takes an infinite number of steps. We show that, for a proper execution with a finite number of threads, avoiding deadlock and infinite threads gives a complete execution. For executions with an infinite number of threads, we need a third condition. This condition, *thread fairness* ensures that, if a step is enabled for a thread, then a step is eventually taken for the thread. The *complete execution theorem* says that the proper executions that are deadlock-free, contain no infinite threads, and are thread fair are exactly the complete executions.

The remainder of the chapter investigates requirements on a system implementing a *TCB* program to ensure that there is no deadlock, that every thread terminates, and that it is thread fair. To avoid deadlock and nonterminating threads, we place conditions on the program code and on which pairs of actions can have concurrent threads. We use *control relations* to express the restrictions on concurrency. We show how the control relation and the program code together determine if a program is deadlock-free, and if it allows only finite threads. The requirements for thread fairness are conditions on the run-time system.

Given a system implementing a *TCB* program, we can obtain proper executions from the implemented system by starting it from a quiescent configuration, and only stopping the system if there is no step enabled for a currently executing thread. This essentially means we let the system run "long enough" that each enabled step is taken. If, in addition, we ensure that this system avoids deadlock and nontermination, and is thread fair, then running the system "long enough" is guaranteed to give complete executions.

The material in this chapter can thus be regarded as guidance for implementing Seuss systems so that all executions are complete.

4.1.1 Procedure sets

For some of this work, we consider the program as a set of procedures. The identity of the boxes containing the actions is of secondary importance, so we use the following sets containing the actions, methods, and procedures from the program, and the subsets of partial and total elements of each type.

Definition 4.1

$$\begin{aligned}
\mathbf{A} &\triangleq \{ D.p \mid D \in \mathbf{B} \wedge p \in \mathit{Actions}(D) \} \\
\mathbf{A}_p &\triangleq \{ D.p \mid D \in \mathbf{B} \wedge p \in \mathit{PartActs}(D) \} \\
\mathbf{A}_t &\triangleq \{ D.p \mid D \in \mathbf{B} \wedge p \in \mathit{TotActs}(D) \} \\
\mathbf{M} &\triangleq \{ D.p \mid D \in \mathbf{B} \wedge p \in \mathit{Methods}(D) \} \\
\mathbf{M}_p &\triangleq \{ D.p \mid D \in \mathbf{B} \wedge p \in \mathit{PartMeths}(D) \} \\
\mathbf{M}_t &\triangleq \{ D.p \mid D \in \mathbf{B} \wedge p \in \mathit{TotMeths}(D) \} \\
\mathbf{P} &\triangleq \mathbf{A} \cup \mathbf{M} \\
\mathbf{P}_p &\triangleq \mathbf{A}_p \cup \mathbf{M}_p \\
\mathbf{P}_t &\triangleq \mathbf{A}_t \cup \mathbf{M}_t
\end{aligned}$$

We use α, μ, and π for typical members of **A**, **M**, *and* **P**, *respectively.*

We use the following function when we need to refer to the box of a procedure.

Definition 4.2 *For $\pi \in \mathbf{P}$,*

$$\mathit{Box}(\pi) \triangleq D \quad \text{if } \langle \exists\, p \;::\; \pi = D.p \rangle$$

We define an equivalence relation on **P**.

Definition 4.3 *For* $\pi, \pi' \in \mathbf{P}$,

$$\pi \boxminus \pi' \quad \triangleq \quad Box(\pi) = Box(\pi')$$

If $\pi \boxminus \pi'$, *we say* π box-equals π'.

4.2 Proper executions and complete executions

A proper execution is one that starts in a quiescent configuration, and can only be extended by starting another thread. It is a *maximal* execution for the threads that have started in it.

Definition 4.4 *For* $\varepsilon \in Z$,

$$\begin{array}{rcl} Maximal(\varepsilon) & \triangleq & |\varepsilon| = \infty \ \vee \\ & & (\ |\varepsilon| < \infty \ \wedge \\ & & \quad \langle \forall\ D \ : \ D \in \mathbf{B} \ : \ Enabled(Final(\varepsilon), D) = \bot \ \rangle\ ) \\ Proper(\varepsilon) & \triangleq & qt(Start(\varepsilon)) \ \wedge \ Maximal(\varepsilon) \end{array}$$

For X *a set of executions,*

$$Proper(X) \quad \triangleq \quad \{\, \varepsilon \mid \varepsilon \in X \ \wedge \ Proper(\varepsilon) \,\}$$

To define complete executions, we note that each thread starts and ends with a step at the same box. The following functions return, for a given execution and box, the number of steps in the execution that start a thread at the box, and the number of steps that end a thread at the box.

Definition 4.5 *For $\varepsilon \in Z$, $\mathbf{L} \in Lab$, and $D \in \mathbf{B}$,*

$$\begin{aligned}
NumSteps(\varepsilon, \mathbf{L}) &\triangleq \langle \# \, i \; : \; 0 \leq i < |\varepsilon| \; : \; \varepsilon\langle i \rangle = \mathbf{L} \; \rangle \\
NumStarts(\varepsilon, D) &\triangleq NumSteps(\varepsilon, \textsf{\textbf{action-start}}(D)) \\
NumEnds(\varepsilon, D) &\triangleq NumSteps(\varepsilon, \textsf{\textbf{action-end}}(D)) \; + \\
& \; NumSteps(\varepsilon, \textsf{\textbf{action-reject}}(D))
\end{aligned}$$

If every thread that starts at box D terminates, then the number of start steps equals the number of end steps. Conversely, if a thread does not terminate, its box has more start steps than it has end steps.

Definition 4.6 *For $\varepsilon \in Z$,*

$$\begin{aligned}
Complete(\varepsilon) \;\; \triangleq \;\; & Proper(\varepsilon) \; \wedge \\
& \langle \forall \; D \; : \; D \in \mathbf{B} \; : \; NumStarts(\varepsilon, D) \; = \; NumEnds(\varepsilon, D) \; \rangle
\end{aligned}$$

If X is a set of executions,

$$Complete(X) \;\; \triangleq \;\; \{ \; \varepsilon \; | \; \varepsilon \in X \; \wedge \; Complete(\varepsilon) \, \}$$

The following theorem states that in a complete execution, no box is permanently nonidle. The proof of the theorem is in Appendix B.

Theorem 4.7

$$\begin{array}{l} \langle \forall\ \varepsilon \\ \quad :\ \varepsilon \in \mathit{Complete}(Z) \\ \quad :\ \langle \forall\ D, i \\ \qquad :\ D \in \mathbf{B}\ \wedge\ 0 \leq i \leq |\varepsilon| \\ \qquad :\ \langle \exists\, j\ \ :\ \ i \leq j \leq |\varepsilon|\ \ :\ \ \varepsilon[j].D.\phi = \text{IDLE}\ \rangle \\ \qquad \rangle \\ \quad \rangle \end{array}$$

The following theorem gives a simple characterization of complete finite executions. The proof is in Appendix B.

Theorem 4.8

$$\langle \forall\, \varepsilon\ \ :\ \ \varepsilon \in Z\ \wedge\ |\varepsilon| < \infty\ \ :\ \ \mathit{Complete}(\varepsilon)\ \equiv\ qt(\mathit{Start}(\varepsilon))\ \wedge\ qt(\mathit{Final}(\varepsilon))\ \rangle$$

The number of threads in an execution is the number of start steps across all the boxes.

Definition 4.9 *For $\varepsilon \in Z$,*

$$\mathit{NumThreads}(\varepsilon)\ \triangleq\ \langle \Sigma\, D\ \ :\ \ D \in \mathbf{B}\ \ :\ \ \mathit{NumStarts}(\varepsilon, D)\ \rangle$$

If $\mathit{NumThreads}(\varepsilon)$ is finite for some ε, but ε is infinite, then one of the threads in ε has an infinite number of steps. None of these can be an end step, because a thead takes no steps after its end step. Thus the execution is not complete. A complete execution with a finite number of threads is thus a finite execution. So we get the following corollary to Theorem 4.8.

```
box D
  action  a  ::  E.n
  method m  ::  (*···*)
end

box E
  action  b  ::  D.m
  method n  ::  (*···*)
end
```

Figure 4.1: Program that can deadlock

Corollary 4.10

$$\langle\forall\ \varepsilon$$
$$:\ \varepsilon \in Z\ \wedge\ \mathit{NumThreads}(\varepsilon) < \infty$$
$$:\ \mathit{Complete}(\varepsilon)\ \equiv\ \mathit{Proper}(\varepsilon)\ \wedge\ |\varepsilon| < \infty\ \wedge\ \mathit{qt}(\mathit{Final}(\varepsilon))$$
$$\rangle$$

A proper execution that has a finite number of threads, and is not complete, either has an infinite thread, or it ends in a nonquiescent configuration in which no step is enabled.

We discuss these possibilities in the following sections. We first discuss configurations in which no conditional step is enabled, and then we consider infinite threads.

4.3 Deadlock

Consider the program in Figure 4.1. This program contains two actions, $D.a$, which calls method $E.n$, and $E.b$, which calls method $D.m$. Now consider the following execution. Starting from a quiescent configuration, the first step starts a thread for

$D.a$, and the next step starts a thread for $E.b$. Both threads execute concurrently. When the execution of $D.a$'s thread reaches the call to $E.m$, an entry for this call is placed at the back of E's call queue. Box D waits until this call completes. When the execution of $E.b$'s thread reaches the call to $D.n$, an entry for this call is placed at the back of D's call queue. Box E waits until this call completes. Let $\mathbf{C}$ be the configuration where both boxes are waiting. By Theorem 3.50, no conditional step is enabled for either box. We have $W.\mathbf{C}.D = E$, and $W.\mathbf{C}.E = D$. There is a cycle in $W.\mathbf{C}$, so box D is waiting for box E, which is waiting for D, so ultimately, D is waiting for itself to take a step, which is not possible. Similarly, box E is waiting for itself. We say that the boxes are *deadlocked*. We define deadlock using $W.\mathbf{C}$.

Definition 4.11 *For* $\mathbf{C} \in PC$*, a sequence* $T \in \mathbf{B}^+$ *is a* knot *in* $\mathbf{C}$ *if*

$$\langle \forall\, i \;:\; 0 \leq i < |T| \;:\; W.\mathbf{C}.(T[i]) = T[i \oplus 1] \;\rangle$$

where $\oplus$ *is addition modulo* $|T|$*. For* $D \in \mathbf{B}$,

$$\begin{aligned} dl(\mathbf{C}, D) \;&\triangleq\; \langle \exists\, T, n \;:\; T \text{ is a knot in } \mathbf{C} \wedge n \geq 0 \;:\; (W.\mathbf{C})^n.D \in T \;\rangle \\ dl(\mathbf{C}) \;&\triangleq\; \langle \exists\, D \;:\; D \in \mathbf{B} \;:\; dl(\mathbf{C}, D) \;\rangle \\ dl(\varepsilon) \;&\triangleq\; \langle \exists\, i \;:\; 0 \leq i \leq |\varepsilon| \;:\; dl(\varepsilon[i]) \;\rangle \end{aligned}$$

If $dl(\mathbf{C}, D)$*, we say that* D *is* deadlocked in $\mathbf{C}$*; if* $dl(\mathbf{C})$ *or* $dl(\varepsilon)$*, we say that* $\mathbf{C}$ *or* ε *is* deadlocked.

A knot is a cycle of boxes, each of which is waiting for the next, and so no conditional step is enabled for any box in the knot. A box is deadlocked in a configuration if it is in a knot, or (transitively) waiting for a box in a knot. The following theorem states that deadlock is a *stable* property. That is, if a box is deadlocked at some configuration in an execution, then it is deadlocked at every configuration from that configuration on. The proof is in Appendix B.

Theorem 4.12

$$\begin{array}{l}\langle\forall\ \varepsilon, k, D \\ \quad :\ \varepsilon \in Z\ \wedge\ 0 \le k \le |\varepsilon|\ \wedge\ D \in \mathbf{B}\ \wedge\ dl(\varepsilon[k], D) \\ \quad :\ \langle\forall\, i\ :\ k \le i \le |\varepsilon|\ :\ dl(\varepsilon[i], D)\ \rangle \\ \quad \rangle\end{array}$$

The following corollary states that a deadlocked execution is not complete. Threads that are deadlocked cannot complete execution.

Corollary 4.13

$$\langle\forall\, \varepsilon\ :\ \varepsilon \in Z\ \wedge\ dl(\varepsilon)\ :\ \neg \mathit{Complete}(\varepsilon)\ \rangle$$

Proof

Assume $\varepsilon \in Z$ and $dl(\varepsilon)$. By Definition 4.11, we can choose i, such that $0 \le i \le |\varepsilon|$, and $dl(\varepsilon[i])$. We have

$$\begin{array}{ll} & \mathit{true} \\ \equiv & \quad \{\ \text{assumption}\ \} \\ & dl(\varepsilon[i]) \\ \equiv & \quad \{\ \text{Definition 4.11}\ \} \\ & \langle\exists\, D\ ::\ dl(\varepsilon[i], D)\ \rangle \\ \Rightarrow & \quad \{\ \text{Theorem 4.12}\ \} \\ & \langle\exists\, D\ ::\ \langle\forall\, j\ :\ i \le j < |\varepsilon|\ :\ dl(\varepsilon[j], D)\ \rangle\ \rangle \\ \Rightarrow & \quad \{\ \text{deadlocked box is waiting}\ \} \\ & \langle\exists\, D\ ::\ \langle\forall\, j\ :\ i \le j < |\varepsilon|\ :\ \varepsilon[j].D.\phi \in \{\text{PWAIT}, \text{WAIT}\}\ \rangle\ \rangle \\ \Rightarrow & \quad \{\ \text{weakening}\ \} \\ & \langle\exists\, D\ ::\ \langle\forall\, j\ :\ i \le j < |\varepsilon|\ :\ \varepsilon[j].D.\phi \ne \text{IDLE}\ \rangle\ \rangle \\ \equiv & \quad \{\ \text{predicate calculus}\ \} \end{array}$$

$$\neg\langle \forall\, D \ :: \ \langle \exists\, j \ : \ i \leq j < |\varepsilon| \ : \ \varepsilon[j].D.\phi = \text{IDLE} \ \rangle \ \rangle$$

$\Rightarrow$ { Theorem 4.7 }

$$\neg Complete(\varepsilon)$$

(End of proof)

The final theorem in this section says that if there is no conditional step enabled in a configuration, then every box is either quiescent or deadlocked. The proof is in Appendix B.

Theorem 4.14

$$\begin{array}{l} \langle \forall\ \mathbf{C} \\ \ :\ \mathbf{C} \in PC \\ \ :\ \langle \forall\, D \ :: \ Enabled(\mathbf{C}, D) = \perp \ \rangle \ \equiv \ \langle \forall\, D \ :: \ qt(\mathbf{C}.D) \ \vee \ dl(\mathbf{C}, D) \ \rangle \\ \ \rangle \end{array}$$

4.4 Infinite procedure calls

Consider the program in Figure 4.2. The program contains boxes X and D. Box X has a local integer variable x, an action *inc* that increments the value of x, and a method *dec* that decrements the value of x and returns the decremented value. Box D contains an action a that loops until $X.dec$ returns a nonpositive value. If a thread for action $D.a$ is executed by itself, then the thread is finite. If $X.inc$ executes concurrently with $D.a$, and there are enough threads for $X.inc$ to keep the value of $X.x$ positive, then the thread for $D.a$ loops endlessly, taking an infinite number of steps without terminating.

A thread that takes an infinite number of steps does not contain an end step, because a thread takes no steps after its end step. Thus, for a complete execution, we must ensure that no thread is infinite. We define a predicate that is true of all

```
box X
  var x : integer
  action   inc   ::   x := x + 1
  method dec(out y : integer)
                  ::   x := x − 1 ; y := x
end

box D
  var n : integer init 1
  action a   ::   while n > 0 do X.dec( ; n)
end
```

Figure 4.2: Program with nonterminating procedure

executions containing a infinite thread. First, we identify an execution that contains all the steps from a procedure call.

Definition 4.15 *For* $\varepsilon \in Z$*, and* $D.p \in \mathbf{P}$*, then* ε *is a* full execution *of* $D.p$ *if*

$$\begin{array}{l} Start(\varepsilon).D.\phi = \text{IDLE} \ \wedge \ Proc(\varepsilon[1].D.\gamma) = p \qquad \wedge \\ \langle \forall\, i \ : \ 0 < i < |\varepsilon| \ : \ (\varepsilon[i]).D.\phi \neq \text{IDLE} \ \wedge \ \neg dl(\varepsilon[i], D) \ \rangle \ \wedge \\ (\ |\varepsilon| = \infty \ \vee \ dl(Final(\varepsilon), D) \ \vee \ Final(\varepsilon).D.\phi = \text{IDLE} \ ) \end{array}$$

From this definition, if ε is a full execution for $D.p$, then in the initial configuration, D is idle. In every configuration, except the first and possibly the last, D is not idle. In every configuration, except possibly the last, D is not deadlocked. In the configuration after the first step, D is not idle, and its current call is for p. So the first step is a procedure initialization step for p. Procedure p is the current call for D for the whole of the execution, since D must become idle to clear its current call. The execution is infinite, or it is finite, and in the final configuration D is deadlocked, or it is idle. These cover all the possibilities for an execution of a procedure in a maximal execution.

An infinite thread must include a procedure call with an infinite number of steps. Note that an infinite full execution for a procedure does not necessarily contain an infinite number of steps for the procedure call's thread, since it may contain a finite number of steps for the procedure call's thread, with an infinite number of steps for other threads.

The steps taken by a procedure call are all taken on behalf of the same root action. We define the root action for a step in an execution, using the root actions of the boxes in the configurations. This allows us to indentify pairs of steps in a full execution that are from the same thread.

Definition 4.16 *Let*

$$\begin{aligned}\textsf{\textbf{proc-init}} \;\triangleq\;& \textsf{\textbf{partial-action-init}} \cup \textsf{\textbf{partial-method-init}} \cup \\ & \textsf{\textbf{total-action-init}} \cup \textsf{\textbf{total-method-init}}\end{aligned}$$

For $\varepsilon \in Z$, *and* $0 \leq i < |\varepsilon|$,

$$Root(\varepsilon, i) \;\triangleq\; \begin{cases} \bot & \text{if } \varepsilon\langle i\rangle \subseteq \textsf{\textbf{action-start}} \\ Root(\varepsilon[i+1], D) & \text{if } \varepsilon\langle i\rangle = l(D) \;\wedge\; \varepsilon\langle i\rangle \subseteq \textsf{\textbf{proc-init}} \\ Root(\varepsilon[i], D) & \text{if } \varepsilon\langle i\rangle = l(D, E) \;\vee \\ & \quad (\varepsilon\langle i\rangle = l(D) \;\wedge \\ & \quad\; \varepsilon\langle i\rangle \not\subseteq \textsf{\textbf{proc-init}} \cup \textsf{\textbf{action-start}}) \end{cases}$$

The root action for an **action-start** step is $\bot$, since such a step is never taken on behalf of a currently executing action call. For any other type of step, the root action for the step is the root action for its first locus. For the procedure initialization steps (represented by **proc-init**), we use the root action for the locus in the configuration after the step, since its locus is idle before the step, and executing for the initialized procedure after the step. For every other step, we take the root action for its first locus before the step, since for these steps, the first locus is executing the current

procedure call before the step.

We now define a predicate that is true of a full execution for a procedure π if it contains a finite number of steps for π's thread. Note the the first step of a full execution is always a step initializing the call. So every step with the same root as this step is in π's thread.

Definition 4.17 *For $\pi \in \mathbf{P}$, and $\varepsilon \in Z$, such that ε is a full execution for π,*

$$FiniteThread(\varepsilon) \triangleq \langle \# \, i \; : \; 0 \leq i < |\varepsilon| \; : \; Root(\varepsilon, i) = Root(\varepsilon, 0) \; \rangle \; < \; \infty$$

Definition 4.18 *For $\varepsilon \in Proper(Z)$,*

$$\begin{aligned} inf(\varepsilon) \;\; \triangleq \;\; & \langle \exists \; \alpha, i, j \\ & \; : \alpha \in \mathbf{A} \;\; \wedge \;\; 0 \leq i \leq j \leq \infty \;\; \wedge \\ & \quad \varepsilon\langle i \ldots j \rangle \textit{ is a full execution for } \alpha \\ & \; : \neg FiniteThread(\varepsilon\langle i \ldots j \rangle) \\ & \; \rangle \end{aligned}$$

4.5 Executions with a finite number of threads

We now show that, if we limit our attention to executions containing a finite number of threads, then the complete executions are exactly the proper executions that are not deadlocked and contain no infinite procedure calls.

Theorem 4.19

$$\begin{aligned} & \langle \forall \; \varepsilon \\ & \;\; : \varepsilon \in Z \;\; \wedge \;\; NumThreads(\varepsilon) < \infty \\ & \;\; : Complete(\varepsilon) \;\; \equiv \;\; Proper(\varepsilon) \;\; \wedge \;\; \neg dl(\varepsilon) \;\; \wedge \;\; \neg inf(\varepsilon) \\ & \;\; \rangle \end{aligned}$$

Proof

Assume that $\varepsilon \in Z$, and $NumThreads(\varepsilon) < \infty$. We show

$$Proper(\varepsilon) \;\Rightarrow\; (\; \neg dl(\varepsilon) \wedge \neg inf(\varepsilon) \;\equiv\; |\varepsilon| < \infty \wedge qt(Final(\varepsilon)) \;)$$

The result then follows from this and Corollary 4.10. Assume $Proper(\varepsilon)$. If we have $\neg dl(\varepsilon)$ and $\neg inf(\varepsilon)$, then since the number of threads is finite, and no thread has an infinite number of steps, we have $|\varepsilon| < \infty$. Also, since ε is proper, no step is enabled in its final configuration. Since ε is not deadlocked, by Theorem 4.14, every box is quiescent in the final configuration, so $qt(Final(\varepsilon))$. Conversely, if we have $|\varepsilon| < \infty$ and $qt(Final(\varepsilon))$, then the first gives us that there there are no infinite threads, and the second means that there is no deadlock, since, by Theorem 4.12, a deadlocked execution is deadlocked in its final configuration.
(End of proof)

4.6 Thread fairness

Theorem 4.19 does not apply to executions with an infinite number of threads. To see this, suppose we have actions α and α' that execute on disjoint sets of boxes. Consider an execution that consists of some of the steps of a thread for α, leaving this thread incomplete and with an enabled conditional step, followed by an infinite number of threads for α', each of which has a finite number of steps. There is no infinite thread in this execution, and there is no deadlock, yet it is not complete.

The problem here is that, when there are multiple steps enabled at each configuration in an infinite execution, there is no guarantee that a given enabled step is ever taken. This problem is one that arises whenever we have a system with multiple threads of control, where each step in an execution consists of a nondeterministic choice from among the enabled steps. An execution of such a

system can choose to ignore any given enabled step indefinitely. The standard technique for dealing with this problem is to define a *fairness* condition [13]. From Theorem 3.51, a conditional step remains enabled until it is taken, so the fairness that we require in this case is that every enabled conditional step is eventually taken. We express this in the following way.

Definition 4.20 *For* $\varepsilon \in Z$,

$$
\begin{aligned}
tf(\varepsilon) \triangleq\ & \langle \forall\ D, k \\
& \quad : D \in \mathbf{B} \ \wedge \ 0 \leq k < |\varepsilon| \ \wedge \ Enabled(\varepsilon[k], D) \neq \bot \\
& \quad : \langle \exists\ i \ : \ k \leq i < |\varepsilon| \ : \ \varepsilon\langle i \rangle = Enabled(\varepsilon[k], D) \ \rangle \\
& \quad \rangle
\end{aligned}
$$

If $tf(\varepsilon)$, *we say that* ε *is* thread fair.

Thread fairness for finite executions holds if there is no conditional step enabled in the final configuration.

Theorem 4.21

$$
\begin{aligned}
& \langle \forall\ \varepsilon \\
& \quad : \varepsilon \in Z \ \wedge \ |\varepsilon| < \infty \\
& \quad : tf(\varepsilon) \ \equiv \ \langle \forall\ D \ : \ D \in \mathbf{B} \ : \ Enabled(Final(\varepsilon), D) = \bot \ \rangle \\
& \quad \rangle
\end{aligned}
$$

4.6.1 Fairness for rendezvous procedure calls

The problem of contention for access to a box to execute a procedure call under the rendezvous semantics, which we handled by introducing the call queue, is also an issue that can be dealt with by imposing a fairness condition.

The fairness required for procedure calls is different in nature to thread fairness. A rendezvous method call step does not have the stability property of the

conditional steps in the queue semantics. In an execution under the rendezvous semantics, a method call step for boxes D and E is enabled if box D is in phase ACCEPT, and has a method call to E at the front of its code, and box E is in phase IDLE. The last of these conditions is falsified if E starts a procedure for a thread other than D's.

Once a rendezvous method call step is enabled for D and E, it is either taken, or it becomes disabled, because E takes a different procedure call step. If the procedure call started at E terminates, the step for D and E is again enabled. Otherwise, the procedure call does not terminate, and D is blocked for the rest of the execution.

The executions we want to avoid are one where the method call step is enabled infinitely often but is never taken. In the standard terminology of fairness, executions that have this property are called *strongly fair*, whereas thread fair executions are *weakly fair*.

4.7 The complete execution theorem

We now give an extension of Theorem 4.19 that holds for execution with a finite or infinite number of threads.

Theorem 4.22 (Complete execution theorem)

$$\langle \forall\, \varepsilon \;:\; \varepsilon \in Z \;:\; \mathit{Complete}(\varepsilon) \;\equiv\; \mathit{Proper}(\varepsilon) \;\wedge\; \neg \mathit{dl}(\varepsilon) \;\wedge\; \neg \mathit{inf}(\varepsilon) \;\wedge\; \mathit{tf}(\varepsilon)\; \rangle$$

Proof

Assume $\varepsilon \in Z$. First consider the case that $\mathit{NumThreads}(\varepsilon)$ is finite. If $\mathit{Proper}(\varepsilon)$, then we have

$$\neg \mathit{dl}(\varepsilon) \;\wedge\; \neg \mathit{inf}(\varepsilon)$$

$\equiv$ { Theorem 4.19, Corollary 4.10 }

$$
\begin{aligned}
& |\varepsilon| < \infty \ \wedge \ qt(Final(\varepsilon)) \\
\Rightarrow \quad & \{ \text{ No step enabled in a quiescent configuration, Theorem 4.21 } \} \\
& tf(\varepsilon)
\end{aligned}
$$

So we have $\neg dl(\varepsilon) \ \wedge \ \neg inf(\varepsilon) \ \Rightarrow \ tf(\varepsilon)$. This gives us

$$
\begin{aligned}
& Proper(\varepsilon) \ \wedge \ \neg dl(\varepsilon) \ \wedge \ \neg inf(\varepsilon) \ \wedge \ tf(\varepsilon) \\
\equiv \quad & \{ \text{ above } \} \\
& Proper(\varepsilon) \ \wedge \ \neg dl(\varepsilon) \ \wedge \ \neg inf(\varepsilon) \\
\equiv \quad & \{ \text{ Theorem 4.19 } \} \\
& Complete(\varepsilon)
\end{aligned}
$$

Now consider the case where $NumThreads(\varepsilon)$ is infinite. We prove the equivalence as two implications.

Case $\Rightarrow$:

Assume $Complete(\varepsilon)$. Since $NumThreads(\varepsilon)$ is infinite, so is $|\varepsilon|$, and thus $Proper(\varepsilon)$ follows from Definition 4.6 and Definition 4.4. In a complete execution, every thread reaches its final step and ceases to be active. This means that the thread cannot be deadlocked, nor can it have an infinite number of steps. Also, all conditional steps for every thread are taken, since every thread finishes, so the execution is thread fair.

Case $\Leftarrow$:

Assume $Proper(\varepsilon)$, $\neg dl(\varepsilon)$, $\neg inf(\varepsilon)$, and $tf(\varepsilon)$. For any i, such that $0 \leq i < |\varepsilon|$, we call $\varepsilon\langle i \rangle$ a *start step* if

$$
\varepsilon\langle i \rangle \ \subseteq \ \textbf{action-start} \ \cup \ \textbf{guard-test} \ \cup \ \textbf{total-call}
$$

and we call $\varepsilon\langle i\rangle$ an *end step* if

$$\begin{aligned}\varepsilon\langle i\rangle \quad \subseteq \quad & \textbf{action-end} \ \cup \ \textbf{action-reject} \ \cup \\ & \textbf{test-accept} \ \cup \ \textbf{test-reject} \ \cup \ \textbf{total-return}\end{aligned}$$

A start step begins a procedure call, and an end step completes it. For any i such that $\varepsilon\langle i\rangle$ is a start step, let ε_i to be the segment of ε starting with step i, and ending with the end step that completes the procedure call started by step i, or, if there is no end step corresponding to step i, the infinite suffix of ε starting with step i. We show that ε_i is finite for every i such that $\varepsilon\langle i\rangle$ is a start step, and thus every procedure call, including every action call, terminates. This gives us $\mathit{Complete}(\varepsilon)$. Let D be the unconditional locus of step $\varepsilon\langle i\rangle$, and let

$$V_i \quad = \quad \{\ E \ \mid \ \langle \exists\, j, n \ : \ 0 \le j \le |\varepsilon_i| \ \wedge \ 0 \le n \ : \ (W.(\varepsilon_i[j]))^n.D = E \ \rangle\ \}$$

The set V_i contains D, and every box for which D is waiting during execution ε_i. The boxes in V_i are all the boxes whose execution affects the progress of $\varepsilon\langle i\rangle$'s thread.

Let $\hat{\gamma} = \varepsilon[i+1].D.\gamma$. This is D's call queue after the call started by $\varepsilon\langle i\rangle$ has been added. Execution ε_i is finite iff every call in $\hat{\gamma}$ is eventually removed from D's call queue. We prove that ε_i is finite by induction on $\#V_i$. Note that $\#V_i \le \#\mathbf{B}$ for every ε_i.

For the basis, assume $\#V_i = 1$. Thus $V_i = \{D\}$, and all the calls in $\hat{\gamma}$ do not make any method calls during their execution. We show that the front entry in $\hat{\gamma}$ is eventually removed from D's call queue. Since there are no procedure calls, D is not waiting at any configuration in ε_i, and there is an entry in its call queue throughout the execution (except perhaps at $\mathit{Final}(\varepsilon_i)$), and thus it is not quiescent. By Theorem 3.50, a conditional step for D is enabled at every configuration in ε_i, except the last. Since $\mathit{tf}(\varepsilon)$, there are a finite number of steps from other threads between each step for box D. Since $\neg \mathit{inf}(\varepsilon)$, the call at the front of the queue takes

a finite number of steps to complete. When it does so it is removed from the queue. Iterating this argument, we have that every call in $\hat{\gamma}$ is eventually removed from D's call queue, and thus ε_i is finite.

For the induction step, Suppose $\#V_i = n$, for $n > 0$, and assume that every ε_k is finite, for $\varepsilon\langle k\rangle$ a start step, and $\#V_k < n$, We again show that the front entry in $\hat{\gamma}$ is eventually removed from D's call queue. The argument is similar to that for the basis, except that, in this case, the thread may contain method calls. Suppose that $\varepsilon_i\langle j\rangle$ is a method call step with conditional locus D. That is, it is a method call step from a procedure call executing at D. This is a start step, corresponding to step $\varepsilon\langle i+j\rangle$. Let $k = i + j$. We have $V_k \subseteq V_i$, since every step in the visited set for this call is in the visited set for the call started by $\varepsilon\langle i\rangle$. If $V_k = V_i$, then $D \in V_k$. In this case there is deadlock, because we have $(W.\mathbf{C})^n.D = D$ for some configuration $\mathbf{C}$ in ε_k, and some $n > 0$. Thus, since $\neg dl(\varepsilon)$, we have $\#V_k < \#V_i$, and, by the induction hypothesis, ε_k is finite. The final step of this execution makes a step enabled at box D. Again by thread fairness and the lack of infinite procedure calls, we have that the front entry in D's queue is eventually removed. Iterating this argument, as above, we have that ε_i is finite.
(End of proof)

Theorem 4.22 gives a set of conditions equivalent to $Complete(\varepsilon)$, for any execution ε. A system implementing a *TCB* program is thus complete if and only if every execution of the system is proper, is not deadlocked, contains no infinite threads, and is thread fair. We argued at the beginning of this chapter that $Proper(\varepsilon)$ can be ensured by letting the system run for a sufficiently long time. In the following sections, we discuss ways to ensure that each execution of the system satisfies the other three conditions.

4.8 Control relations

The program in Figure 4.1 can deadlock, and that in Figure 4.2 can have an infinite thread. In the first program, deadlock is possible if threads for $D.a$ and $E.b$ are run concurrently. In the second, a thread for $D.a$ may take an infinite number of steps if it is run concurrently with an infinite number of threads for $X.inc$.

One way to avoid deadlock in an execution of the program in Figure 4.1 is to require that threads for $D.a$ and $E.b$ are not run concurrently. In this case, there can be no deadlock. Similarly, we can avoid an infinite thread in an execution of the program in Figure 4.2 if we require that threads for $D.a$ and $X.inc$ are not run concurrently. So we see that the potential for a program to deadlock, or to have an infinite thread, depends on which pairs of actions we allow to have concurrent threads.

Control relations express restrictions on concurrency. We show how these relations can be used to ensure that executions are deadlock-free, and all threads are finite.

Definition 4.23 *A* control relation *for a program is a symmetric binary relation on* $\mathbf{A}$*. The set of all control relations is* CR*.*

The intended meaning of a control relation $\mathbf{T} \in CR$ is that threads for α and α' may be executed concurrently if and only if $\alpha \ \mathbf{T} \ \alpha'$. Note that Definition 4.23 requires neither reflexivity nor irreflexivity of control relations. If $\alpha \ \mathbf{T} \ \alpha$, then multiple threads for α can be run concurrently; otherwise, at most one thread for α can be active at any time during an execution. However, the notion of "executing concurrently" is rather tenuous for threads for α and α', where $\alpha \boxminus \alpha'$ (which is true if $\alpha = \alpha'$). By design, each box executes a single procedure call at a time. While one action call is executing on a box, all others on the same box wait their turn in the call queue.

For a given control relation, some configurations *respect* the relation, meaning that all pairs of actions active in the configuration are in the control relation, and some do not. An execution respects a control relation if each of its configurations does. To define this, we must identify actions with multiple concurrent threads. The set $\mathit{Actives}(\mathbf{C})$ cannot be used for this purpose, since this does not distinguish actions with one active thread from those with multiple active threads.

Definition 4.24 *For* $\mathbf{C} \in PC$,

$$
\begin{aligned}
\mathit{Multis}(\mathbf{C}) \;\triangleq\; \{\ & D.a \\
\mid\ & D \in \mathbf{B} \ \wedge\ a \in \mathit{Actions}(D) \ \wedge \\
& \langle \#\, i \ :\ 0 \le i < |\mathbf{C}.D.\gamma| \ :\ \mathbf{C}.D.\gamma[i] = (a)\ \rangle > 1 \\
\}\ &
\end{aligned}
$$

Definition 4.25 *For* $\mathbf{T} \in CR$, $\mathbf{A}' \subseteq \mathbf{A}$, $\mathbf{C} \in PC$, *and* $\varepsilon \in Z$,

$$
\begin{aligned}
\mathbf{A}'\ \mathit{resp}\ \mathbf{T} &\triangleq \langle \forall\, \alpha, \alpha' \ :\ \alpha, \alpha' \in \mathbf{A}' \wedge \alpha \neq \alpha' \ :\ \alpha\ \mathbf{T}\ \alpha'\ \rangle \\
\mathbf{C}\ \mathit{resp}\ \mathbf{T} &\triangleq \mathit{Actives}(\mathbf{C})\ \mathit{resp}\ \mathbf{T}\ \wedge\ \langle \forall\, \alpha \ :\ \alpha \in \mathit{Multis}(\mathbf{C}) \ :\ \alpha\ \mathbf{T}\ \alpha\ \rangle \\
\varepsilon\ \mathit{resp}\ \mathbf{T} &\triangleq \langle \forall\, i \ :\ 0 \le i \le |\varepsilon| \ :\ \varepsilon[i]\ \mathit{resp}\ \mathbf{T}\ \rangle
\end{aligned}
$$

If X *resp* $\mathbf{T}$, *we say* X respects $\mathbf{T}$.

An execution that respects $\mathbf{T}$ allows only concurrency permitted by $\mathbf{T}$. The condition on $\mathit{Multis}(\mathbf{C})$ deals with actions having multiple concurrent calls, as discussed above. The relation *resp* is monotonic in the control relation, as the following theorem shows. Larger control relations allow more concurrency, and thus are respected by more configurations or executions.

Theorem 4.26

$$
\begin{array}{l}
\langle\forall\ \mathbf{T},\mathbf{T}' \\
\quad :\ \mathbf{T},\mathbf{T}' \in CR\ \ \wedge\ \ \mathbf{T} \subseteq \mathbf{T}' \\
\quad :\ \langle\forall\ \mathbf{A}'\ :\ \mathbf{A}' \subseteq\ \mathbf{A}\ \ :\ \mathbf{A}'\ \mathit{resp}\ \mathbf{T} \Rightarrow\ \mathbf{A}'\ \mathit{resp}\ \mathbf{T}'\ \rangle\quad \wedge \\
\qquad \langle\forall\ \mathbf{C}\ :\ \mathbf{C} \in PC\ :\ \mathbf{C}\ \mathit{resp}\ \mathbf{T} \Rightarrow\ \mathbf{C}\ \mathit{resp}\ \mathbf{T}'\ \rangle\quad \wedge \\
\qquad \langle\forall\ \ \varepsilon\ \ :\ \ \varepsilon\ \in\ Z\ \ :\ \varepsilon\ \ \mathit{resp}\ \mathbf{T} \Rightarrow\ \ \varepsilon\ \ \mathit{resp}\ \mathbf{T}'\ \rangle \\
\quad \rangle
\end{array}
$$

Proof

From Definition 4.25.

(End of proof)

The simplest control relation is $\emptyset$, the empty relation; this allows at most a single active action call at any point in an execution. An execution that respects this control relation is sequential, that is, an execution in which each action call is started and run to completion before another action call is started. We use the empty control relation to define sequential executions.

Definition 4.27 *For $\varepsilon \in Z$,*

$$\varepsilon\ \mathit{is}\ \text{sequential}\ \ \triangleq\ \ \varepsilon\ \mathit{resp}\ \emptyset$$

4.8.1 Implementing control relations

Control relations are used to define subsets of the set of executions having desirable properties. A suitably chosen control relation guarantees a deadlock-free execution, or an execution with no infinite thread. If threads for actions α and α' can deadlock, we can avoid this possibility by using a control relation that does not contain (α, α'). If threads for α, α', and α'' can deadlock, we use a control relation that does not contain one of (α, α'), (α, α''), and (α', α'').

If control relation $\mathbf{T}$ guarantees a deadlock-free execution, then so does $\mathbf{T}'$, for any $\mathbf{T}' \subseteq \mathbf{T}$, since every execution that respects $\mathbf{T}'$ also respects $\mathbf{T}$. So the intersection of a set of control relations, each of which guarantees a single property of the execution, gives a control relation that guarantees all of the properties.

Control relations thus give uniform way to ensure desirable properties of executions. Part of the implementation of a *TCB* program is a *scheduler*, a program that decides which threads to start, in which order. A thread for action α is started when the scheduler sends a **start**(α) message to α's box. When the action completes, the box sends an **accept**(α) or **reject**(α) message to the scheduler, depending on the outcome of the thread's execution. The processors execute independently, so the scheduler has no control over the order in which threads end.

The scheduler keeps a record of all actions that have active threads from the sequence of start and end messages. Schedulers implement a control relation by ensuring that the set of active actions respects the control relation. For a control relation $\mathbf{T}$, a scheduler sends a **start**(α) message only if $\alpha \mathbf{T} \alpha'$ for every α' with an active thread.

We consider further requirements for a scheduler for *TCB* in Chapter 6. For now, we note that, if we can define a control relation that guarantees a property of the executions of a program, then we can implement the program using a scheduler that respects the control relation, and all executions of the implementation have the desired property.

4.9 Avoiding deadlock

We saw that for the program in Figure 4.1, a control relation can be used to avoid deadlock. Specifically, if $\mathbf{T}$ is a control relation where $(D.a, E.b) \notin \mathbf{T}$, then any execution of the program that respects $\mathbf{T}$ does not deadlock.

We define a predicate on control relations that is true precisely when a control

relation does not allow deadlock in an execution.

Definition 4.28 *For* $\mathbf{T} \in CR$,

$$DF(\mathbf{T}) \triangleq \langle \forall\, \varepsilon \ :\ \varepsilon \in Proper(Z) \wedge \varepsilon \ resp\ \mathbf{T} \ :\ \neg dl(\varepsilon) \ \rangle$$

If $DF(\mathbf{T})$, *we say that* $\mathbf{T}$ *is* deadlock-free.

Deadlock-freedom is antimonotonic in the control relation. A smaller control relation allows less concurrency, which means fewer possibilities for deadlock.

Theorem 4.29

$$\langle \forall\, \mathbf{T}, \mathbf{T}' \ :\ \mathbf{T}, \mathbf{T}' \in CR \ \wedge\ \mathbf{T} \subseteq \mathbf{T}' \ :\ DF(\mathbf{T}) \ \Leftarrow\ DF(\mathbf{T}') \ \rangle$$

From this theorem, we see that is any control relation is deadlock-free, then so is the empty control relation. Equivalently, if the empty control relation is not deadlock-free, then no control relation is deadlock-free. The empty control relation allows only sequential executions, so if it is not deadlock-free, then there is a deadlocked sequential execution.

Let ε be a deadlocked proper execution that respects $\emptyset$, and let $\varepsilon[i]$ be the last nondeadlocked configuration in ε. The start configuration is quiescent, so such an i exists. Thus, $W.(\varepsilon[i])$ is acyclic, and $W.(\varepsilon[i+1])$ is cyclic. Step $\varepsilon\langle i\rangle$ is a method call step to a box that is waiting. Since there is only one executing thread, the source and agent of this call are executing for the same thread. We call this a *cyclic call.* There is no hope of ensuring deadlock-free execution for a program containing an action that allows a cyclic call.

The requirements to ensure deadlock-free execution are first, that the empty relation is deadlock-free. We do not attempt to implement programs for which this is not the case. Second, that the execution respects a deadlock-free control relation.

4.9.1 Nonblocking control relations

Definition 4.28 does not help particularly in deciding *a priori* if a particular control relation allows deadlock. The definition essentially says that a control relation does not allow deadlock if it does not allow deadlock. We define a condition that is sufficient to ensure that a control relation is deadlock-free, and is simpler to check than the conditions of Definition 4.28.

No rule in the semantics creates a statement that is not already present in the code. When the code component for a procedure call is initialized, it is initialized to the body code from the procedure's definition. Thus, for boxes D and E, and configuration $\mathbf{C}$ from a proper execution, if $W.\mathbf{C}.D = E$, then the code for current procedure in D contains a method call statement for a method in E.

We define the following relation on procedures.

Definition 4.30 *For* $\pi \in \mathbf{P}$*, we say that* $\mu \in \mathbf{M}$ appears in π *if there is a call to* μ *in the code of the body of* π*. We write this as* $\pi \succ \mu$*.*

If $\pi \succ \mu$, then there may possibly be a call to μ during an execution of π. Note that if π is partial, $\pi \succ \mu$ if μ appears as a test or in the body of one of π's alternatives. If π is total, $\pi \succ \mu$ if μ appears in the body of π. We have the following theorem, which shows that $\succ$ limits the values of $W.\mathbf{C}$ for every reachable $\mathbf{C}$.

Theorem 4.31

$$\begin{array}{l} \langle \forall\ \varepsilon, i, D, E \\ \quad : \varepsilon \in \mathit{Proper}(Z) \ \wedge \ 0 \le i \le |\varepsilon| \ \wedge \ W.(\varepsilon[i]).D = E \\ \quad : \langle \exists\ \pi, \pi' \ : \ \pi \succ \pi' \ : \ \mathit{Box}(\pi) = D \ \wedge \ \mathit{Box}(\pi') = E \ \rangle \\ \quad \rangle \end{array}$$

Note that if $\pi \succ \mu$, then it is not the case that every call to π includes a call to μ. If the code of π contains the statement "**if** *false* **then** μ", then $\pi \succ \mu$, but no call

to μ is ever made by executing this statement. But, from Theorem 4.31, if there is a call to μ during an execution of π, then $\pi \succ \mu$. This relation gives us an "upper bound" on the method calls that may be made during a procedure call.

If procedure π calls method μ, then $Box(\pi)$ waits for $Box(\mu)$ until the call completes. It must also wait for any procedure call executing at $Box(\mu)$ before the call to μ is started. The following relation gives all the procedures whose execution may block the execution of a call to a given procedure. It is a combination of the "appears in" relation, and box equality.

Definition 4.32

$$bb \quad \triangleq \quad \succ ; \boxminus$$

For $\pi, \pi' \in \mathbf{P}$*, if* $\pi\ bb\ \pi'$ *we say that* π *is* blocked by π'*.*

We define a "blocked-by" graph for a program.

Definition 4.33

$$BB \quad \triangleq \quad (\mathbf{P}, bb)$$

Thus BB is the directed graph with vertex set $\mathbf{P}$, and an edge from π to π' iff $\pi\ bb\ \pi'$.

If there is a cycle in BB, then there is the possibility of reaching deadlock during an execution in which there are no restrictions on concurrency. As we saw above, the possibility for deadlock can be eliminated if use a control relation that limits concurrency.

Suppose we have a subset $\mathbf{A}' \subseteq \mathbf{A}$. We remove from BB all vertices for procedures that are not called by actions in $\mathbf{A}'$, and all edges incident on them. The remaining graph contains all the calls that may be made during a concurrent

execution of the actions in $\mathbf{A}'$. If this restricted graph is acyclic, then there is no possibility for deadlock during such an execution. This suggests that we consider subgraphs of BB according to the sets of actions that can have concurrent threads, as given by a control relation.

We first define the nodes in representing procedures that may be called during the execution of a thread for an action, or a set of actions.

Definition 4.34 *For $\alpha \in \mathbf{A}$, and $\mathbf{A}' \subseteq \mathbf{A}$,*

$$\begin{aligned} Range(\alpha) &\triangleq \{\, \pi \;:\; \pi \in \mathbf{P} \;:\; \alpha \succ^* \pi \,\} \\ Range(\mathbf{A}') &\triangleq \langle \cup\, \alpha \;:\; \alpha \in \mathbf{A}' \;:\; Range(\alpha) \;\rangle \end{aligned}$$

The set $Range(\alpha)$ contains α, and all methods reachable from it by $\succ$. This gives a bound on the procedures that are called during execution of a thread for α.

The theorem we wish to prove is that for any subset $\mathbf{A}' \subseteq \mathbf{A}$, the subgraph of BB on the vertices in $Range(Actives(\mathbf{C}))$ is a bound on $W.\mathbf{C}$ in any configuration $\mathbf{C}$ from a proper execution. Therefore, if the subgraph is acyclic, $\mathbf{C}$ is not deadlocked.

We define the *restriction* of a graph to a subset of its vertices.

Definition 4.35 *For a graph $G = (V, E)$, and $U \subseteq V$,*

$$G \restriction U \;\triangleq\; (U, E') \quad \textit{where } E' = E \cap (U \times U)$$

Using this, we define subgraphs of BB for subsets of the actions.

Definition 4.36

$$BB(\mathbf{A}') \;\triangleq\; BB \restriction Range(\mathbf{A}')$$

Now we have a theorem

Theorem 4.37

$$
\begin{array}{l}
\langle\forall\ \varepsilon, i \\
\quad :\ \varepsilon \in Proper(Z)\ \wedge\ 0 \leq i \leq |\varepsilon| \\
\quad :\ BB(Actives(\varepsilon[i]))\ \mathit{is\ acyclic}\ \Rightarrow\ \neg dl(\varepsilon[i]) \\
\quad \rangle
\end{array}
$$

Now we define a condition on control relations based on the above observations.

Definition 4.38 *For* $\mathbf{T} \in CR$*,*

$$\mathbf{T}\ \mathit{is\ nonblocking}\ \triangleq\ \langle\forall\ \mathbf{A}'\ :\ \mathbf{A}'\ \mathit{resp}\ \mathbf{T}\ :\ BB(\mathbf{A}')\ \mathit{is\ acyclic}\ \rangle$$

A control relation is nonblocking if there are no cycles in the blocked-by graph for any set of actions that can execute concurrently. The following theorem follows immediately from Theorem 4.37 and Defintion 4.38.

Theorem 4.39

$$\langle\forall\ \mathbf{T}\ :\ \mathbf{T} \in CR\ \wedge\ \mathbf{T}\ \mathit{is\ nonblocking}\ :\ DF(\mathbf{T})\ \rangle$$

The theorem says that a nonblocking control relation is deadlock-free, so there is no deadlock in any proper execution.

Note that if BB is acyclic, then so is $BB(\mathbf{A}')$, for any $\mathbf{A}' \subseteq \mathbf{A}$. In this case, any control relation is nonblocking, and so all executions are deadlock-free. One way to ensure that BB is acyclic is to define a partial order on the boxes, and write the code for the procedures such that a call to a method in box E appears in a procedure in box D only if D is before E in the order. Such a program can never deadlock.

To show that a TCB program is implementable without deadlock, it is sufficient to show that $BB(\alpha)$ is acyclic for every $\alpha \in \mathbf{A}$. To guarantee that the execution

of an implemented system does not deadlock, it is sufficient to ensure that the implementation ensures executions that respect a nonblocking control relation.

A nonblocking control relation is not necessary to ensure deadlock-free execution, as the following example shows. Consider a program with methods μ and μ', in different boxes, defined as follows.

$$\begin{array}{lll} \mu(\mathbf{in}\ b : \mathit{boolean}) & :: & \mathbf{if}\ b\ \mathbf{then}\ \mu'(\mathit{false}) \\ \mu'(\mathbf{in}\ b : \mathit{boolean}) & :: & \mathbf{if}\ b\ \mathbf{then}\ \mu(\mathit{false}) \end{array}$$

There is a cycle in relation $\succ$ on these methods, which means there is a possibility for a cyclic call. But deadlock is not possible, since a call to $\mu(\mathit{false})$ makes no method calls, and a call to $\mu(\mathit{true})$ calls $\mu'(\mathit{false})$, which makes no method calls. Suppose we have actions α and α' in different boxes, as follows.

$$\begin{array}{lll} \alpha & :: & \mu(\mathit{false}) \\ \alpha' & :: & \mu'(\mathit{false}) \end{array}$$

Threads for these actions cannot cause deadlock when run concurrently. In fact, they execute on disjoint sets of boxes, so neither blocks the other. From Definition 4.38, if control relation $\mathbf{T}$ is nonblocking, then $(\alpha, \alpha') \notin \mathbf{T}$, so threads for α and α' cannot be executed concurrently under any nonblocking control relation.

For some programs, the approach to avoiding deadlock using the graph BB excludes concurrent execution of actions that cannot cause deadlock. This is because we ignore semantic information — in the above example, the parameter values passed in method calls — and instead use only syntactic information about the possible calls between procedures. For this slight loss of precision, the nonblocking condition reduces the task of determining if a control relation can allow deadlock from a check of all possible executions to a static check of the program code.

4.10 Avoiding infinite threads

As we saw in Section 4.4, for the program in Figure 4.2, there are no infinite threads if the program is run under a control relation $\mathbf{T}$ such that $(D.a, X.inc) \notin \mathbf{T}$. We now define a condition on control relations that ensures terminating threads.

Definition 4.40 *For* $\pi \in \mathbf{P}$, $\mathbf{P}' \subseteq \mathbf{P}$, *and* $\mathbf{T} \in CR$,

$$
\begin{array}{rcl}
term(\pi, \mathbf{T}) & \triangleq & \langle \forall\ \varepsilon \\
& & \quad : \varepsilon \in Z \ \wedge\ \varepsilon \ resp\ \mathbf{T} \ \wedge\ \varepsilon \textit{ is a full execution for } \pi \\
& & \quad : \langle \#\, i \ :\ 0 \leq i < |\varepsilon| \ :\ Root(\varepsilon, i) = Root(\varepsilon, 0)\ \rangle < \infty \\
& & \quad \rangle \\
term(\mathbf{P}', \mathbf{T}) & \triangleq & \langle \forall\ \pi \ :\ \pi \in \mathbf{P}' \ :\ term(\pi, \mathbf{T})\ \rangle
\end{array}
$$

If $term(\mathbf{A}, \mathbf{T})$, then all threads in an execution respecting $\mathbf{T}$ have a finite number of steps. As with Definition 4.28, this definition does not say much more than that the control relation and the program do not allow infinite procedure calls.

Showing termination of sequential code is a problem outside the scope of this work. The language *TCB* is certainly capable of expressing programs with nonterminating actions, and, since the Halting Problem is insoluble (see [30]), there is no algorithm that can detect such programs.

The techniques used for proving termination for a sequential program (see [15], for example), generally require that, for each action, we construct a function (called a *measure*) that maps the program configuration to some well-founded set, and show that each step taken by a thread for the action decreases the measure. Since there is no infinite decreasing sequence of values from a well-founded set, this ensures that the each thread is terminating.

Proving termination of procedure calls executing concurrently with other threads is a more complicated problem than proving termination for a sequential

program. Again, we do not address this issue further here. We assume that the program has been certified, by some means, to allow only terminating threads.

4.11 Implementing thread fairness

Consider an implementation model for *TCB* programs, where each box is implemented by a separate computing unit (a single processor, or a process on a processor). The thread-fairness condition says that if a step is enabled for one of these units, then that step is eventually taken. This corresponds to the reasonable assumption that the units are sufficiently independent that each takes a step from time to time, if it is able. The thread fairness condition is, for many implementations, guaranteed by the nature of the implementation. We assume that the run-time system of any implementation guarantees thread fairness. Again, we do not address this issue in greater detail here.

On the question of strong fairness for rendezvous procedure calls, let ε be a thread fair execution under the queue semantics. Between any procedure call step in ε and its corresponding procedure, there are a finite number of configurations where the agent for the call is idle. If there is no corresponding procedure initialization step in the execution, then there is an infinite prefix where the agent is not idle.

We claim that execution ε is equivalent to an execution where all procedure call steps are removed, and the procedure initialization steps are replace with rendezvous procedure call steps. In this case, no rendezvous call step is enabled in an infinite number of configurations but never taken. So weak thread fairness, together with the call queues, provides an implementation of strong fairness for procedure calls.

We return to this transformation of queue semantics execution to rendezvous semantics executions in Chapter 5.

4.12 Summary

We have identified complete executions as the desirable executions for a *TCB* program. Theorem 4.22 gives gives exactly the conditions needed to ensure that an execution be complete. It must be proper, deadlock-free, contain no infinite threads, and be thread fair.

We showed how control relations, and restrictions on the program code guarantee that proper executions are deadlock-free, and there are no infinite threads. We argued that the run-time system can ensure a proper execution that is thread fair. Together, these give us a way to implement programs so that all executions are complete.

4.12.1 Avoiding run-time errors

As noted in Section 2.5, we do not handle run-time errors in the semantics we have presented. In Section 2.12 we show a way to extend the semantics to deal with this issue. In this extension, there is a new phase, FAIL, and a run-time error at a box causes it to enter this phase, and remain in it for the remainder of the execution. An execution with a run-time error is not complete. Avoiding run-time errors is an issue for the sequential language and the expression language, and we do not address it further here. We assume that programs are certified, by some means, to avoid run-time errors.

Chapter 5

Reduction

5.1 Introduction

The last chapter outlined sufficient conditions on the implementation of a *TCB* program that ensure that every execution is complete. The conditions include restrictions on concurrency, expressed using control relations. In this chapter we show a further restriction on concurrency that ensures that for every complete execution ε there is a sequential execution ε' such that ε and ε' have closely related behaviour. Again, we use a control relation to express the necessary restriction on concurrency.

With the results from the last chapter, this result gives us a way to implement a *TCB* program such that the behaviours of all concurrent executions of the system can be deduced by considering just the sequential executions.

To show the existence of a sequential execution for any complete ε respecting the control relation by showing a way to rearrange the steps of ε so that the steps from each thread are contiguous. The rearrangement consists of a sequence of local transformations, such as reversing the order of two consecutive steps, or replacing a sequence of steps by a single step, or removing a step entirely.

We define a *reduction relation* between executions, written $\leadsto$. We show

that a complete execution ε and its sequential rearrangement ε' formed by the above procedure satisfy $\varepsilon \rightsquigarrow \varepsilon'$. We also show, from the definition of $\rightsquigarrow$, that this guarantees that the behaviours of ε and ε' are closely related, and that we can deduce the behaviour of ε from the behaviour of ε'.

5.1.1 Right-movers and left-movers in *TCB*

In the two-phase locking protocol, a single action consists of a sequence of resource acquisition steps, followed by a single update step, and a sequence of resource release steps. We saw that acquisition steps can be delayed, or "moved right", and release steps can be advanced, or "moved left". Steps that acquire resources are call *right-movers*, and steps that release resources are called *left-movers*.

In *TCB*, there are two type of resources controlling the execution of a thread. Before a thread can execute at a box it must first place an entry in the call queue, and then this call must reach the front of the queue and be initialized by the box. The resources required for a thread to execute at a box are the queue and the box itself. A thread that has an entry in a call queue holds the queue resource. This is a shared resource, since there can be multiple entries in the queue. A thread that is executing at a box holds the box resource. This is an exclusive resource, since only one thread at a time can execute at a box.

We identify a single step in each thread called the *decision step*. This is the step that first puts a box executing for the thread in phase ACCEPT or REJECT. Once a box enters this phase, the outcome of the thread, in terms of acceptance or rejection, is determined. We use this step to mark the position in the execution about which we coalesce the steps of the thread. We call the decision step an *accept decision step* if it is from an accepting thread, and a *reject decision step* if it is from a rejecting thread.

Every step, other than the decision step, is classified as a step that acquires

a resource, a step that releases a resource, or a step that computes locally, neither acquiring or releasing a resource. The first group are the right-movers, and the other two are the left-movers.

In every thread all steps before the decision step are right-movers. Threads do not, however, have a two-phase structure, since there can be right-movers — resource acquisition steps — after the decision step. These are the steps for a total method call. A **total-call** step acquires a queue resource, and a **t-method-init** step acquires a box resource. Both these steps appear after the decision step in a thread.

5.1.2 Transforming an execution

To reduce a thread to a sequence of contiguous steps we apply a sequence of local transformations to an execution. Each transformation replaces a finite sequence of steps with a different sequence, such that the replacement has the same starting and ending configuration. The transformations are *swapping* two adjacent steps, *replacing* a sequence of steps with a single step, and *removing* a step.

The swap transformations allow a right-mover to be swapped with a step from a different thread that is immediately to its right in the execution. The right-mover moves right over the other step. Similarly, a left-mover moves left over a step from a different thread.

We define two new types of step: rendezvous call steps, and steps representing a complete procedure call. We show that show that a pair of consecutive steps consisting of a procedure call step followed by a procedure initialization step for the same thread can be replaced by a single rendezvous call step, and we show that a contiguous sequence of steps from the same thread representing the complete execution of a procedure call can be replaced by a single step representing the whole call.

We call the steps representing complete procedure calls *atomic* steps. The

atomic steps for an action represent a complete execution of a call to that action, that is, a complete thread. In reducing the execution, our final aim is an execution where all threads have been reduced to a single atomic step.

The final type of transformation is the removal of an atomic step representing a rejecting thread. We have defined *TCB* such that a rejecting thread does not change the persistent state of a program, so this transformation always gives a valid execution.

The material in this chapter is organized as follows. Section 5.2 defines the rendezvous call steps and the atomic steps. We define some new sets of executions using these steps. Section 5.3 classifies the steps as decision steps, right-movers, and left-movers. Section 5.4 defines a *reduction relation* on executions in terms of the sequence of accept decision steps. For executions ε and ε', we write $\varepsilon \rightsquigarrow \varepsilon'$ if ε' is the execution formed from ε by moving some right-movers right, and moving some left movers left. We show that there is a close correspondence between the configuration in ε and ε' if $\varepsilon \rightsquigarrow \varepsilon'$. Section 5.6 defines the transformations described above, and shows that they can all be applied an any execution, with the exception of swapping an atomic step for a total method call with another step. Section 5.7 defines a control relation that limits executions to ones in which we can swap atomic total method calls, as required for the reduction. Section 5.8 states the first reduction theorem, and gives its proof.

5.2 Compound steps

We define two new sets of steps that are used during reduction to replace sequences of steps. The first set of steps are those defined by the procedure call rules in the rendezvous semantics, that is rules **p-action-start-rdv**, **p-action-start-rdv**, **guard-test-rdv**, and **total-call-rdv**. The second set of steps are the atomic steps, which represent the complete execution of a procedure call. We define some new sets of

Configuration	D	E
$\varepsilon[i]$	$(\text{ACCEPT}, \sigma_0, \gamma_0, \theta_0)$	$(\text{IDLE}, \sigma_1, \perp)$
$\varepsilon[i+1]$	$(\text{WAIT}, \sigma_0, \gamma_0, \theta_0)$	$(\text{IDLE}, \sigma_1, (m, D, \tilde{v}))$
$\varepsilon[i+2]$	$(\text{WAIT}, \sigma_0, \gamma_0, \theta_0)$	$(\text{ACCEPT}, \sigma_1', (m, D), \theta')$

Table 5.1: Configurations for a method call from D to E

executions using these labels.

5.2.1 Rendezvous calls

Suppose $\varepsilon \in Z$ contains the following pair of steps for some i, D and E.

$$\begin{aligned} \varepsilon\langle i\rangle &= \textbf{total-call}(D, E) \\ \varepsilon\langle i+1\rangle &= \textbf{t-method-init}(E) \end{aligned}$$

Suppose further that $Root(\varepsilon, i) = Root(\varepsilon, i+1)$. From the definition of the $Root$ function, we can deduce that D and E are in the same member of $Stacks(\varepsilon[i+2])$. Since, in configuration $\varepsilon[i+1]$, E is not in a call stack, and D is the top element in its stack, and, in $\varepsilon[i+2]$, E is in a call stack, $Kt.(\varepsilon[i+2]).E = D$. Thus the front entry in E's call queue has source D. Since E's configuration is well-formed, there is only one entry in the queue with D as caller. So the entry added to the queue by step $\varepsilon\langle i\rangle$ must be the front entry in the queue in $\varepsilon[i+2]$. We thus have that $\varepsilon[i].E.\phi = \perp$. From this, the definition of the rules, and the well-formedness of the configurations, we can deduce the values of the configurations for D and E for this segment of the execution. The results are show in Table 5.1. In these equations we write (m, D) for the singleton list $\langle(m, D)\rangle$. We now note that configurations $\varepsilon[i]$ and $\varepsilon[i+2]$ satisfy the conditions for rule **(total-call-rdv)** from the rendezvous semantics (using the convention for singleton lists mentioned above).

The rule **(total-call-rdv)** requires that E be quiescent before the step. Also, a **total-call**(D, E) step followed by a **t-method-init**(E) step are steps from the same thread only if E is quiescent before the first step.

The above argument can be repeated for the other three procedure call rules in the rendezvous semantics. Each corresponds to two steps from the queue semantics. We use Definition 3.35 to define the following relations.

p-action-start-rdv

t-action-start-rdv

guard-test-rdv

total-call-rdv

We use Definition 3.38 to define the following labels, for $D, E \in \mathbf{B}$.

p-action-start-rdv(D)

t-action-start-rdv(D)

guard-test-rdv(D, E)

total-call-rdv(D, E)

We define the set of configurations in which a box is quiescent.

Definition 5.1 *For* $D \in \mathbf{B}$,

$$PC_q(D) \triangleq \{ \mathbf{C} \mid \mathbf{C} \in PC \wedge qt(\mathbf{C}.D) \}$$

We now define an relation equal to each of the new relations. We use the above subsets of PC to restrict the before configurations for these relations to those where the agent is quiescent.

Theorem 5.2

$$
\begin{array}{l}
\langle \forall\ D \\
\quad : D \in \mathbf{B} \\
\quad : \textsf{\textbf{t-action-start-rdv}}(D) \ = \\
\quad\ \ \textsf{\textbf{action-start}}(D); \textsf{\textbf{t-action-init}}(D) \ \cap \ (PC_q(D) \times PC) \ \wedge \\
\quad\ \ \textsf{\textbf{p-action-start-rdv}}(D) \ = \\
\quad\ \ \textsf{\textbf{action-start}}(D); \textsf{\textbf{p-action-init}}(D) \ \cap \ (PC_q(D) \times PC) \\
\quad \rangle \\
\langle \forall\ D, E \\
\quad : D, E \in \mathbf{B} \\
\quad : \textsf{\textbf{guard-test-rdv}}(D, E) \ = \\
\quad\ \ \textsf{\textbf{guard-test}}(D, E); \textsf{\textbf{p-method-init}}(E) \ \cap \ (PC_q(E) \times PC) \ \wedge \\
\quad\ \ \textsf{\textbf{total-call-rdv}}(D, E) \ = \\
\quad\ \ \textsf{\textbf{total-call}}(D, E); \textsf{\textbf{t-method-init}}(E) \ \cap \ (PC_q(E) \times PC) \\
\quad \rangle
\end{array}
$$

Proof

Use the definitions of the appropriate rules.

(End of proof)

5.2.2 Atomic steps

We define *atomic steps* to represent the complete execution of a call to a procedure. First we define an execution that contains all the steps of a procedure call, and no steps from other threads.

Definition 5.3 *For $\varepsilon \in Z$, $D \in \mathbf{B}$, and $p \in Procs(D)$,*

$$\varepsilon \textit{ is a } \text{compact execution} \textit{ of } D.p \;\triangleq\; F(\varepsilon, D) \;\wedge\; G(\varepsilon) \;\wedge\; Proc(\varepsilon[1].D.\gamma) = p$$

where

$$\begin{aligned} F(\varepsilon, D) \;&\triangleq\; |\varepsilon| < \infty \;\wedge\; qt(Start(\varepsilon).D) \;\wedge\; qt(Final(\varepsilon).D) \;\wedge \\ &\quad \neg qt(\varepsilon[1].D) \;\wedge\; \langle \forall\, i \;:\; 1 < i < |\varepsilon| \;:\; \varepsilon[i].D.\phi \neq \text{IDLE} \;\rangle \\ G(\varepsilon) \;&\triangleq\; (\; Root(\varepsilon, 0) = Root(\varepsilon, 1) \;\vee\; Root(\varepsilon, 0) = \bot \;) \;\wedge \\ &\quad Root(\varepsilon, 1) \neq \bot \;\wedge \\ &\quad \langle \forall\, i \;:\; 1 < i < |\varepsilon| \;:\; Root(\varepsilon, i) = Root(\varepsilon, 1) \;\rangle \end{aligned}$$

The definition of a compact execution has three parts. The first part, predicate $F(\varepsilon, D)$, requires that ε be finite, that box D be quiescent in the start and final configurations, that it be nonquiescent after the first step, and that it be nonidle after the second step. The intention is that the first step places an entry in D's call queue, making it nonquiescent, and the second step initializes this call, making D nonidle. Box D is nonidle until the final step, which leaves it quiescent. The segment of ε from step 1 to the end is a terminating full execution of a procedure call at D. The second part, predicate $G(\varepsilon)$, requires that every step in ε have the same root action, with the possible exception of the first step. We allow the first step to be an **action-start** step, and this has root action $\bot$. The third part, $Proc(\varepsilon[1].D.\gamma) = p$, requires that the first step of ε be one that puts a call for procedure p in D's call queue. Together these conditions limit an execution to one that contains a single execution of a call to $D.p$, and no steps from other threads.

We now define five types of atomic step. There are steps for an accepting call to an action, or to a partial method, steps for rejecting calls to each, and a step for a call to a total method.

Definition 5.4 *For $D.a \in \mathbf{A}$,*

$$
\begin{aligned}
\textsf{\textbf{accept}}(D.a) \;\triangleq\; \{ \; & (\mathbf{C}, \mathbf{C}') \\
& \mid \langle \exists \; \varepsilon \\
& \qquad : \varepsilon \in Z \;\; \wedge \;\; \varepsilon \textit{ is a compact execution of } D.a \\
& \qquad : \textit{Start}(\varepsilon) = \mathbf{C} \;\; \wedge \;\; \textit{Final}(\varepsilon) = \mathbf{C}' \;\; \wedge \\
& \qquad \;\; \textit{Last}(\varepsilon) = \textsf{\textbf{action-end}}(D) \\
& \qquad \rangle \\
& \}
\end{aligned}
$$

$$
\begin{aligned}
\textsf{\textbf{reject}}(D.a) \;\triangleq\; \{ \; & (\mathbf{C}, \mathbf{C}') \\
& \mid \langle \exists \; \varepsilon \\
& \qquad : \varepsilon \in Z \;\; \wedge \;\; \varepsilon \textit{ is a compact execution of } D.a \\
& \qquad : \textit{Start}(\varepsilon) = \mathbf{C} \;\; \wedge \;\; \textit{Final}(\varepsilon) = \mathbf{C}' \;\; \wedge \\
& \qquad \;\; \textit{Last}(\varepsilon) = \textsf{\textbf{action-reject}}(D) \\
& \qquad \rangle \\
& \}
\end{aligned}
$$

For $D \in \mathbf{B}$, $E.m \in \mathbf{M}$, and $D \neq E$,

$$
\begin{aligned}
\textsf{\textbf{pm-accept}}(D, E.m) \;\triangleq\; \{ \; & (\mathbf{C}, \mathbf{C}') \\
& \mid \langle \exists \; \varepsilon \\
& \qquad : \varepsilon \in Z \;\; \wedge \;\; \varepsilon \textit{ is a compact execution of } E.m \\
& \qquad : \textit{Start}(\varepsilon) = \mathbf{C} \;\; \wedge \;\; \textit{Final}(\varepsilon) = \mathbf{C}' \;\; \wedge \\
& \qquad \;\; \textit{Last}(\varepsilon) = \textsf{\textbf{test-accept}}(D, E) \\
& \qquad \rangle \\
& \}
\end{aligned}
$$

$$
\begin{aligned}
\textsf{\textbf{pm-reject}}(D, E.m) \;\triangleq\; & \{\ (\mathbf{C}, \mathbf{C}') \\
& \mid \langle \exists\ \varepsilon \\
& \quad :\ \varepsilon \in Z\ \wedge\ \varepsilon \textit{ is a compact execution of } E.m \\
& \quad :\ \mathit{Start}(\varepsilon) = \mathbf{C}\ \wedge\ \mathit{Final}(\varepsilon) = \mathbf{C}'\ \wedge \\
& \quad\ \ \mathit{Last}(\varepsilon) = \textsf{\textbf{test-reject}}(D, E) \\
& \quad \rangle \\
& \}
\end{aligned}
$$

$$
\begin{aligned}
\textsf{\textbf{tm}}(D, E.m) \;\triangleq\; & \{\ (\mathbf{C}, \mathbf{C}') \\
& \mid \langle \exists\ \varepsilon \\
& \quad :\ \varepsilon \in Z\ \wedge\ \varepsilon \textit{ is a compact execution of } E.m \\
& \quad :\ \mathit{Start}(\varepsilon) = \mathbf{C}\ \wedge\ \mathit{Final}(\varepsilon) = \mathbf{C}'\ \wedge \\
& \quad\ \ \mathit{Last}(\varepsilon) = \textsf{\textbf{total-return}}(D, E) \\
& \quad \rangle \\
& \}
\end{aligned}
$$

We define

$$
\begin{aligned}
\textsf{\textbf{accept}} \;\triangleq\; & \langle \cup\ \alpha\ :\ \alpha \in \mathbf{A}\ :\ \textsf{\textbf{accept}}(\alpha)\ \rangle \\
\textsf{\textbf{reject}} \;\triangleq\; & \langle \cup\ \alpha\ :\ \alpha \in \mathbf{A}\ :\ \textsf{\textbf{reject}}(\alpha)\ \rangle \\
\textsf{\textbf{pm-accept}} \;\triangleq\; & \langle \cup\ D, E, m \\
& \ :\ D \in \mathbf{B}\ \wedge\ E.m \in \mathbf{M}\ \wedge\ D \neq E \\
& \ :\ \textsf{\textbf{pm-accept}}(D, E.m) \\
& \ \rangle \\
\textsf{\textbf{pm-reject}} \;\triangleq\; & \langle \cup\ D, E, m \\
& \ :\ D \in \mathbf{B}\ \wedge\ E.m \in \mathbf{M}\ \wedge\ D \neq E \\
& \ :\ \textsf{\textbf{pm-reject}}(D, E.m) \\
& \ \rangle
\end{aligned}
$$

$$
\begin{aligned}
\textsf{\textbf{tm}} \;&\triangleq\; \langle \cup \; D, E, m \\
&\qquad : \; D \in \mathbf{B} \;\wedge\; E.m \in \mathbf{M} \;\wedge\; D \neq E \\
&\qquad : \; \textsf{\textbf{tm}}(D, E.m) \\
&\qquad \rangle \\
\textsf{\textbf{atomic}} \;&\triangleq\; \textsf{\textbf{accept}} \cup \textsf{\textbf{reject}} \cup \textsf{\textbf{pm-accept}} \cup \textsf{\textbf{pm-reject}} \cup \textsf{\textbf{tm}}
\end{aligned}
$$

The following theorems give some of the properties of the **atomic** steps. The first theorem is for accepting **atomic** steps.

Theorem 5.5 *Let*

$$
\begin{aligned}
QT(\mathbf{C}, \mathbf{C}', D) \;&\triangleq\; qt(\mathbf{C}.D) \wedge qt(\mathbf{C}'.D) \\
H(\mathbf{C}, \mathbf{C}', D) \;&\triangleq\; QT(\mathbf{C}, \mathbf{C}', D) \vee \mathbf{C}.D = \mathbf{C}'.D
\end{aligned}
$$

in

$$
\begin{aligned}
&\langle \forall \; D, a, \mathbf{C}, \mathbf{C}' \\
&\quad : \; D.a \in \mathbf{A} \;\wedge\; (\mathbf{C}, \mathbf{C}') \in \textsf{\textbf{accept}}(D.a) \\
&\quad : \; QT(\mathbf{C}, \mathbf{C}', D) \;\wedge\; \langle \forall \; D' \; : \; D' \in \mathbf{B} \; : \; H(\mathbf{C}, \mathbf{C}', D') \rangle \\
&\quad \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\langle \forall \; D, E, m, \mathbf{C}, \mathbf{C}' \\
&\quad : \; D \in \mathbf{B} \;\wedge\; E.m \in \mathbf{M}_p \;\wedge\; D \neq E \;\wedge \\
&\qquad (\mathbf{C}, \mathbf{C}') \in \textsf{\textbf{pm-accept}}(D, E.m) \\
&\quad : \; \mathbf{C}.D.\phi = \textsc{guard} \;\wedge\; \mathbf{C}'.D.\phi = \textsc{accept} \;\wedge\; \mathbf{C}.D.\gamma = \mathbf{C}'.D.\gamma \;\wedge \\
&\qquad QT(\mathbf{C}, \mathbf{C}', E) \;\wedge\; \langle \forall \; D' \; : \; D' \in \mathbf{B} \;\wedge\; D' \neq D \; : \; H(\mathbf{C}, \mathbf{C}', D') \rangle \\
&\quad \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\langle \forall \ D, E, m, \mathbf{C}, \mathbf{C}' \\
&\quad : D \in \mathbf{B} \ \wedge \ E.m \in \mathbf{M}_t \ \wedge \ D \neq E \ \wedge \\
&\quad\ \ (\mathbf{C}, \mathbf{C}') \in \textsf{tm}(D, E.m) \\
&\quad : \mathbf{C}.D.\phi = \text{ACCEPT} \ \wedge \ \mathbf{C}'.D.\phi = \text{ACCEPT} \ \wedge \ \mathbf{C}.D.\gamma = \mathbf{C}'.D.\gamma \ \wedge \\
&\quad\ \ QT(\mathbf{C}, \mathbf{C}', E) \ \wedge \ \langle \forall \ D' \ : \ D' \in \mathbf{B} \ \wedge \ D' \neq D \ : \ H(\mathbf{C}, \mathbf{C}', D') \ \rangle \\
&\rangle
\end{aligned}
$$

The predicate $QT(\mathbf{C}, \mathbf{C}', D)$ holds iff D is quiescent in both $\mathbf{C}$ and $\mathbf{C}'$. The predicate $H(\mathbf{C}, \mathbf{C}', D)$ holds if either D has the same configuration in $\mathbf{C}$ and $\mathbf{C}'$, or if it is quiescent in both configurations. Note that if $QT(\mathbf{C}, \mathbf{C}', D)$, then we can have $\mathbf{C}.D.\sigma \neq \mathbf{C}'.D.\sigma$, but all other components for D are the same in both configurations. This predicate expresses the condition on the boxes that may be affected by the **atomic** step. That is, the configuration of a box can be only be affected by an **atomic** step if it is quiescent before the step. This is because the step represents an uninterrupted execution of the procedure. If a method call is made during this procedure call, it must be to a box that is quiescent, since a call to a nonquiescent box cannot execute until another thread takes some steps.

For all three accepting **atomic** steps, the agent for the procedure call (box D for the **accept** case, box E for the method calls) is quiescent before and after the step. All boxes, except the source, for the method call steps, satisfy predicate H. The source of a **pm-accept** step is in phase GUARD before the step, and phase ACCEPT after the step. The source of a **tm** step is in phase ACCEPT before and after the step. In both cases, the source's call queue is unaffected by the step.

The theorem for rejecting **atomic** steps is simpler. A **reject** step does not change the configuration, and a **pm-reject** step changes the configuration only of its source.

Theorem 5.6

$$\begin{array}{l}\langle\forall\ D, a, \mathbf{C}, \mathbf{C}' \\ \quad : D.a \in \mathbf{A} \ \wedge\ (\mathbf{C}, \mathbf{C}') \in \textsf{\textbf{reject}}(D.a) \\ \quad : qt(\mathbf{C}.D) \ \wedge\ \mathbf{C} = \mathbf{C}' \\ \quad \rangle\end{array}$$

$$\begin{array}{l}\langle\forall\ D, E, m, \mathbf{C}, \mathbf{C}' \\ \quad : D \in \mathbf{B} \ \wedge\ E.m \in \mathbf{M}_p \ \wedge\ D \neq E \ \wedge \\ \quad\ \ (\mathbf{C}, \mathbf{C}') \in \textsf{\textbf{pm-reject}}(D, E.m) \\ \quad : \mathbf{C}.D.\phi = \textsc{guard} \ \wedge\ \mathbf{C}'.D.\phi = \textsc{reject} \ \wedge \\ \quad\ \ qt(\mathbf{C}.E) \ \wedge\ \langle\forall\ D' \ :\ D' \in \mathbf{B} \ \wedge\ D' \neq D \ :\ \mathbf{C}.D' = \mathbf{C}'.D' \ \rangle \\ \quad \rangle\end{array}$$

5.2.3 Executions with compound steps

We define some sets new sets of execution, using the rendezvous procedure call steps and the atomic steps.

We define groups of procedure call rules, according to the function of each rule, queue call, initialization, or rendezvous call. We first define some groups of rules for procedure call and initialization.

Definition 5.7

$$\begin{array}{rcl}\textsf{\textbf{queue}} & \triangleq & \textsf{\textbf{action-start}} \ \cup\ \textsf{\textbf{guard-test}} \ \cup\ \textsf{\textbf{total-call}} \\ \textsf{\textbf{init}} & \triangleq & \textsf{\textbf{p-action-init}} \ \cup\ \textsf{\textbf{p-method-init}} \ \cup \\ & & \textsf{\textbf{t-action-init}} \ \cup\ \textsf{\textbf{t-method-init}} \\ \textsf{\textbf{rdv}} & \triangleq & \textsf{\textbf{p-action-start-rdv}} \ \cup\ \textsf{\textbf{t-action-start-rdv}} \ \cup \\ & & \textsf{\textbf{guard-test-rdv}} \ \cup\ \textsf{\textbf{total-call-rdv}}\end{array}$$

The steps in **queue** are the queue semantics procedure call steps; those in **init** are the queue semantics procedure initialization steps; and those in **rdv** are the rendezvous procedure call steps. Note that the steps in **queue** are the only steps with an unconditional locus. The rules for the **rdv** steps have no unconditional loci.

We define three sets of executions. The first set contains executions with any of the labels defined so far, the labels from the queue semantics, the rendezvous calls, and the atomic steps. This is the most general type of execution that we use. We call these *mixed* executions. The second set contains mixed executions with no **queue** or **init** steps, so all procedure call steps are **rdv** steps. We call the *rendezvous* executions. The final set contains executions with only **accept** and **reject** steps, that is, every step is the complete execution of a thread. We call these *atomic* executions.

First we define some restricted sets of configurations for these executions. The first set contains only configurations where every box is quiescent. These are the configurations used for atomic executions. The second contains only configurations where each box is either quiescent, or nonidle, and has a single entry in its call queue. These are the configurations used for the rendezvous semantics. They exclude configurations where there are more than one entry in the call queue, or configurations where a box is idle and has an entry in its call queue.

Definition 5.8 *If* $\mathbf{C} \in PC$*, then*

$$
\begin{array}{rcl}
PC_q & \triangleq & \{ \, \mathbf{C} \mid \mathbf{C} \in PC \, \wedge \, qt(\mathbf{C}) \, \} \\
rv(\mathbf{C}) & \triangleq & \langle \forall \; D \\
 & & \quad : D \in \mathbf{B} \\
 & & \quad : qt(\mathbf{C}.D) \;\vee\; (\; |\mathbf{C}.D.\gamma| = 1 \;\wedge\; \mathbf{C}.D.\phi \neq \textsc{idle} \;) \\
 & & \quad \rangle \\
PC_r & \triangleq & \{ \, \mathbf{C} \mid \mathbf{C} \in PC \, \wedge \, rv(\mathbf{C}) \, \}
\end{array}
$$

Definition 5.9 *Let*

$$
\begin{array}{rcl}
Lab_0 & \triangleq & \{\ \mathbf{L} \\
& & \mid \mathbf{L} \textit{ is a label generated by one of} \\
& & \quad \textsf{\textbf{(p-action-start-rdv)}} \\
& & \quad \textsf{\textbf{(t-action-start-rdv)}} \\
& & \quad \textsf{\textbf{(guard-test-rdv)}} \\
& & \quad \textsf{\textbf{(total-call-rdv)}} \\
& & \} \\
Lab_1 & \triangleq & \{\ D, E, m, l \\
& & :\ D, E \in \mathbf{B}\ \wedge\ D \neq E\ \wedge\ m \in PartMeths(E)\ \wedge \\
& & \quad l\ \in\ \{\textsf{\textbf{pm-accept}}, \textsf{\textbf{pm-reject}}\} \\
& & :\ l(D, E.m) \\
& & \} \\
Lab_2 & \triangleq & \{\ D, E, m \\
& & :\ D, E \in \mathbf{B}\ \wedge\ D \neq E\ \wedge\ m \in TotMeths(E) \\
& & :\ \textsf{\textbf{total-method}}(D, E.m) \\
& & \}
\end{array}
$$

in

$$
\begin{array}{rcl}
Lab_a & \triangleq & \{\ \alpha, l\ :\ \alpha \in \mathbf{A}\ \wedge\ l\ \in\ \{\textsf{\textbf{accept}}, \textsf{\textbf{reject}}\}\ :\ l(\alpha)\ \} \\
Lab_m & \triangleq & Lab\ \cup\ Lab_a\ \cup\ Lab_0\ \cup\ Lab_1\ \cup\ Lab_2 \\
Lab_r & \triangleq & \{\ \mathbf{L}\ \mid\ \mathbf{L} \in Lab_m\ \wedge\ \mathbf{L} \not\subseteq \textsf{\textbf{queue}} \cup \textsf{\textbf{init}}\ \} \\
Z_a & \triangleq & Executions(PC_q, Lab_a) \\
Z_m & \triangleq & Executions(PC, Lab_m) \\
Z_r & \triangleq & Executions(PC_r, Lab_r)
\end{array}
$$

The final aim of reduction is to transform an execution $\varepsilon \in Z$ to an execution in Z_a. The intermediate executions during reduction are from Z_m and Z_r. We extend the definition of *Root* to cover the new labels.

Definition 5.10 *For $\varepsilon \in Z_m$, $0 \le i < |\varepsilon|$, and $\varepsilon\langle i\rangle \in Lab_m \setminus Lab$,*

$$
\begin{array}{rcll}
Root(\varepsilon, i) & \triangleq & \alpha & \text{if } \varepsilon\langle i\rangle = \textsf{\textbf{accept}}(\alpha) \;\vee \\
& & & \phantom{\text{if }} \varepsilon\langle i\rangle = \textsf{\textbf{reject}}(\alpha) \\
& & Root(\varepsilon[i], D) & \text{if } \varepsilon\langle i\rangle = \textsf{\textbf{guard-test-rdv}}(D, E) \;\vee \\
& & & \phantom{\text{if }} \varepsilon\langle i\rangle = \textsf{\textbf{total-call-rdv}}(D, E) \;\vee \\
& & & \phantom{\text{if }} \varepsilon\langle i\rangle = \textsf{\textbf{pm-accept}}(D, \mu) \;\vee \\
& & & \phantom{\text{if }} \varepsilon\langle i\rangle = \textsf{\textbf{pm-reject}}(D, \mu) \;\vee \\
& & & \phantom{\text{if }} \varepsilon\langle i\rangle = \textsf{\textbf{tm}}(D, \mu) \\
& & Root(\varepsilon[i+1], D) & \text{if } \varepsilon\langle i\rangle = \textsf{\textbf{p-action-start-rdv}}(D) \;\vee \\
& & & \phantom{\text{if }} \varepsilon\langle i\rangle = \textsf{\textbf{t-action-start-rdv}}(D)
\end{array}
$$

5.3 Step types

We can identify groups of the 25 labels in Lab_m containing labels that play a similar part during reduction. We have already seen the groups of resource aquisition steps, **queue**, **init**, and **rdv**. We define some groups for the other steps.

5.3.1 Decision steps

We define the groups of *decision steps.* We show that every complete thread has exactly one decision step. The decision step acts as the central step, the point about which we coalesce the steps for the thread. We choose as a decision step the first step that leaves a box executing for the thread in phase ACCEPT or REJECT. For a total action, this is the **t-action-init** that initializes the action. For a partial action, this is a **guard-accept** or **guard-reject** step in the action's box, or in one of the

partial methods called as a test. A compound step is a decision step if it contains a decision step. Thus the atomic steps for partial methods and for actions, and the rendezvous call **t-action-start-rdv** are decision steps, We distinguish between accept and reject decision steps.

Definition 5.11

$$\begin{aligned}
\textsf{\textbf{ACC}} &\triangleq \textsf{\textbf{accept}} \cup \textsf{\textbf{pm-accept}} \cup \textsf{\textbf{guard-accept}} \cup \\
&\quad\ \textsf{\textbf{t-action-init}} \cup \textsf{\textbf{t-action-start-rdv}} \\
\textsf{\textbf{REJ}} &\triangleq \textsf{\textbf{reject}} \cup \textsf{\textbf{pm-reject}} \cup \textsf{\textbf{guard-reject}} \\
\textsf{\textbf{DEC}} &\triangleq \textsf{\textbf{ACC}} \cup \textsf{\textbf{REJ}}
\end{aligned}$$

For any $\mathbf{L} \in Lab_m$*, if* $\mathbf{L} \subseteq \textsf{\textbf{ACC}}$*, we call* $\mathbf{L}$ *an* accept decision step*; if* $\mathbf{L} \subseteq \textsf{\textbf{REJ}}$*, we call* $\mathbf{L}$ *a* reject decision step*; and if* $\mathbf{L} \subseteq \textsf{\textbf{DEC}}$*, we call* $\mathbf{L}$ *a* decision step*.*

We use the following definition to state the threorems about decision steps.

Definition 5.12 *For* $\varepsilon \in Z_m$*, and* $\pi \in \mathbf{P}$*,*

$$\begin{aligned}
&\varepsilon \textit{ is a } \text{terminating execution} \textit{ for } \pi \ \triangleq \\
&\qquad \varepsilon \textit{ is a full execution for } \pi \ \wedge \ \neg dl(\varepsilon) \ \wedge \ |\varepsilon| < \infty
\end{aligned}$$

The following theorem shows that there is exactly one decision step in every thread.

Theorem 5.13

$$\begin{aligned}
&\langle \forall \ \alpha, \varepsilon \\
&\ \ : \alpha \in \mathbf{A} \ \wedge \ \varepsilon \in Z_m \ \wedge \ \varepsilon \textit{ is a terminating execution for } \alpha \\
&\ \ : \langle \exists!\, i \ : \ 0 \le i < |\varepsilon| \ : \ Root(\varepsilon, i) = \alpha \ \wedge \ \varepsilon\langle i \rangle \subseteq \textsf{\textbf{DEC}} \ \rangle \\
&\ \ \rangle
\end{aligned}$$

The next theorem shows that the type of the decision step corresponds to the type of the thread.

Theorem 5.14

$$\begin{array}{l} \langle\forall\ \alpha, \varepsilon, i \\ \quad :\ \alpha \in \mathbf{A}\ \wedge\ \varepsilon \in Z_m\ \wedge\ \varepsilon\ \textit{is a terminating execution for}\ \alpha\ \wedge \\ \quad\ \ \mathit{Root}(\varepsilon, i) = \alpha\ \wedge\ \varepsilon\langle i\rangle \subseteq \mathsf{\mathbf{DEC}} \\ \quad :\ \varepsilon\langle i\rangle\ \subseteq\ \mathsf{\mathbf{ACC}}\ \equiv\ \mathit{Last}(\varepsilon)\ \subseteq\ \mathsf{\mathbf{action\text{-}end}} \cup \mathsf{\mathbf{accept}} \\ \quad \rangle \end{array}$$

5.3.2 Right-movers and left-movers

We define groups for the steps that hold resources and compute locally, and for steps that end procedures, releasing resources.

Definition 5.15

$$\begin{array}{rcl} \mathsf{\mathbf{local}} & \triangleq & \mathsf{\mathbf{local\text{-}step}}\ \cup\ \mathsf{\mathbf{proc\text{-}term}} \\ \mathsf{\mathbf{end}} & \triangleq & \mathsf{\mathbf{action\text{-}end}}\ \cup\ \mathsf{\mathbf{test\text{-}accept}}\ \cup\ \mathsf{\mathbf{total\text{-}return}}\ \cup \\ & & \mathsf{\mathbf{action\text{-}reject}}\ \cup\ \mathsf{\mathbf{test\text{-}reject}} \end{array}$$

Every step in Lab_m, apart from **tm**, is in one of the groups **DEC**, **queue**, **init**, **rdv**, **local**, or **end**. We define two final groups, for the steps that move right during reduction, and those that move left.

Definition 5.16

$$\begin{array}{rcl} \mathsf{\mathbf{RM}} & \triangleq & \mathsf{\mathbf{queue}}\ \cup\ \mathsf{\mathbf{init}}\ \cup\ \mathsf{\mathbf{rdv}} \\ \mathsf{\mathbf{LM}} & \triangleq & \mathsf{\mathbf{local}}\ \cup\ \mathsf{\mathbf{end}}\ \cup\ \mathsf{\mathbf{tm}} \end{array}$$

For $\mathbf{L} \in Lab_m$*, if* $\mathbf{L} \subseteq \textsf{\textbf{RM}}$*, we call* $\mathbf{L}$ *a* right-mover*, and if* $\mathbf{L} \subseteq \textsf{\textbf{LM}}$*, we call* $\mathbf{L}$ *a* left-mover*.*

The groups **DEC**, **RM**, and **LM** partition Lab_m. We show later that the right-movers can move right over any step from a different thread. That is, in any execution ε, where $|\varepsilon| = 2$, and the first step of ε is a right-mover, and the last step is a step from a different thread, then there is a two-step execution ε' where

$$\begin{aligned} &Start(\varepsilon') \ = \ Start(\varepsilon) \ \wedge \ Final(\varepsilon') \ = \ Final(\varepsilon) \ \wedge \\ &SSeq(\varepsilon') \ = \ \langle\, \varepsilon\langle 1\rangle \ ,\ \varepsilon\langle 0\rangle \,\rangle \end{aligned}$$

That is, ε' has the same start and final configurations as ε, and it has the same steps, but in the reverse order. Thus is any execution where ε appears, it can be replaced with ε' with no change to the rest of the execution. Similarly, we show that the left-movers, apart from **tm**, can move left in any execution. To complete the reduction for TCB, we need to move **tm** steps left. We discuss the requirements for this later.

5.3.3 The format of a thread

The following theorems show some results about the steps before and after the decision step in a thread. The first shows the type of steps that may appear before the decision step.

Theorem 5.17

$$\begin{aligned} &\langle \forall\ \varepsilon, i, k \\ &\quad : \varepsilon \in Z_m \ \wedge\ 0 \le i < k < |\varepsilon| \ \wedge\ \varepsilon\langle k\rangle \subseteq \textsf{\textbf{DEC}} \ \wedge\ Root(\varepsilon, i) = Root(\varepsilon, k) \\ &\quad : \varepsilon\langle i\rangle \subseteq \textsf{\textbf{action-start}} \ \cup\ \textsf{\textbf{guard-test}} \ \cup\ \textsf{\textbf{p-action-init}} \ \cup \\ &\qquad\qquad \textsf{\textbf{p-method-init}} \ \cup\ \textsf{\textbf{p-action-start-rdv}} \ \cup\ \textsf{\textbf{guard-test-rdv}} \\ &\quad \rangle \end{aligned}$$

The second theorem shows the type of steps that may appear after an accept decision step.

Theorem 5.18

$$\langle \forall\ \varepsilon, i, k$$
$$: \varepsilon \in Z_m \ \wedge\ 0 \leq k < i < |\varepsilon| \ \wedge\ \varepsilon\langle k \rangle \subseteq \textbf{ACC} \ \wedge\ \mathit{Root}(\varepsilon, i) = \mathit{Root}(\varepsilon, k)$$
$$: \varepsilon\langle i \rangle \subseteq \textbf{total-call} \ \cup\ \textbf{t-method-init} \ \cup\ \textbf{total-call-rdv} \ \cup\ \textbf{action-end} \ \cup$$
$$\textbf{test-accept} \ \cup\ \textbf{total-return} \ \cup\ \textbf{local} \ \cup\ \textbf{tm}$$
$$\rangle$$

The third theorem show the type of steps that may appear after a reject decision step.

Theorem 5.19

$$\langle \forall\ \varepsilon, i, k$$
$$: \varepsilon \in Z_m \ \wedge\ 0 \leq k < i < |\varepsilon| \ \wedge\ \varepsilon\langle k \rangle \subseteq \textbf{REJ} \ \wedge\ \mathit{Root}(\varepsilon, i) = \mathit{Root}(\varepsilon, k)$$
$$: \varepsilon\langle i \rangle \subseteq \textbf{action-reject} \ \cup\ \textbf{test-reject}$$
$$\rangle$$

We note from these theorems that all steps before a **DEC** step are from **RM**. All steps after a **REJ** step are from **LM**. The steps after an **ACC** step are a mixture of right-movers and left-movers.

The final result in this section shows that only certain steps can change the persistent state of the system, that is, the values of the box variables.

Theorem 5.20

$$\begin{aligned}
&\langle \forall\ \mathbf{L}, \mathbf{C}, \mathbf{C}', D \\
&\quad : \mathbf{L} \in Lab_m\ \wedge\ \mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}'\ \wedge \\
&\quad\quad \mathbf{L} \not\subseteq \textbf{accept}\ \cup\ \textbf{pm-accept}\ \cup\ \textbf{tm}\ \cup \\
&\quad\quad\quad \textbf{local-step}\ \cup\ \textbf{test-accept}\ \cup\ \textbf{total-return} \\
&\quad : PersistEq(\mathbf{C}, \mathbf{C}') \\
&\quad \rangle
\end{aligned}$$

From the above theorems, we note that the only steps that change the persistent state during an execution are accept decision steps, or steps that appear in a thread after an accept decision step. The persistent state is not changed by a thread before its decision step, and it is not changed by any step from a rejecting thread. This was our intention in defining *TCB*, that the decision to accept or reject be made prior to any change to the box variables, and that rejecting threads leave the state unchanged.

5.3.4 A strategy for reduction

Theorem 5.18 shows that there are right-movers in a thread after an accept decision step. The aim of reduction is to bring all the steps of the thread to the decision step, so the right-movers after the decision steps must be moved left. We accomplish this using **tm** steps.

Suppose there is a call to method μ during a thread's execution. The first two steps for this call are right-movers. First is a **total-call**, and then a **t-method-init**. Suppose the remainder of the steps for the call are all **local** or **end** steps, so there are no further method calls during the execution of the call. We can move the **total-call** step right, and the **local** and **end** steps left, until they are contiguous with the **t-method-init** step. This gives a compact execution of the method, which

can be replaced with a **tm** step.

Now suppose the call to μ is part of another method call. Originally, the steps for this call contain two right-movers, the **total-call** and **t-method-init** steps for the call to μ. After the transformation suggested above, these (along with a number of left-movers) have been replaced with a single **tm** step, a left-mover.

This is the strategy used to move a right-mover after an accept decision step: we move it right until it can be replaced by a **tm** step, and then move this step left. For a **total-call** step this gives a rather circuitous route for reduction, a case of "one step forward, two steps back".

5.4 Reduction relations

We define *reduction relations* on executions. We define a relation $\leadsto$ such that if $\varepsilon \leadsto \varepsilon'$, then ε' can be formed from ε by a sequence of steps moving a right-mover right or a left-mover left. This means that the sequence of decision steps is the same in ε and ε', since decision steps are neither right-movers nor left-movers. The aim is to reduce an execution to an atomic execution by moving steps in this way.

We show that if $\varepsilon \leadsto \varepsilon'$, then there is a relationship between the configurations of ε and ε' that allows us to infer properties of ε from properties of ε'. If ε' is an atomic execution, then ε has behaviour "equivalent" to an atomic execution.

We define this relation is stages. Define a relation $\xrightarrow{*}$ that is sufficient for reducing finite executions, and we show how to build $\leadsto$ from this so that we can reduce infinite executions as well.

5.4.1 Accept decision steps

For the purposes of reduction, the basic structure of an execution is given by the sequence of accept decision steps. These are the decision steps from threads that change the state of the box variables.

Definition 5.21 *For $\varepsilon \in Z_m$, and p an ascending sequence containing every i, such that $\varepsilon\langle i\rangle \subseteq$* **ACC**,

$$\begin{array}{rcll} NumA(\varepsilon) & \triangleq & |p| & \\ AS(\varepsilon,i) & \triangleq & \varepsilon\langle p[i]\rangle & \\ AC(\varepsilon,i) & \triangleq & \varepsilon[p[i]] & \\ EB(\varepsilon,i) & \triangleq & \varepsilon\langle 0\ldots(p[0]-1)\rangle & \text{if } \ i=0 \\ & & \varepsilon\langle (p[i-1]+1)\ldots(p[i]-1)\rangle & \text{if } \ 0<i<NumA(\varepsilon) \\ ASeq(\varepsilon) & \triangleq & \langle i \ : \ 0\leq i<NumA(\varepsilon) \ : \ Root(\varepsilon,p[i]) \ \rangle & \end{array}$$

We call $NumA(\varepsilon)$ the number of accept decision steps *in ε, $AS(\varepsilon,i)$ the ith* accept decision step *of ε, $AC(\varepsilon,i)$ the ith* accept configuration *of ε, $EB(\varepsilon,i)$ the ith* execution block *of ε, and $ASeq(\varepsilon)$ the* accept sequence *of ε.*

For an execution ε, $NumA(\varepsilon)$ is the number of accept decision steps in ε. If $NumA(\varepsilon) = 3$, then ε can be partitioned into accept decision steps and execution blocks in the following way.

$$EB(\varepsilon,0) \ ; \ AS(\varepsilon,0) \ ; \ EB(\varepsilon,1) \ ; \ AS(\varepsilon,1) \ ; \ EB(\varepsilon,2) \ ; \ AS(\varepsilon,2) \ ; \ \hat{\varepsilon}$$

The execution starts with the steps from execution block $EB(\varepsilon,0)$. The configuration after these steps is accept configuration $AC(\varepsilon,0)$. The next step the accept decision step $AS(\varepsilon,0)$, followed by the steps from execution block $EB(\varepsilon,1)$. The configuration after these steps is accept configuration $AC(\varepsilon,1)$. The execution continues in this way. Note that the steps after the final accept decision step, given as $\hat{\varepsilon}$ above, are in no execution block.

5.4.2 Similar executions

We define a relation on executions using the accept sequence.

Definition 5.22 *For* $\varepsilon, \varepsilon' \in Z_m$,

$$\begin{aligned} \varepsilon \doteq \varepsilon' \quad \triangleq \quad & \mathit{Start}(\varepsilon) = \mathit{Start}(\varepsilon') \ \wedge \ \mathit{ASeq}(\varepsilon) = \mathit{ASeq}(\varepsilon') \ \wedge \\ & (|\varepsilon| = \infty \ \equiv \ |\varepsilon'| = \infty) \ \wedge \ (\mathit{Final}(\varepsilon) = \mathit{Final}(\varepsilon') \ \vee \ |\varepsilon| = \infty) \end{aligned}$$

If $\varepsilon \doteq \varepsilon'$, *we say that* ε *is* similar to ε'.

The following are straightforward consequences of this definition.

Theorem 5.23 $\doteq$ *is an equivalence relation.*

Theorem 5.24

$$\begin{aligned} &\langle \forall \ \varepsilon_0, \varepsilon_1, \varepsilon_1', \varepsilon_2 \\ &\quad : \ \varepsilon_0, \varepsilon_1, \varepsilon_1', \varepsilon_2 \in Z_m \ \wedge \ \varepsilon_0; \varepsilon_1; \varepsilon_2 \ \mathit{is\ defined} \ \wedge \ \varepsilon_1 \doteq \varepsilon_1' \\ &\quad : \ \varepsilon_0; \varepsilon_1'; \varepsilon_2 \ \mathit{is\ defined} \ \wedge \ \varepsilon_0; \varepsilon_1; \varepsilon_2 \doteq \varepsilon_0; \varepsilon_1'; \varepsilon_2 \\ &\quad \rangle \end{aligned}$$

Theorem 5.24 says that similarity is a congruence relation, that is, for any execution ε, replacing a finite segment of ε with a similar finite execution yields an execution similar to ε.

A pair of executions with no accept decision steps are similar iff they have the same start and final configuration. This gives the following theorem, which says that an execution consisting of a single stuttering step that is not an accept decision step is similar to an empty execution with the same start configuration.

Theorem 5.25

$$\begin{aligned} &\langle \forall \ \mathbf{L}, \mathbf{C} \\ &\quad : \ \mathbf{L} \in \mathit{Lab}_m \ \wedge \ \mathbf{L} \not\subseteq \mathsf{ACC} \ \wedge \ \mathbf{C}\langle \mathbf{L} \rangle \mathbf{C} \\ &\quad : \ (\ \langle \mathbf{C}, \mathbf{C} \rangle \ , \ \langle \mathbf{L} \rangle \ ) \ \doteq \ (\ \langle \mathbf{C} \rangle \ , \ \bot \ ) \\ &\quad \rangle \end{aligned}$$

Using Theorems 5.24 and 5.25, we can remove a stuttering step from an execution, and the resulting execution is similar to the original.

5.4.3 Swapping steps

If two adjacent steps in an execution execute at disjoint locations, then the changes they make to the configuration are independent, and so can be taken in either order. The execution with these steps in the opposite order is similar to the original.

We define conditions for two steps to be swapped. If we are to swap two steps, the steps must be from different threads. We do not want to change the order of steps from the same thread. For some steps, we can tell from the labels whether they are from the same of different threads. For example, if we have execution ε, where

$$SSeq(\varepsilon) \quad = \quad \langle\, \textbf{local-step}(D)\;,\; \textbf{local-step}(D)\,\rangle$$

then we can show that these steps are from the same thread. If, on the other hand, we have

$$SSeq(\varepsilon) \quad = \quad \langle\, \textbf{action-start}(D)\;,\; \textbf{total-call}(E, D)\,\rangle$$

then we can show that these steps are from different threads. Consider the following case.

$$SSeq(\varepsilon) \quad = \quad \langle\, \textbf{action-start}(D)\;,\; \textbf{p-action-init}(D)\,\rangle$$

These steps may be from the same thread, and they may be from different threads. If D's call queue is empty before the **action-start** step, then the **p-action-init** step initializes the action call placed by the first step, and the steps are from the same thread. Otherwise, if D's call queue is nonempty before the **action-start** step, then

the p-**action-init** step initializes a different action call, and the steps are from different threads.

Thus we need to know the context before we can tell if a pair of steps can be exchanged. Because of this, we discuss swapping steps in the context of executions with two steps. We define the following sets of two-step executions, and we define the swap of an execution in one of these sets.

Definition 5.26 *For* $\mathbf{L}, \mathbf{L}' \in Lab_m$*,*

$$TwoStep(\mathbf{L}, \mathbf{L}') \quad \triangleq \quad \{\, \varepsilon \mid \varepsilon \in Z_m \;\wedge\; |\varepsilon| = 2 \;\wedge\; Root(\varepsilon, 0) \neq Root(\varepsilon, 1) \,\}$$

For $\varepsilon \in TwoStep(\mathbf{L}, \mathbf{L}')$*,* $Swap(\varepsilon)$ *is an execution* $\varepsilon' \in TwoStep(\mathbf{L}', \mathbf{L})$*, where*

$$Start(\varepsilon) \;=\; Start(\varepsilon') \quad \wedge \quad Final(\varepsilon) \;=\; Final(\varepsilon')$$

If no such ε' *exists, then* $Swap(\varepsilon) = \bot$*.*

The swap of a two-step execution is a two-step execution with the same steps, in reverse order, and the same start and final configuration. The steps of a two-step execution ε can be swapped iff $Swap(\varepsilon) \neq \bot$. We define a relation on labels that is true only if the labels can always be swapped.

Definition 5.27

$$\mathbf{L} \curvearrowleft \mathbf{L}' \quad \triangleq \quad \langle \forall\, \varepsilon \;:\; \varepsilon \in TwoStep(\mathbf{L}, \mathbf{L}') \;:\; Swap(\varepsilon) \neq \bot \,\rangle$$

If $\mathbf{L} \curvearrowleft \mathbf{L}'$, then steps $\mathbf{L}$ and $\mathbf{L}'$ can be swapped in any execution where $\mathbf{L}$ is followed by $\mathbf{L}'$, and the steps are for different threads. We have the following theorem.

Theorem 5.28

$$\begin{aligned}
&\langle \forall\ \mathbf{L}, \mathbf{L}', \varepsilon \\
&\quad : \mathbf{L}, \mathbf{L}' \in \mathit{Lab}_m \ \wedge\ \mathbf{L} \curvearrowleft \mathbf{L}' \ \wedge\ (\ \mathbf{L} \not\subseteq \textsf{ACC} \ \vee\ \mathbf{L}' \not\subseteq \textsf{ACC}\ ) \ \wedge \\
&\quad\ \ \varepsilon \in \mathit{TwoStep}(\mathbf{L}, \mathbf{L}') \\
&\quad : \varepsilon \doteq \mathit{Swap}(\varepsilon) \\
&\quad \rangle
\end{aligned}$$

The theorem says that if steps $\mathbf{L}$ and $\mathbf{L}'$ are swappable, that is $\mathbf{L} \curvearrowleft \mathbf{L}'$, and at least one of them is not an accept decision step, then any execution ε with these two steps is similar to $\mathit{Swap}(\varepsilon)$.

5.4.4 Reduce-equivalence

Similarity between executions plays an important part in reduction. As we rearrange an execution, we do not change the accept sequence, the start configuration, or the final configuration (if any). So we only reduce an execution to similar executions. These conditions alone are insufficient for infinite executions, as the following example demonstrates.

Consider a program containing two total actions, α and α'. Action α nondeterministically assigns 1 or 2 to x, and action α' increments y. In the initial configuration, $x = 0$ and $y = 0$. Consider now two executions of this program. Execution ε is a sequential execution that consists of a single thread for α that assigns 1 to x, followed by an infinite sequence of threads for α' that successively increment y. Execution ε' has the same form, except that the initial thread assigns 2 to x. For these executions we have $\varepsilon \doteq \varepsilon'$, but all configurations, except the start, are different between the executions. We do not want ε to be a reduction of ε', or vice versa.

We define a relation that distinguishes these two executions.

Definition 5.29 *For* $\varepsilon, \varepsilon' \in Z_m$,

$$\varepsilon \leftrightarrow \varepsilon' \;\triangleq\; \varepsilon \doteq \varepsilon' \;\wedge\; \langle \forall\, i \;:\; 0 \leq i < NumA(\varepsilon) \;:\; AC(\varepsilon, i) = AC(\varepsilon', i) \;\rangle$$

If $\varepsilon \leftrightarrow \varepsilon'$, *we say that* ε *and* ε' *are* reduce-equivalent.

The following theorems summarize the important properties of reduce-equivalence.

Theorem 5.30 $\leftrightarrow$ *is an equivalence relation.*

Theorem 5.31

$$\begin{array}{l}\langle \forall\ \varepsilon_0, \varepsilon_1, \varepsilon_1', \varepsilon_2 \\ \quad : \varepsilon_0, \varepsilon_1, \varepsilon_1', \varepsilon_2 \in Z_m \;\wedge\; \varepsilon_0; \varepsilon_1; \varepsilon_2 \textit{ is defined} \;\wedge\; \varepsilon_1 \leftrightarrow \varepsilon_1' \\ \quad : \varepsilon_0; \varepsilon_1'; \varepsilon_2 \textit{ is defined} \;\wedge\; \varepsilon_0; \varepsilon_1; \varepsilon_2 \leftrightarrow \varepsilon_0; \varepsilon_1'; \varepsilon_2 \\ \quad \rangle\end{array}$$

Theorem 5.31 means that reduce-equivalence is a congruence, as with similarity.

The following theorem shows that in the case where at least one of the executions has no accept decision steps (and thus no accept configurations), then similarity and reduce-equivalence are the same relation.

Theorem 5.32

$$\begin{array}{l}\langle \forall\ \varepsilon, \varepsilon' \\ \quad : \varepsilon, \varepsilon' \in Z_m \;\wedge\; (\; NumA(\varepsilon) = 0 \;\vee\; NumA(\varepsilon') = 0 \;) \\ \quad : \varepsilon \leftrightarrow \varepsilon' \;\equiv\; \varepsilon \doteq \varepsilon' \\ \quad \rangle\end{array}$$

Proof

Immediate from the definitions.

(End of proof)

From Theorems 5.32 and 5.28 we get the following theorem, which says that when it is possible to swap two steps, neither of which is an accept decision step, the swapped execution is reduce-equivalent to the original.

Theorem 5.33

$$
\begin{array}{l}
\langle \forall\ \mathbf{L}, \mathbf{L}', \varepsilon \\
\quad :\ \mathbf{L}, \mathbf{L}' \in Lab_m \ \wedge\ \mathbf{L} \curvearrowleft \mathbf{L}' \ \wedge\ \mathbf{L} \not\subseteq \textsf{\textbf{ACC}} \ \wedge\ \mathbf{L}' \not\subseteq \textsf{\textbf{ACC}} \ \wedge \\
\quad\ \ \ \varepsilon \in TwoStep(\mathbf{L}, \mathbf{L}') \\
\quad :\ \varepsilon \leftrightarrow Swap(\varepsilon) \\
\quad \rangle
\end{array}
$$

5.4.5 Finite reduction

Reduce-equivalence gives is a strong relation on executions. Clearly two executions that are reduce-equivalent have very similar behaviours. But reduce-equivalence is too strong a relation for reduction.

Consider $\varepsilon \in Complete(Z)$, and suppose $\varepsilon' \in Z_a$ is an atomic reduction of ε. We have $\varepsilon \doteq \varepsilon'$. Consider accept decision steps $AS(\varepsilon, i)$ and $AS(\varepsilon', i)$. The former is a **t-action-init** or **guard-accept** step, while the latter is a **accept** step.

Accept configuration $AC(\varepsilon', i)$ is quiescent, so there are no active threads. Accept configuration $AC(\varepsilon, i)$, on the other hand, shows the effect of partly completed threads. Let $AS(\varepsilon, k)$ be the accept decision step for a thread that is active in $AC(\varepsilon, i)$. If $k < i$, then steps from this thread appear after $AC(\varepsilon, i)$. To make a compact execution of this thread, some of its steps must move left over $AS(\varepsilon, i)$. Similarly, if $k \geq i$, then some of the thread's steps must move right over $AS(\varepsilon, i)$. We define some relations on executions that allow such a swap below. This relation allows a swap where a right-mover moves right over an accept decision step, or a left-mover moves left.

Consider a pair of steps $\mathbf{L}_r$ and $\mathbf{L}_a$, where $\mathbf{L}_r$ is a right-mover, and $\mathbf{L}_a$ is an accept decision step. Suppose we show $\mathbf{L}_r \curvearrowleft \mathbf{L}_a$, that is, that $\mathbf{L}_r$ can move right over $\mathbf{L}_a$. Choose $\varepsilon \in \mathit{TwoStep}(\mathbf{L}_r, \mathbf{L}_a)$, and let $\varepsilon' = \mathit{Swap}(\varepsilon)$. By Theorem 5.28, we have $\varepsilon \doteq \varepsilon'$. There is only one accept configuration in each execution. We have

$$
\begin{aligned}
AC(\varepsilon, 0) &= \varepsilon[1] \\
AC(\varepsilon', 0) &= \varepsilon'[0]
\end{aligned}
$$

Since $\varepsilon[0] = \varepsilon'[0]$, we get $(AC(\varepsilon', 0), AC(\varepsilon, 0)) \in \mathbf{L}_r$. That is, the accept configuration in ε is not equal to that in ε', but it is reachable from it by a single $\mathbf{L}_r$ step.

Configuration $AC(\varepsilon, 0)$ shows the effect of executing $\mathbf{L}_r$, and configuration $AC(\varepsilon', 0)$ does not. So in ε', the right-mover has been delayed until after the decision step.

Consider now a pair of steps $\mathbf{L}_l$ and $\mathbf{L}_a$, where $\mathbf{L}_l$ is a left-mover, and $\mathbf{L}_a$ is an accept decision step, and $\mathbf{L}_a \curvearrowleft \mathbf{L}_l$. If $\varepsilon \in \mathit{TwoStep}(\mathbf{L}_a, \mathbf{L}_l)$, and $\varepsilon' = \mathit{Swap}(\varepsilon)$, then we get $(AC(\varepsilon, 0), AC(\varepsilon', 0)) \in \mathbf{L}_l$. In this case, the left-mover is moved left of the accept configuration, so the accept configuration is ε does not show the effect of $\mathbf{L}_l$, whereas that in ε' does. We define relations on executions using **RM** and **LM**. By definition, $(\mathbf{C}, \mathbf{C}') \in \mathbf{RM}$, means that there is a right-mover $\mathbf{L}$, such that $\mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}'$, and similarly for $(\mathbf{C}, \mathbf{C}') \in \mathbf{LM}$.

Definition 5.34 *For* $\varepsilon, \varepsilon' \in Z_m$*,*

$$\begin{array}{lcl} \varepsilon \rightarrow_R \varepsilon' & \triangleq & \varepsilon \doteq \varepsilon' \;\wedge \\ & & \langle \forall\; i \\ & & \quad :\; 0 \leq i < NumA(\varepsilon) \\ & & \quad :\; AC(\varepsilon,i) \;=\; AC(\varepsilon',i) \;\vee\; (\; AC(\varepsilon',i)\;,\; AC(\varepsilon,i)\;) \;\in\; \textsf{RM} \\ & & \quad \rangle \\ \varepsilon \rightarrow_L \varepsilon' & \triangleq & \varepsilon \doteq \varepsilon' \;\wedge \\ & & \langle \forall\; i \\ & & \quad :\; 0 \leq i < NumA(\varepsilon) \\ & & \quad :\; AC(\varepsilon,i) \;=\; AC(\varepsilon',i) \;\vee\; (\; AC(\varepsilon,i)\;,\; AC(\varepsilon',i)\;) \;\in\; \textsf{LM} \\ & & \quad \rangle \end{array}$$

If $\varepsilon \rightarrow_R \varepsilon'$*, we say that* ε right reduces *to* ε'*, and if* $\varepsilon \rightarrow_L \varepsilon'$*, we say that* ε left reduces *to* ε'*.*

Note the asymmetry in these definitions. For $\rightarrow_R$, we require that an accept configuration in the original execution be reachable from the corresponding accept configuration in the reduced execution by zero or one **RM** steps, whereas for $\rightarrow_L$, we require that an accept configuration in the reduced execution be reachable from the corresponding accept configuration in the original execution by zero or one **LM** steps. This captures the intention that reduction involves delaying **RM** steps, and advancing **LM** steps.

The following theorems give some properties of these relations.

Theorem 5.35 $\rightarrow_R$ *and* $\rightarrow_L$ *are reflexive.*

Theorem 5.36

$$\langle \forall\ \varepsilon, \varepsilon' \\ :\ \varepsilon, \varepsilon' \in Z_m \\ :\ (\ \varepsilon \leftrightarrow \varepsilon'\ \equiv\ \varepsilon \rightarrow_R \varepsilon' \wedge \varepsilon \rightarrow_L \varepsilon'\ )\ \wedge\ (\ \varepsilon \leftrightarrow \varepsilon'\ \equiv\ \varepsilon \rightarrow_R \varepsilon' \wedge \varepsilon' \rightarrow_R \varepsilon\ ) \\ \rangle$$

Reduction, in general requires a sequence of right- and left-reduce steps. This can be expressed with the following relation.

Definition 5.37

$$\stackrel{*}{\rightarrow}\ \triangleq\ (\rightarrow_R \cup \rightarrow_L)^*$$

If $\varepsilon \stackrel{*}{\rightarrow} \varepsilon'$*, then we say that* ε finitely reduces to ε'*.*

Note that $\stackrel{*}{\rightarrow}$ is reflexive and transitive by construction.

5.4.6 The reduction relation

Relation $\stackrel{*}{\rightarrow}$ is sufficient for reducing finite executions, but it is not sufficient for infinite executions, as the following example demonstrates.

Consider a program with total actions α and α', where α and α' execute on disjoint sets of boxes. Clearly, each step of a thread for α commutes with a step from α'. Assume, for simplicity, every thread for α can be represented by two steps, an accept decision step, $\mathbf{L}_0$, and a left-mover $\mathbf{L}_1$. Assume we have an execution ε where every thread for α has been reduced to $\mathbf{L}_0$ and $\mathbf{L}_1$, and every thread for α' has been reduced to $\mathbf{L}' = \textbf{accept}(\alpha')$. Suppose the sequence of steps in ε has the

following form.

$$\mathbf{L}_0 \,,\, \mathbf{L}' \,,\, \mathbf{L}_1 \,,$$
$$\mathbf{L}_0 \,,\, \mathbf{L}' \,,\, \mathbf{L}' \,,\, \mathbf{L}_1 \,,$$
$$\mathbf{L}_0 \,,\, \mathbf{L}' \,,\, \mathbf{L}' \,,\, \mathbf{L}' \,,\, \mathbf{L}_1 \,,$$
$$\ldots$$

That is, there is one $\mathbf{L}'$ step between the first pair of $\mathbf{L}_0$ and $\mathbf{L}_1$ steps, two $\mathbf{L}'$ steps between the second pair, and so on, with the number of intervening $\mathbf{L}'$ steps increasing by one for each successive pair of $\mathbf{L}_0$ and $\mathbf{L}_1$ steps. Since actions α and α' affect disjoint parts of the state space, we expect to be able to reduce ε to the sequential execution ε', which has the following sequence of steps.

$$\mathbf{L}_0 \,,\, \mathbf{L}_1 \,,\, \mathbf{L}' \,,$$
$$\mathbf{L}_0 \,,\, \mathbf{L}_1 \,,\, \mathbf{L}' \,,\, \mathbf{L}' \,,$$
$$\mathbf{L}_0 \,,\, \mathbf{L}_1 \,,\, \mathbf{L}' \,,\, \mathbf{L}' \,,\, \mathbf{L}' \,,$$
$$\ldots$$

Here the $\mathbf{L}_1$ steps have been moved left so they are adjacent to the $\mathbf{L}_0$ steps. Moving left over n **accept** steps takes n applications of $\rightarrow_L$. Definition 5.34 allows a left-mover to move left over each accept step for a single application of $\rightarrow_L$. But it only allows a given left-mover to move over one accept step for each application. With n applications of $\rightarrow_L$, we can make at most the first n threads for α sequential. Since the sequence is infinite,

$$\langle \forall\, n \;:\; n > 0 \;:\; \neg(\varepsilon \rightarrow_L^n \varepsilon') \;\rangle$$

and thus, since there are no right-movers in ε,

$$\neg(\varepsilon \xrightarrow{*} \varepsilon')$$

We now define a relation $\leadsto$, where $\varepsilon \leadsto \varepsilon'$ for the above example. In the above example, n applications of $\rightarrow_L$ are enough to reduce the first n calls to α. After n applications, the steps in the first part of the execution are not moved again for the remainder of the reduction. We can generate a sequence of executions

$$\varepsilon_0 \, , \, \varepsilon_1 \, , \, \varepsilon_1 \, , \, \ldots$$

where $\varepsilon \xrightarrow{*} \varepsilon_i$, for $i \geq 0$, and ε_i is sequential up to at least the thread containing the ith accept decision step. Note that ε_i and ε_j, for $0 \leq i \leq j$, share a common sequential prefix, containing at least i accept decision steps.

We can regard the ε_i's as a sequence of *approximations* to a sequential reduction for ε. Let ε' be the limit of this sequence, that is, an execution where the sequential execution includes all the accept decision steps. We define a relation $\leadsto$ below such that $\varepsilon \leadsto \varepsilon'$ in this case.

We first define a relation to express the idea of reducing a prefix of an execution.

Definition 5.38 *For* $\varepsilon, \varepsilon' \in Z_m$*, and* $|\varepsilon'| < \infty$*,*

$$\varepsilon \overset{init}{\rightarrow} \varepsilon' \quad \triangleq \quad \langle \exists\, \varepsilon'' \;:\; \varepsilon'' \in Z_m \;:\; \varepsilon \xrightarrow{*} \varepsilon' ; \varepsilon'' \,\rangle$$

If $\varepsilon \overset{init}{\rightarrow} \varepsilon'$*, we say* ε initially reduces *to* ε'*.*

If $\varepsilon \overset{init}{\rightarrow} \varepsilon'$, then ε can be finitely reduced to an execution that starts with the finite execution ε'. We define a sequence finite approximations to a reduction of an execution.

Definition 5.39 *For* $\varepsilon \in Z_m$*, and* $\mathcal{E} = \langle i \ : \ 0 \leq i \ : \ \varepsilon_i \ \rangle$ *a sequence over* Z_m*, where* $|\varepsilon_i| < \infty$ *for all* i*,*

$$
\begin{aligned}
&\mathcal{E} \textit{ is a } \text{convergent reduction sequence} \textit{ for } \varepsilon \ \triangleq \\
&\quad \langle \forall\ i \\
&\quad\ : 0 \leq i \\
&\quad\ : \varepsilon_i \sqsubseteq \varepsilon_{i+1} \ \wedge \ \varepsilon \overset{init}{\rightarrow} \varepsilon_i \ \wedge \\
&\qquad (\ NumA(\varepsilon_i) < NumA(\varepsilon) \ \Rightarrow \\
&\qquad\quad \langle \exists\ j \ : \ i < j \ : \ NumA(\varepsilon_i) < NumA(\varepsilon_j) \ \rangle \ ) \\
&\quad \rangle
\end{aligned}
$$

We abbreviate "convergent reduction sequence" to CRS.

The final conjunct in the term in Definition 5.39 is a stronger condition than just requiring that the executions in a CRS be increasing in length. To see why this is necessary, consider ε_i such that $NumA(\varepsilon_i) < NumA(\varepsilon)$, where for some action α, **reject**(α) is enabled in $Final(\varepsilon_i)$. Let ε' be an execution with a single **reject**(α) step, If we define

$$
\begin{aligned}
\varepsilon' &\triangleq (\ \langle \ Final(\varepsilon_i) \ , \ Final(\varepsilon_i) \ \rangle \ , \ \langle \ \textbf{reject}(\alpha) \ \rangle \ ) \\
\varepsilon_{i+1} &\triangleq \varepsilon_i ; \varepsilon' \\
\varepsilon_{i+2} &\triangleq \varepsilon_{i+1} ; \varepsilon' \\
&\vdots
\end{aligned}
$$

This gives a sequence of executions that satisfies all the other conditions of Definition 5.39, and is increasing in length. All executions in this sequence from ε_i have the same accept sequence as ε_i, which is a proper prefix of ε's accept sequence. This is not a sequence that approximates a reduction of ε.

The set Z_m is a complete partial order under $\sqsubseteq$, so every chain has a limit

in Z_m. We use this to define the reduction relation $\rightsquigarrow$.

Definition 5.40 *For $\varepsilon, \varepsilon' \in Z_m$,*

$$\varepsilon \rightsquigarrow \varepsilon' \quad \triangleq \quad \langle \exists\, \mathcal{E} \ :\ \mathcal{E} \textit{ is a CRS for } \varepsilon \ :\ \varepsilon' = \langle \sqcup\, i \ :\ 0 \leq i \ :\ \mathcal{E}[i] \ \rangle \ \rangle$$

If $\varepsilon \rightsquigarrow \varepsilon'$, we say ε reduces *to ε'.*

The following result shows that $\rightsquigarrow$ is an extension of $\stackrel{*}{\rightarrow}$.

Theorem 5.41

$$\stackrel{*}{\rightarrow} \ \subsetneq \ \rightsquigarrow$$

Proof

To show $\stackrel{*}{\rightarrow} \subseteq \rightsquigarrow$, we note that for any ε and ε' such that $\varepsilon \stackrel{*}{\rightarrow} \varepsilon'$, the sequence $\langle i \ :\ 0 \leq i \ :\ \varepsilon'\langle 0 \ldots i \rangle \ \rangle$ is a CRS for ε'. To show $\stackrel{*}{\rightarrow} \neq \rightsquigarrow$, we refer to the example above, where we showed ε and ε' such that $\neg(\varepsilon \stackrel{*}{\rightarrow} \varepsilon')$. The finite sequential prefixes of the executions ε_i form a CRS for ε', so we have $\varepsilon \rightsquigarrow \varepsilon'$.
(End of proof)

5.5 Relating concurrent and sequential configurations

Before we proceed with the reduction rules, we first discuss what we can infer about configurations in ε from the properties of a sequential execution ε' such that $\varepsilon \rightsquigarrow \varepsilon'$. An important concept for stating these theorems is that of a thread being *uncommitted.*

Definition 5.42 *For* $\mathbf{C} \in PC$, $\alpha \in \mathbf{A}$, *and* $\mathbf{A}' \subseteq \mathbf{A}$,

$$
\begin{array}{rcl}
\mathit{Uncommitted}(\mathbf{C}, \alpha) & \triangleq & \langle \forall\ D \\
& & :\ D \in \mathbf{B}\ \wedge\ \mathit{Root}(\mathbf{C}, D) = \alpha \\
& & :\ \mathbf{C}.D.\phi \in \{\text{GUARD}, \text{PWAIT}, \text{REJECT}\} \\
& & \rangle \\
\mathit{Uncommitted}(\mathbf{C}, \mathbf{A}') & \triangleq & \langle \forall\ \alpha\ :\ \alpha \in \mathbf{A}'\ :\ \mathit{Uncommitted}(\mathbf{C}, \alpha)\ \rangle
\end{array}
$$

The idea of this definition is that $\mathit{Uncommitted}(\mathbf{C}, \alpha)$ holds in configuration $\mathbf{C}$ if there is no thread for α in $\mathbf{C}$ that has taken an accept decision step and is the code from the body of a procedure. The action of deciding to update the system state is called "committing" in transaction processing parlance, whence the name of this function. If $\mathit{Uncommitted}(\mathbf{C}, \alpha)$, then either there is no thread active for α in $\mathbf{C}$, or a thread that is active for α has yet to take a decision step, or it has decided to reject. If $\mathit{Uncommitted}(\mathbf{C}, \mathbf{A})$, then no action is the program has committed. This is clearly the case if $qt(\mathbf{C})$.

If $\mathit{Uncommitted}(\mathbf{C}, \alpha)$, then to reach a configuration where this does not hold requires an accept decision step, as the following lemma shows.

Lemma 5.43

$$
\begin{array}{l}
\langle \forall\ \varepsilon, \alpha \\
\quad :\ \varepsilon \in Z_m\ \wedge\ \alpha \in \mathbf{A}\ \wedge\ |\varepsilon| < \infty\ \wedge\ \mathit{NumA}(\varepsilon) = 0\ \wedge \\
\quad :\ \mathit{Uncommitted}(\mathit{Start}(\varepsilon), \alpha)\ \Rightarrow\ \mathit{Uncommitted}(\mathit{Final}(\varepsilon), \alpha) \\
\quad \rangle
\end{array}
$$

The theorem below gives us the first correspondence between configurations in an execution and its reduction. It says that if any execution block in the original execution contains a configuration where no action is committed, then the values for all the box variables in that configuration is the same as it is in the corresponding

accept configuration of the reduced execution.

Theorem 5.44

$$\begin{aligned}
&\langle \forall\ \varepsilon, \varepsilon', i, k \\
&\quad :\ \varepsilon, \varepsilon' \in Z_m \ \wedge\ \varepsilon \rightsquigarrow \varepsilon' \ \wedge\ 0 \le i < \mathit{NumA}(\varepsilon) \ \wedge \\
&\qquad 0 \le k < |EB(\varepsilon, i)| \ \wedge\ \mathit{Uncommitted}(\ EB(\varepsilon, i)[k]\ ,\ \mathbf{A}\ ) \\
&\quad :\ \mathit{PersistEq}(\ EB(\varepsilon, i)[k]\ ,\ AC(\varepsilon', i)\ ) \\
&\quad \rangle
\end{aligned}$$

Theorem 5.44 shows a close correspondence between the configurations in the original execution and those in the reduced execution. Note, in particular, that if the reduced execution ε' is atomic, then $AC(\varepsilon', i)$ is a quiescent configuration prior to the accepting thread i. From this, we get the following. Suppose we show that "q is invariant" holds for any sequential execution of a program. That is, in any sequential execution, all the quiescent configurations between threads satisfy predicate q. For any i, such $0 \le i \le |\varepsilon|$, and $\mathit{Uncommitted}(\varepsilon[i], \mathbf{A})$, we can conclude that q holds in $\varepsilon[i]$.

One way to use this result is to execute the concurrent program in a way that ensures that from time to time there is a quiescent configuration. In a quiescent configuration $\varepsilon[i]$, $\mathit{Uncommitted}(\varepsilon[i], \mathbf{A})$ holds trivially. If we examine the quiescent configurations in the concurrent execution, they satisfy all the invariance properties of the sequential executions.

We can also apply this result to certain progress properties. Suppose we show that in any sequential execution of a program that "q is unavoidable", by which we mean that q holds at some configuration in the execution, and, once q holds, it continues to hold for the remainder of the execution. If q is unavoidable, then in any execution there is a finite prefix where q does not hold, and a nonempty suffix where q holds in every configuration. Using Theorem 5.44, we claim that if a

concurrent execution contains an infinite number of quiescent configurations, then for all of these, except some finite prefix, q holds.

This is related to Valiant's suggestion for *Bulk Synchronous Processing.* In [32], he suggests a model for concurrent execution where processes (threads, in our terms) are run asynchronously for the most part (much as we model the execution of *TCB* programs), but, from time to time, the system is synchronized, meaning that all partly completed processes are allowed to complete execution, without starting any new processes, until the system has reached a quiescent configuration.

Theorem 5.44 gives a way of getting a consistent view of the whole state space of a program. For many purposes this is too much. Imagine a system with many boxes, and a predicate q, whose value depends a variables in just a small subset of the boxes. If we wish to test if q holds at a point in a concurrent execution, we must ensure that it is tested in a consistent configuration, that is, one that does not show the effects of a partly completed thread. One way to do this is to use the above observations, and to force the execution to become quiescent from time to time, and to test if q holds in each quiescent configuration. This means that all threads, even those that cannot possibly affect the value of q, must be ended before we can test for q. It seems that is should be enough to test for q in a configuration in which no thread that can possibly change the value of q is committed. This is indeed the case, as the next theorem shows.

The theorem concerns expressions defined across several boxes. We extend the expression evaluation operator to evaluate these expressions in a program configuration. For $\mathbf{C} \in PC$, we define

$$[\![D.x]\!]\mathbf{C} \quad \triangleq \quad [\![x]\!](\mathbf{C}.D.\sigma)$$

For operators and constants, $[\![e]\!]\mathbf{C}$ is defined similarly to $[\![e]\!]\sigma$.

Theorem 5.45 *If e is an expression in the box variables, and $\mathbf{A}'$ is the set of actions*

that call procedures that can change the value of e, then

$$
\begin{aligned}
&\langle \forall\ \varepsilon, \varepsilon', i, k \\
&\quad : \varepsilon, \varepsilon' \in Z_m \ \wedge\ \varepsilon \leadsto \varepsilon' \ \wedge\ 0 \leq i < NumA(\varepsilon) \ \wedge \\
&\quad\quad 0 \leq k < |EB(\varepsilon, i)| \ \wedge\ Uncommitted(\ EB(\varepsilon, i)[k]\ ,\ \mathbf{A}'\ ) \\
&\quad : [\![e]\!](\ EB(\varepsilon, i)[k]\ ) \ = \ [\![e]\!](\ AC(\varepsilon', i)\ ) \\
&\quad \rangle
\end{aligned}
$$

This theorem says that if $\varepsilon \leadsto \varepsilon'$, and there is a configuration in ε where no action that can change the value of e is committed, then the value of e in this configuration is the same as its value in the correponding accept configuration in ε'.

Theorem 5.45 suggests the following way of constructing programs that guarantee that a predicate that is unavoidable in any sequential execution is eventually observed to hold in a concurrent execution. Suppose

$$
q \ = \ D.x > 0 \ \wedge\ E.y = 3
$$

and suppose we show that in any sequential execution, q is unavoidable. We amend the program is by adding procedures to D and E, as shown in Figure 5.1. We show only the procedures added to the program. Here $D.testq$ is a partial action with a single alternative. The condition checks if the part of q local to D holds, and the test is a call to $E.testq$. This latter procedure accepts only if the part of q local to E holds. Action $D.testq$ accepts iff q holds.

If we run the amended program under a control relation that does not allow $D.testq$ to run concurrently with any action that can change the value of q, then if $D.testq$ accepts, we know that it has done so in a configuration in which none of these other actions is active.

We claim that the property "q is unavoidable" for the sequential executions of the original program corresponds to the property "eventually $D.testq$ accepts"

```
box D
        ⋮
  action testq  ::  x > 0 & E.testq  ⟶  (∗ ··· ∗)
        ⋮
end

box E
        ⋮
  method testq  ::  y = 3  ⟶  (∗ ··· ∗)
        ⋮
end
```

Figure 5.1: Adding a probe to a program

in any concurrent execution of the augmented program that respects the control relation.

We call an action such as $D.testq$ a *probe*. Obviously, the format we used above works only for predicates that can be expressed as conjunctions of local terms.

We can evaluate a general expression q in the following way. Suppose D_0, $D_1, \ldots, D_{n-1}$ are the boxes with variables in q. We write an action $D_0.testq$. This has a call to $D_1.testq$ as the test. The parameters for this call are the values of D_0's variables from q. Method $D_1.testq$ passes these parameters on to method $D_2.testq$, along with the values for D_1's variables in q. Continuing in this way, the values for all variables in q are passed to $D_{n-1}.testq$, which can evaluate q.

5.6 Reduction rules

We reduce an execution by performing a sequence of local transformations, each of which is one of the following.

- remove a **reject** step

- replace adjacent **queue** and **init** steps with a **rdv** step
- replace a compact execution of a procedure with an **atomic** step
- move a right-mover right
- move a left-mover left

We prove a set of theorems that give rules for applying applying these transformations. Each theorem gives conditions on finite execution ε when one of the above transformations can be applied to give ε' such that $\varepsilon \stackrel{*}{\to} \varepsilon'$. First we have the rule for removing **reject** steps.

Theorem 5.46 (Reject removal rule)

$$\begin{array}{l} \langle \forall\ \mathbf{C}, \alpha \\ \ \ :\ \mathbf{C} \in PC\ \wedge\ \alpha \in \mathbf{A}\ \wedge\ (\mathbf{C}, \mathbf{C}) \in \textsf{\textbf{reject}}(\alpha) \\ \ \ :\ (\ \langle\ \mathbf{C}\ ,\ \mathbf{C}\ \rangle\ ,\ \langle\ \textsf{\textbf{reject}}(\alpha)\ \rangle\ )\ \stackrel{*}{\to}\ (\ \langle\ \mathbf{C}\ \rangle\ ,\ \perp\ ) \\ \ \ \rangle \end{array}$$

Proof

Use Theorems 5.25 and 5.32.

(End of proof)

The next theorem shows the four cases when a **rdv** step can replace a **queue** and an **init** step. To state these theorems, we define a set of two-step executions, $TS(\mathbf{L}, \mathbf{L}')$, for every pair of labels $\mathbf{L}, \mathbf{L}' \in Lab_m$. This set differs from $TwoStep(\mathbf{L}, \mathbf{L}')$ in that it requires that the steps be from the *same* thread.

Theorem 5.47 (Rendezvous rule) *For* $\mathbf{L}, \mathbf{L}' \in Lab_m$, *let*

$$
\begin{aligned}
TS(\mathbf{L}, \mathbf{L}') \;\triangleq\; \{ \; & \varepsilon \\
| \; & \varepsilon \in Z_m \;\wedge\; |\varepsilon| = 2 \;\wedge\; Root(\varepsilon, 0) = Root(\varepsilon, 1) \;\wedge \\
& SSeq(\varepsilon) = \langle \mathbf{L}, \mathbf{L}' \rangle \\
\} &
\end{aligned}
$$

in

$$
\begin{aligned}
& \langle \forall \; D, \varepsilon \\
& \;\; : D \in \mathbf{B} \;\wedge\; \varepsilon \in TS(\, \textsf{\textbf{action-start}}(D) \,,\, \textsf{\textbf{p-action-init}}(D) \,) \\
& \;\; : \varepsilon \xrightarrow{*} (\; \langle\, Start(\varepsilon) \,,\, Final(\varepsilon) \,\rangle \;,\; \langle\, \textsf{\textbf{p-action-start-rdv}}(D) \,\rangle \;) \\
& \;\; \rangle
\end{aligned}
$$

$$
\begin{aligned}
& \langle \forall \; D, \varepsilon \\
& \;\; : D \in \mathbf{B} \;\wedge\; \varepsilon \in TS(\, \textsf{\textbf{action-start}}(D) \,,\, \textsf{\textbf{t-action-init}}(D) \,) \\
& \;\; : \varepsilon \xrightarrow{*} (\; \langle\, Start(\varepsilon) \,,\, Final(\varepsilon) \,\rangle \;,\; \langle\, \textsf{\textbf{t-action-start-rdv}}(D) \,\rangle \;) \\
& \;\; \rangle
\end{aligned}
$$

$$
\begin{aligned}
& \langle \forall \; D, E, \varepsilon \\
& \;\; : D, E \in \mathbf{B} \;\wedge\; D \neq E \;\wedge \\
& \;\;\;\;\; \varepsilon \in TS(\, \textsf{\textbf{guard-test}}(D, E) \,,\, \textsf{\textbf{p-method-init}}(E) \,) \\
& \;\; : \varepsilon \xrightarrow{*} (\; \langle\, Start(\varepsilon) \,,\, Final(\varepsilon) \,\rangle \;,\; \langle\, \textsf{\textbf{guard-test-rdv}}(D, E) \,\rangle \;) \\
& \;\; \rangle
\end{aligned}
$$

$$
\begin{array}{l}
\langle \forall \ D, E, \varepsilon \\
\quad : D, E \in \mathbf{B} \ \wedge \ D \neq E \ \wedge \\
\quad\quad \varepsilon \in \mathit{TS}(\ \textsf{\textbf{total-call}}(D,E)\ ,\ \textsf{\textbf{t-method-init}}(E)\ ) \\
\quad : \varepsilon \xrightarrow{*} (\ \langle\ \mathit{Start}(\varepsilon)\ ,\ \mathit{Final}(\varepsilon)\ \rangle\ ,\ \langle\ \textsf{\textbf{total-call-rdv}}(D,E)\ \rangle\ ) \\
\quad \rangle
\end{array}
$$

The next theorem covers the five cases for replacing a compact execution of a thread with an atomic step.

Theorem 5.48 (Atomic rule)

$$
\begin{array}{l}
\langle \forall \ \varepsilon, D, a \\
\quad : \varepsilon \in Z_m \ \wedge \ D \in \mathbf{B} \ \wedge \ a \in \mathit{TotActs}(D) \ \wedge \\
\quad\quad \varepsilon \ \mathit{is\ a\ compact\ execution\ for}\ D.a \ \wedge \ \mathit{Last}(\varepsilon) = \textsf{\textbf{action-end}}(D) \\
\quad : \varepsilon \xrightarrow{*} (\ \langle\ \mathit{Start}(\varepsilon)\ ,\ \mathit{Final}(\varepsilon)\ \rangle\ ,\ \langle\ \textsf{\textbf{accept}}(D.a)\ \rangle\ ) \\
\quad \rangle
\end{array}
$$

$$
\begin{array}{l}
\langle \forall \ \varepsilon, D, a \\
\quad : \varepsilon \in Z_m \ \wedge \ D \in \mathbf{B} \ \wedge \ a \in \mathit{TotActs}(D) \ \wedge \\
\quad\quad \varepsilon \ \mathit{is\ a\ compact\ execution\ for}\ D.a \ \wedge \ \mathit{Last}(\varepsilon) = \textsf{\textbf{action-reject}}(D) \\
\quad : \varepsilon \xrightarrow{*} (\ \langle\ \mathit{Start}(\varepsilon)\ ,\ \mathit{Final}(\varepsilon)\ \rangle\ ,\ \langle\ \textsf{\textbf{reject}}(D.a)\ \rangle\ ) \\
\quad \rangle
\end{array}
$$

$$
\begin{array}{l}
\langle \forall \ \varepsilon, D, E, m \\
\quad : \varepsilon \in Z_m \ \wedge \ D, E \in \mathbf{B} \ \wedge \ D \neq E \ \wedge \ m \in \mathit{PartMeths}(E) \ \wedge \\
\quad\quad \varepsilon \ \mathit{is\ a\ compact\ execution\ for}\ E.m \ \wedge \ \mathit{Last}(\varepsilon) = \textsf{\textbf{test-accept}}(D,E) \\
\quad : \varepsilon \xrightarrow{*} (\ \langle\ \mathit{Start}(\varepsilon)\ ,\ \mathit{Final}(\varepsilon)\ \rangle\ ,\ \langle\ \textsf{\textbf{pm-accept}}(D, E.m)\ \rangle\ ) \\
\quad \rangle
\end{array}
$$

$$
\begin{array}{l}
\langle \forall\ \varepsilon, \varepsilon', D, E, m \\
\quad : \varepsilon \in Z_m \ \wedge\ D, E \in \mathbf{B} \ \wedge\ D \neq E \ \wedge\ m \in \mathit{PartMeths}(E) \ \wedge \\
\qquad \varepsilon \ \mathit{is\ a\ compact\ execution\ for}\ E.m \ \wedge \\
\qquad \mathit{Last}(\varepsilon) = \textsf{\textbf{test-reject}}(D, E) \\
\quad : \varepsilon \xrightarrow{*} (\ \langle\ \mathit{Start}(\varepsilon)\ ,\ \mathit{Final}(\varepsilon)\ \rangle\ ,\ \langle\ \textsf{\textbf{pm-reject}}(D, E.m)\ \rangle\ ) \\
\quad \rangle
\end{array}
$$

$$
\begin{array}{l}
\langle \forall\ \varepsilon, \varepsilon', D, E, m \\
\quad : \varepsilon \in Z_m \ \wedge\ D, E \in \mathbf{B} \ \wedge\ D \neq E \ \wedge\ m \in \mathit{TotMeths}(E) \ \wedge \\
\qquad \varepsilon \ \mathit{is\ a\ compact\ execution\ for}\ E.m \ \wedge \\
\qquad \mathit{Last}(\varepsilon) = \textsf{\textbf{total-return}}(D, E) \\
\quad : \varepsilon \xrightarrow{*} (\ \langle\ \mathit{Start}(\varepsilon)\ ,\ \mathit{Final}(\varepsilon)\ \rangle\ ,\ \langle\ \textsf{\textbf{tm}}(D, E.m)\ \rangle\ ) \\
\quad \rangle
\end{array}
$$

Next we have the rule for right-movers. Any right-mover can move right, with a slight restriction on **queue** steps. Note that $\bot$ is not an execution, so there is no execution ε such that $\varepsilon \xrightarrow{*} \bot$. Thus, the term $\varepsilon \xrightarrow{*} \mathit{Swap}(\varepsilon)$ implies $\mathit{Swap}(\varepsilon) \neq \bot$.

Theorem 5.49 (Right-mover rule)

$$
\begin{array}{l}
\langle \forall\ \mathbf{L}, \mathbf{L}', \varepsilon \\
\quad : \mathbf{L}, \mathbf{L}' \in \mathit{Lab}_m \ \wedge\ \mathbf{L} \subseteq \textsf{\textbf{RM}} \ \wedge\ \varepsilon \in \mathit{TwoStep}(\mathbf{L}, \mathbf{L}') \ \wedge \\
\qquad (\ \mathbf{L} \subseteq \textsf{\textbf{queue}} \ \Rightarrow\ \mathit{ULoci}(\varepsilon\langle 0\rangle)\ \mathrm{disj}\ \mathit{ULoci}(\varepsilon\langle 1\rangle)\ ) \\
\quad : \varepsilon \xrightarrow{*} \mathit{Swap}(\varepsilon) \\
\quad \rangle
\end{array}
$$

The final theorem in this section gives the rule for moving left-movers left. It applies only to **local** and **end** steps, and not to **tm** steps.

Theorem 5.50 (Left-mover rule)

$$\begin{array}{l}\langle\forall\ \mathbf{L},\mathbf{L}',\varepsilon \\ \quad :\ \mathbf{L},\mathbf{L}'\in Lab_m\ \wedge\ \mathbf{L}'\subseteq \textsf{local}\cup\textsf{end}\ \wedge\ \varepsilon\in TwoStep(\mathbf{L},\mathbf{L}') \\ \quad :\ \varepsilon \stackrel{*}{\rightarrow} Swap(\varepsilon) \\ \quad \rangle\end{array}$$

We now have five reduction rules: the reject removal rule, the rendezvous rule, the atomic rule, the right-mover rule, and the left-mover rule. All these rules apply in any execution. So, for example, in any execution, if there is a **local** step following a **queue** step from a different thread, the **local** step can be moved left. Showing that this is possible involves showing that the steps *commute*, that is, the steps have the same effect on the configuration, regardless of the order in which they are taken.

5.7 Reduction rule for tm steps

Suppose ε is a complete execution in which no thread calls a total method. In this case, each thread has no right-movers after the decision step, only **local** and **end** steps. Thus each thread obeys the two-phase locking protocol. As we have seen, if all threads are two-phase, then we can serialize any execution. Threads for *TCB* are not two-phase because of the total method calls. In this section, we consider reduction rules for **tm** steps. We show the following rule for **tm** steps in any execution.

Theorem 5.51

$$\begin{aligned}
&\langle \forall\ \mathbf{L}, \mathbf{L}', \varepsilon \\
&\quad : \mathbf{L}, \mathbf{L}' \in Lab_m \ \wedge\ \mathbf{L} \not\subseteq \textsf{\textbf{atomic}} \cup \textsf{\textbf{end}} \ \wedge\ \mathbf{L}' \subseteq \textsf{\textbf{tm}} \ \wedge \\
&\quad\ \ \varepsilon \in TwoStep(\mathbf{L}, \mathbf{L}') \\
&\quad : \varepsilon \overset{*}{\rightarrow} Swap(\varepsilon) \\
&\quad \rangle
\end{aligned}$$

That is, in any execution a **tm** step can move left over any step that is not an **atomic** step, or an **end** step. If we organize the reduction carefully, we never need to move **tm** steps left over **end** steps. We consider here how to ensure that **tm** steps move left over **atomic** steps.

Consider the program in Figure 1.5. We saw that if there are two calls to $X.add$ from different threads, then these calls can happen in either order, and the final result is the same. The same is not true for a call to $X.add$ and a call to $X.mult$. In this case the final value of $X.x$ depends on the order in which the calls are made. Using the commutativity of $X.add$ with itself, we can reduce any concurrent execution of threads for $D.aa$ and $E.a$ to a sequential execution. For the concurrent execution of $D.aa$ and $E.b$ where the call to $X.mult$ from $E.b$ falls between the two calls to $X.add$ from $D.aa$ reaches a final configuration that is not reachable by executing the two actions sequentially in either order.

Since we cannot reduce an execution containing concurrent threads, one of which calls $X.mult$, and the other of which calls $X.add$, we exclude such executions from consideration. As with deadlock, and termination, we define a control relation to exclude the problematic executions.

The control relation we define here, called *weak compatibility*, and written $\sim$, is defined so that if $\alpha \sim \alpha'$, then there is sufficient commutativity between the procedures that may be called during a thread for α and procedures that may be

called during a thread for α'.

To decide what commutativity on procedures is sufficient for the reduction, we must first decide how to handle rejecting threads. The reject removal rule allows us to remove a **reject** step from an execution. All rejecting threads are two-phase, with **queue** and **init** steps before the decision step, and **end** steps after it. Because of the two-phase structure, we can reduce a rejecting thread to an atomic step in any execution. We can then apply the reject removal rule to remove the thread from the execution.

If we apply this procedure to every rejecting thread in the execution, we never have to move a **tm** step over a rejecting atomic step. This is the approach we take in this chapter. Essentially, we are pretending that if a thread rejects, then we can pretend that the thread was never executed at all. This is the interpretation of rejecting threads given in [25], where a rejecting thread is represented by the empty relation, which corresponds to the empty set of executions.

In Chapter 6 we consider the issue of fairness between action executions, which concerns ensuring that threads for an action are started "often enough" during an execution to ensure progress properties. We show that removing some or all of the rejected threads from an execution can result in an execution that meets a weaker fairness condition than the original. Thus, from the standpoint of fairness, rejected threads *are* computationally significant. We show in that chapter how to reduce an execution so that every thread, accepting and rejecting, from the original execution appears in the reduced execution, and thus the reduced execution meets the same fairness condition as the original execution.

There is a trade-off involved here, as there usually is. To get the stronger reduction that preserves fairness, we need stronger commutativity conditions than we do for the reduction than we present in this chapter. The stronger commutativity conditions mean that fewer pairs of actions can be run concurrently, and so we trade

concurrency for fairness.

5.7.1 Weak compatibility

We now define the weak compatibility control relation that allows us to complete the set of rules we use to show the reduction of a *TCB* execution. We define this in terms of a relation between a procedure and a total method.

Definition 5.52 *For $\pi \in \mathbf{P}$, and $\mu \in \mathbf{M}_t$,*

$$
\begin{array}{rcl}
\pi \;\; wlc \;\; \mu & \triangleq & \langle \forall \; D, E \\
& & \quad : D, E \in \mathbf{B} \;\; \wedge \;\; D \neq E \\
& & \quad : (\pi \in \mathbf{A} \quad \Rightarrow \quad \textsf{\textbf{accept}}(\pi) \curvearrowleft \textsf{\textbf{tm}}(D, \mu)) \;\; \wedge \\
& & \quad \;\; (\pi \in \mathbf{M}_p \quad \Rightarrow \quad \textsf{\textbf{pm-accept}}(E, \pi) \curvearrowleft \textsf{\textbf{tm}}(D, \mu)) \;\; \wedge \\
& & \quad \;\; (\pi \in \mathbf{M}_t \quad \Rightarrow \quad \textsf{\textbf{tm}}(E, \pi) \curvearrowleft \textsf{\textbf{tm}}(D, \mu)) \\
& & \quad \rangle
\end{array}
$$

If π wlc μ, we say that π weakly left-commutes *with μ.*

Thus if $\pi \;\; wlc \;\; \mu$, then a **tm** step for μ can move left over an **atomic** step for π, provided that the step for π is not rejecting. The relation does not use the rejecting **atomic** steps, since during reduction, as discussed above, we can use the reject removal rule to remove these from the execution, so no **tm** step moves left over a **reject** or **pm-reject** step.

We give some examples of weak left commutativity. The first examples concern the box *Sem* in Figure 2.3.

$$
\begin{array}{lll}
Sem.P & wlc & Sem.V \\
Sem.V & wlc & Sem.V
\end{array}
$$

To show $Sem.P$ *wlc* $Sem.V$, we must show for any $D, E \in \mathbf{B}$, such $D \neq E$,

$$\textsf{\textbf{pm-accept}}(E, Sem.P) \quad \curvearrowleft \quad \textsf{\textbf{tm}}(D, Sem.V)$$

That is, we must show that, for any configuration in which a **pm-accept**$(E, Sem.P)$ step is enabled, the configurations reachable by executing this step followed by a **tm**$(D, Sem.V)$ step, are reachable by executing the steps in the opposite order.

Consider a configuration $\mathbf{C}$ in which **pm-accept**$(E, Sem.P)$ is enabled. We have $\mathbf{C}.Sem.\gamma = \bot$, from Theorem 5.5. From the guard of $Sem.P$, we also have that the semaphore is available in $\mathbf{C}$, that is, $\mathbf{C}.Sem.\sigma.n > 0$. From such a configuration, the call to $Sem.P$ followed by the call to $Sem.V$ leaves the value of $\mathbf{C}.Sem$ unchanged, and updates the configurations of D and E to reflect the fact that the procedure calls have been executed. Executing the calls in the reversed order has exactly the same effect, and thus we have weak left commutativity.

To show $Sem.V$ *wlc* $Sem.V$, we must show for any $D, E \in \mathbf{B}$, such that $D \neq E$,

$$\textsf{\textbf{tm}}(E, Sem.V) \quad \curvearrowleft \quad \textsf{\textbf{tm}}(D, Sem.V)$$

Since $Sem.V$ is total, the steps are enabled in any configuration. The effect of executing the procedure calls in either order is to add 2 to $Sem.n$, and to update the configurations of D and E to reflect the execution of the calls. Thus we have weak left commutativity.

Below are some examples of left commutativity using the box *Buff* from Figure 1.2.

$$Buff.get \quad wlc \quad Buff.put$$

$$\neg(\ Buff.put \quad wlc \quad Buff.put\ )$$

We show $\mathit{Buff.get}\ \ \mathit{wlc}\ \ \mathit{Buff.put}$ in a way similar to that we used for $\mathit{Sem.P}$ and $\mathit{Sem.V}$, above. We note that an accepting call to $\mathit{Buff.get}$ is only enabled in a configuration in which $\mathit{Buff.s}$ is not empty. Suppose $\mathit{Buff.s} = x \triangleright \hat{s}$. If the call to $\mathit{Buff.put}$ puts the value y in the queue, we can see that, executing the calls in either order, we end up with $\mathit{Buff.s} = \hat{s} \triangleleft y$, and x is returned to the caller of $\mathit{Buff.get}$. Thus we have weak left commutativity.

To show $\neg(\mathit{Buff.put}\ \ \mathit{wlc}\ \ \mathit{Buff.put})$, we observe that calling $\mathit{Buff.put}$ with input parameter x, and then calling it from a different box with input parameter y leaves $\mathit{Buff.s} = \hat{s} \triangleleft x \triangleleft y$, whereas making these calls in the opposite order gives $\mathit{Buff.s} = \hat{s} \triangleleft y \triangleleft x$. Clearly $\hat{s} \triangleleft x \triangleleft y \neq \hat{s} \triangleleft y \triangleleft x$, unless $x = y$. Thus, we do not have weak left commutativity.

The following theorem gives a simple condition for determining weak left commutativity: a procedure weakly left commutes with a total method if they execute on disjoint sets of boxes.

Theorem 5.53

$$
\begin{array}{l}
\langle \forall\ \pi, \mu \\
\quad :\ \pi \in \mathbf{P}\ \ \wedge\ \ \mu \in \mathbf{M}_t\ \ \wedge \\
\quad\ \ \langle \forall\ \pi', \mu'\ \ :\ \ \pi' \in \mathit{Range}(\pi)\ \ \wedge\ \ \mu' \in \mathit{Range}(\mu)\ \ :\ \ \neg(\pi' \boxminus \mu')\ \rangle \\
\quad :\ \pi\ \ \mathit{wlc}\ \ \mu \\
\quad \rangle
\end{array}
$$

We can now define weak compatibility.

Definition 5.54 (Weak compatibility) *For* $\alpha, \alpha' \in \mathbf{A}$,

$$\begin{aligned}\alpha \sim \alpha' \triangleq\ & \langle \forall\ \pi, \pi' \\ & : \pi \in \mathit{Range}(\alpha)\ \wedge\ \pi' \in \mathit{Range}(\alpha') \\ & : (\pi \in \mathbf{M}_t\ \Rightarrow\ \pi'\ \mathit{wlc}\ \pi)\ \wedge\ (\pi' \in \mathbf{M}_t\ \Rightarrow\ \pi\ \mathit{wlc}\ \pi') \\ & \rangle\end{aligned}$$

If $\alpha \sim \alpha'$, *we say that* α *is* weakly compatible *with* α'.

The symmetric form of the definition guarantees that $\sim$ is symmetric, and thus $\sim\ \in \mathit{CR}$. The definition of weak compatibility gives exactly the conditions needed to define a left-moving rule for total method calls. Note that the definition does not require any commutativity between the actions α and α'.

From the results given above for *wlc* on the methods of box *Sem* in Figure 2.3, we get the following.

$$\begin{aligned} D.act &\sim E.acq \\ D.act &\sim E.rel \end{aligned}$$

For the actions in Figure 1.3 that use the methods of box *Buff* in Figure 1.2, we have

$$\mathit{Prod.make} \sim \mathit{Cons.use}$$

Suppose we have α and α', where

$$\begin{aligned} \mathit{Range}(\alpha)\ &\text{disj}\ \mathbf{M}_t \\ \mathit{Range}(\alpha')\ &\text{disj}\ \mathbf{M}_t \end{aligned}$$

In this case, the conditions for $\alpha \sim \alpha'$ are trivially satisfied. These conditions mean that no total methods are reachable from α or α' by $\succ$, or, in other words, that

only partial methods are called during the execution of any thread for either action. If α is total, this means no thread for α calls a method, since only total methods can be called from a total action. If α is partial, we have that a thread for α calls only partial methods that do not call total methods. Since there can be at most one call to a partial method in any alternative in a partial procedure, this gives us a two-phase structure for all the threads for α, with the decision step in the lowest partial procedure call, all resource acquisition steps before the decision step, and all resource release steps after. For pairs of such threads, the two-phase locking theorem gives us serializability under any control relation.

If we restrict attention to executions that respect weak compatibility, we get the following rule for moving a **tm** step left.

Theorem 5.55 (Weak compatibility rule)

$$
\begin{array}{l}
\langle \forall\ \mathbf{L}, \mathbf{L}', \varepsilon \\
\quad : \mathbf{L}, \mathbf{L}' \in Lab_m \ \wedge\ \mathbf{L} \not\subseteq \textsf{\textbf{REJ}} \cup \textsf{\textbf{end}} \ \wedge\ \mathbf{L}' \subseteq \textsf{\textbf{tm}} \ \wedge \\
\quad\quad \varepsilon \in \mathit{TwoStep}(\mathbf{L}, \mathbf{L}') \ \wedge\ \varepsilon\ \mathit{resp}\ \sim \\
\quad : \varepsilon \overset{*}{\rightarrow} \mathit{Swap}(\varepsilon) \\
\quad \rangle
\end{array}
$$

5.7.2 Reduction that respects control relations

For reduction, we restrict ourselves to TCB executions that respect $\sim$. This allows us to use the weak compatibility rule to move **tm** steps left. It is important, then, that we ensure that every transformation that we apply to an execution ε such that $\varepsilon\ \mathit{resp}\ \sim$ returns an execution ε' such that $\varepsilon'\ \mathit{resp}\ \sim$.

Considering this question more generally, for any $\mathbf{T} \in \mathit{CR}$, if $\varepsilon\ \mathit{resp}\ \mathbf{T}$ and we apply one of the reduction rules to part of ε, yielding the reduced execution ε', does $\varepsilon'\ \mathit{resp}\ \mathbf{T}$? We have the following meta-theorem. The proof is in Appendix B.

Theorem 5.56

$$\begin{array}{l} \langle \forall\ \varepsilon, \varepsilon', \mathbf{T} \\ \quad :\ \varepsilon, \varepsilon' \in Z_m\ \wedge\ \varepsilon\ resp\ \mathbf{T}\ \wedge \\ \qquad \varepsilon \xrightarrow{*} \varepsilon'\ \textit{by Theorem 5.46, 5.47, 5.48, 5.49, 5.50, or 5.55} \\ \quad :\ \varepsilon'\ resp\ \mathbf{T} \\ \quad \rangle \end{array}$$

5.8 The first reduction theorem

We are now ready to state and prove the first reduction theorem. This theorems says that we can reduce any complete execution that respects $\sim$ to an atomic execution.

Theorem 5.57 (First reduction theorem)

$$\langle \forall\, \varepsilon\ :\ \varepsilon \in Complete(Z)\ \wedge\ \varepsilon\ resp\ \sim\ :\ \langle \exists\, \varepsilon'\ :\ \varepsilon' \in Z_a\ :\ \varepsilon \leadsto \varepsilon'\ \rangle\ \rangle$$

5.8.1 Outline of the proof

We prove the theorem using the following lemma. Note that we state the lemma for executions in $Complete(Z_m)$, rather than for $Complete(Z)$. The latter is a subset of the former, so this is a generalization that allows us to use mixed executions for intermediate results during execution.

Lemma 5.58

$$
\begin{array}{l}
\langle \forall\ \varepsilon \\
\quad :\ \varepsilon \in \mathit{Complete}(Z_m)\ \wedge\ \mathit{NumA}(\varepsilon) > 0\ \wedge\ \varepsilon\ \mathit{resp}\ \sim \\
\quad :\ \langle \exists\ \varepsilon', \varepsilon'' \\
\qquad :\ \varepsilon' \in Z_a\ \wedge\ \varepsilon'' \in \mathit{Complete}(Z_m)\ \wedge\ \mathit{NumA}(\varepsilon') = 1 \\
\qquad :\ \varepsilon \stackrel{*}{\rightarrow} \varepsilon'; \varepsilon''\ \wedge\ \varepsilon'; \varepsilon''\ \mathit{resp}\ \sim \\
\qquad \rangle \\
\quad \rangle
\end{array}
$$

We show the proof for this lemma below. Given Lemma 5.58 we complete the proof of Theorem 5.57 as follows.

Given $\varepsilon \in \mathit{Complete}(Z_m)$, we construct a sequence of pairs of executions of the following form.

$$
\begin{array}{ll}
\varepsilon_0 & \varepsilon'_0 \\
\varepsilon_0; \varepsilon_1 & \varepsilon'_1 \\
\varepsilon_0; \varepsilon_1; \varepsilon_2 & \varepsilon'_2 \\
\vdots & \vdots
\end{array}
$$

Each entry in the left column is a sequence of atomic executions, and each entry in the right column is an execution in $\mathit{Complete}(Z_m)$. We construct ε_0 and ε'_0 from ε using Lemma 5.58. For $i \geq 0$, we construct ε_{i+1} and ε'_{i+1} from ε'_i by the same lemma. If $\mathit{NumA}(\varepsilon) < \infty$ this process ends after a finite number of steps. In this case, we make the sequences infinite by repeating the last line.

The first column is a chain of finite executions in $(Z_a, \sqsubseteq)$. Since this ordered set is a CPO, the chain has a limit in Z_a. Let ε' be the limit.

The first column is also a CRS for the sequential execution, since for each entry, the corresponding execution in the second column is the witness required for

showing initial reduction. Each successive entry in the first column has one more accept step than the previous one, up to $NumA(\varepsilon)$, by Lemma 5.58. This gives us $\varepsilon \rightsquigarrow \varepsilon'$ and $\varepsilon' \in Z_a$, and the proof is complete.

One way to think of this proof is that we start with the execution ε, and we place a marker at the beginning. We reorganize the beginning of the ε, until there is a single **accept** step next to the marker. Then we move the marker over this step, and repeat the process with the remainder of the execution. As the marker moves through the execution in this way, the steps of the atomic execution appear in order on its left.

5.8.2 Proof of Lemma 5.58

We assume

$$\varepsilon \in \mathit{Complete}(Z_m) \ \wedge \ \mathit{NumA}(\varepsilon) > 0 \ \wedge \ \varepsilon \ \mathit{resp} \ \sim$$

we construct ε' and ε'', where

$$\varepsilon' \in Z_a \ \wedge \ \varepsilon'' \in \mathit{Complete}(Z_m) \ \wedge \ \mathit{NumA}(\varepsilon') = 1 \ \wedge$$
$$\varepsilon \xrightarrow{*} \varepsilon'; \varepsilon'' \ \wedge \ \varepsilon'; \varepsilon'' \ \mathit{resp} \ \sim$$

We do this by constructing a sequence of pairs $(\varepsilon_i, \varepsilon'_i)$, where each pair satisfies certain conditions. For clarity in the exposition, we use ψ as a base for names for executions as we present the construction of $(\varepsilon_{i+1}, \varepsilon'_{i+1})$ from $(\varepsilon_i, \varepsilon'_i)$. Let **D** be the first accept step in ε. Let **F** be the last step for **D**'s thread. Thus, if this thread is for action α, **D** is the first **ACC** step, and **F** is the first **action-end**$(Box(\alpha))$ step, or the first **accept**(α) step. Note that in the second case, **D** and **F** are the same step.

We use the following predicates on executions.

$$
\begin{array}{lll}
Imm(\varepsilon) & \triangleq & \langle \forall\ k \\
& & \quad :\ 0 \le k < |\varepsilon| \ \wedge\ \varepsilon\langle k\rangle \subseteq \textsf{\textbf{DEC}} \\
& & \quad :\ \langle \exists\ j \\
& & \qquad :\ 0 \le j < k \\
& & \qquad :\ \langle \forall\ i\ :\ 0 \le i < j\ :\ Root(\varepsilon, i) \ne Root(\varepsilon, k)\ \rangle\ \wedge \\
& & \qquad\ \ \langle \forall\ i\ :\ j \le i < k\ :\ Root(\varepsilon, i) = Root(\varepsilon, k)\ \rangle \\
& & \qquad \rangle \\
& & \quad \rangle \\
Def(\varepsilon) & \triangleq & \langle \forall\ k \\
& & \quad :\ 0 \le k < |\varepsilon| \ \wedge\ \varepsilon\langle k\rangle \subseteq \textsf{\textbf{RM}} \\
& & \quad :\ \langle \exists\ j \\
& & \qquad :\ k \le j < |\varepsilon| \\
& & \qquad :\ Root(\varepsilon, j) = Root(\varepsilon, k)\ \wedge\ \varepsilon\langle k\rangle \not\subseteq \textsf{\textbf{RM}} \\
& & \qquad \rangle \\
& & \quad \rangle
\end{array}
$$

If $Imm(\varepsilon)$ holds, then for every thread with a decision step in ε, all steps in ε for the thread up to and including the decision step appear as a contiguous sequence. If $qt(Start(\varepsilon))$, then this mean for every thread with a decision step in ε, all steps from the first step of the thread up to and including the decision step appear as a contiguous sequence. We call such an execution *immediate*.

If $Def(\varepsilon)$ holds then the last step for every thread with steps in ε is not an **RM** step. In particular, since all steps before the decision step are **RM** steps, if $qt(Start(\varepsilon))$, then this means that every thread with steps in ε has a decision step in ε. We call such an execution *definite*.

If $Imm(\varepsilon)$, $Def(\varepsilon)$ and $qt(Start(\varepsilon))$ hold, then every thread in ε begins with

an unbroken sequence of steps ending with a decision step.

Rendezvous reduction

We reduce ε to an execution in which all **RM** steps before **F** are **rdv** steps. Let

$$\varepsilon \quad = \quad \psi_0; \psi_0' \quad \text{where } \mathit{Last}(\psi_0) = \mathbf{F}$$

Choose a box $D \in \mathbf{B}$. Let $\mathbf{L}$ be the last **queue** step before **F** in ψ_0 that has D as its unconditional locus. Step **F** cannot be the next step for **L**'s thread, since a **init** step follows every **queue** step. By construction, no step from **L** up to and including **F** has D as its unconditional locus, so we use the right-mover rule to move **L** right until it reaches the position after **F**, or the position before the **init** step that is next in its thread, whichever is reached first. In the latter case, apply the rendezvous rule to replace the **queue** and **init** steps with a **rdv** step. Note that the new step does not have an unconditional locus. Repeat this procedure for every other **queue** step in ψ_0 that has D as its unconditional locus. Then repeat the whole process for every other box. At the end of this, we have execution ψ_1, where $\psi_0 \xrightarrow{*} \psi_1$ and ψ_1 contains no **queue** steps before **F**. There are no **init** steps before **F**, since $qt(\mathit{Start}(\psi_1))$, and every **init** step is preceded by a **queue** step. Thus all **RM** steps before **F** are **rdv** steps. Let

$$\begin{aligned} \psi_1 &= \varepsilon_0; \psi_1' \quad \text{where } \mathit{Last}(\varepsilon_0) = \mathbf{F} \\ \varepsilon_0' &= \psi_1'; \psi_0' \end{aligned}$$

We have $(\varepsilon_0, \varepsilon_0')$, where

$$\begin{aligned} &\varepsilon_0 \in Z_r \ \wedge \ |\varepsilon_0| < \infty \ \wedge \ \mathit{Last}(\varepsilon_0) = \mathbf{F} \ \wedge \\ &\varepsilon_0; \varepsilon_0' \in \mathit{Complete}(Z_m) \ \wedge \ \varepsilon \xrightarrow{*} \varepsilon_0; \varepsilon_0' \ \wedge \ \varepsilon_0; \varepsilon_0' \ \mathit{resp} \ \sim \end{aligned}$$

Right-mover reduction

We construct a reduction of ε where the execution is immediate up to **F**, and every thread with steps before **F** has a decision step before **F**. Let **L** be any decision step in ε_0. Choose the last step from **L**'s thread before it in ε_0. This is a **rdv** step (since $\varepsilon \in Z_r$, so use the right-mover rule to bring it adjacent to **L**. Repeat this with the remaining steps from **L**'s thread to its left, and then repeat with all other decision steps in ε_0, giving execution ψ_0, where $\varepsilon_0 \xrightarrow{*} \psi_0$. We have $Imm(\psi_0)$ by construction. Now choose a thread whose last step before **F** is in **RM**. Use the right-mover rule to move it to the right of **F**. Repeat this process until there are no such threads, giving execution ψ_1, where $\varepsilon_0 \xrightarrow{*} \psi_0$. Let

$$
\begin{aligned}
\psi_1 &= \varepsilon_1; \varepsilon_1' \quad \text{where } Last(\varepsilon_1) = \mathbf{F} \\
\varepsilon_1' &= \varepsilon_1'; \varepsilon_0'
\end{aligned}
$$

We have $(\varepsilon_1, \varepsilon_1')$, where

$$
\begin{aligned}
&\varepsilon_1 \in Z_r \ \wedge \ |\varepsilon_1| < \infty \ \wedge \ Imm(\varepsilon_1) \ \wedge \ Def(\varepsilon_1) \ \wedge \ Last(\varepsilon_1) = \mathbf{F} \ \wedge \\
&\varepsilon_1; \varepsilon_1' \in Complete(Z_m) \ \wedge \ \varepsilon \xrightarrow{*} \varepsilon_1; \varepsilon_1' \ \wedge \ \varepsilon_1; \varepsilon_1' \ resp \ \sim
\end{aligned}
$$

Reject removal

We construct a reduction of ε in which there are no **REJ** steps before **F**. Let

$$
\psi_0 = \varepsilon_1; \varepsilon_1'
$$

If ψ_0 contains no **REJ** steps before **F** then we are done. Otherwise, let **L** be one of the **REJ** steps before **F**. We reduce the steps for **L**'s thread to a **reject** step. If **L** is atomic we are done. Otherwise, since $Imm(\varepsilon_1)$, the steps before **L** in its thread appear to it immediate left. By Theorem 5.19, all steps after **L** in its thread are **action-reject** or **test-reject** steps. By the left-mover rule, we can move all of

these steps left over steps from other threads until all steps for **L**'s thread are then contiguous. By the atomic rule, these steps can be replaced by a single **reject** step. By the reject removal rule, we can remove the **reject** step from the execution. Repeat this process for every **REJ** step before **F**, ending with execution ψ_1, where $\psi \xrightarrow{*} \psi'$, and ψ_1 contains no **REJ** steps before **F**. Let

$$\begin{array}{rcl} \psi_1 & = & \varepsilon_2; \psi_1' \quad \text{where } \mathit{Last}(\varepsilon_1) = \mathbf{F} \\ \varepsilon_2' & = & \psi_1'; \varepsilon_1' \end{array}$$

We have $(\varepsilon_2, \varepsilon_2')$, where

$$\begin{array}{l} \varepsilon_2 \in Z_r \ \wedge \ |\varepsilon_2| < \infty \ \wedge \ \mathit{Last}(\varepsilon_2) = \mathbf{F} \ \wedge \\ \mathit{Imm}(\varepsilon_2) \ \wedge \ \mathit{Def}(\varepsilon_2) \ \wedge \\ \langle \forall\, i \ : \ 0 \leq i < |\varepsilon_2| \ : \ \varepsilon_2\langle i \rangle \not\subseteq \textsf{REJ} \ \rangle \ \wedge \\ \varepsilon_2; \varepsilon_2' \in \mathit{Complete}(Z_m) \ \wedge \ \varepsilon \xrightarrow{*} \varepsilon_2; \varepsilon_2' \ \wedge \ \varepsilon_2; \varepsilon_2' \ \mathit{resp} \ \sim \end{array}$$

Left-mover reduction

We construct a reduction of ε in which has an **accept** step as its first step. First we have a lemma, which gives the core of the left-mover reduction.

Lemma 5.59 *Let*

$$\begin{array}{rcl} N(\psi) & = & \langle \forall\, i \ : \ 0 \leq i < |\psi| \ : \ \psi\langle i \rangle \not\subseteq \textsf{REJ} \ \rangle \\ \#\textsf{end}(\psi) & = & \langle \#\, i \ : \ 0 \leq i < |\psi| \ : \ \psi\langle k \rangle \ \subseteq \ \textsf{end} \ \rangle \\ \#\textsf{comp}(\psi) & = & \langle \#\ i \\ & & \quad : \ 0 \leq i < |\psi| \\ & & \quad : \ \psi\langle k \rangle \ \subseteq \ \textsf{accept} \cup \textsf{action-end} \cup \textsf{action-reject} \\ & & \quad \rangle \end{array}$$

$$
\begin{array}{l}
\langle \forall\ \psi \\
\quad :\ \psi \in Z_r\ \ \wedge\ \ |\psi| < \infty\ \ \wedge\ \ qt(Start(\psi))\ \wedge \\
\quad\ \ Imm(\psi)\ \ \wedge\ \ Def(\psi)\ \ \wedge\ \ N(\psi)\ \wedge \\
\quad\ \ 0 < \#\textbf{end}(\psi) \\
\quad :\ \langle \exists\ \psi' \\
\qquad :\ \psi' \in Z_r\ \ \wedge\ \ |\psi'| < \infty\ \ \wedge\ \ qt(Start(\psi))\ \wedge \\
\qquad\ \ Imm(\psi')\ \ \wedge\ \ Def(\psi')\ \ \wedge\ \ N(\psi')\ \wedge \\
\qquad\ \ \#\textbf{end}(\psi') < \#\textbf{end}(\psi) \\
\qquad :\ \psi \stackrel{*}{\to} \psi'\ \ \wedge\ \ \#\textbf{comp}(\psi) = \#\textbf{comp}(\psi') \\
\qquad \rangle \\
\quad \rangle
\end{array}
$$

Proof

Assume we have a ψ as in the antecedent of the universal quantification. Let **L** be the first **end** step in ψ_2. Since $qt(Start(\psi))$, we can find step **L**$'$, the step that acquired the box that **L** releases. Since $\psi \in Z_r$, **L**$'$ is a **rdv** step, it is the only **RM** step for the procedure call, so it is the first step of a compact execution of the procedure. Let ψ_0 be the segment of ψ from the **L**$'$ to **L**. We have $\#\textbf{end}(\psi_0) = 1$, by construction. Consider the steps in ψ_0 from **L**'s thread. There are no **end** steps among these steps, and so, since resources are released in reverse order of acquisition, there are no **RM** steps either. If execution is for a partial procedure, we have, since $Imm(\psi)$ and $N(\psi)$, $\psi_0\langle 1\rangle \subseteq$ **ACC**. The steps for the procedure call thread in the region from $\psi_0\langle 2\rangle$ on, for a partial procedure, and from $\psi_0\langle 1\rangle$ on, for a total procedure, are after the thread's decision step. Since there are no **RM** steps, and no **end** steps, by Theorem 5.18, all the steps in this region for the procedure call's thread are thus **local** or **tm** steps. By the left-mover rule, the **local** steps can be moved left. There are no **REJ** or **end** steps between **L**$'$ and **L** from any thread, so, by the weak compatibility rule, the **tm** steps can be moved left. By the left-mover rule, **L** can

be moved left. Thus we move all these steps left until all steps for the procedure call are contiguous at the beginning of the execution. The reduced execution starts with a compact execution for a procedure, so we use the atomic rule to replace these steps with an **atomic** step, giving ψ_1, where $\psi_0 \xrightarrow{*} \psi_1$, $\#\textbf{end}(\psi_1) = 0$, and $\#\textbf{comp}(\psi_1) = \#\textbf{comp}(\psi_0)$.

Let ψ' be ψ with ψ_0 replaced by ψ_1. We have $\psi' \in Z_r$, $|\psi'| < \infty$, $qt(Start(\psi))$, $N(\psi')$, and $\#\textbf{end}(\psi') < \#\textbf{end}(\psi)$, $\psi \xrightarrow{*} \psi'$, and $\#\textbf{comp}(\psi) = \#\textbf{comp}(\psi')$, by construction. To show $Imm(\psi')$, we note that ε' contains all the steps of ε, in the same order, for the threads other than the one reduced. For the reduced thread, all steps for a procedure call have been deleted, and the **rdv** step that starts the call has been replaced by an **atomic** step. When a partial procedure is replaced, the **RM** step removed is replaced by a decision step, so immediacy is preserved. There is nothing to show for a total action, since there are no **RM** steps before the **t-action-start-rdv** step that is its decision step. To show $Def(\psi')$, we note that the reduction removes a **RM** step and some non**RM** steps, and replaces it with a single non-**RM** step for the thread. Thus $Def(\psi')$ must hold.
(End of proof)

To show the main reduction, we note that by the principle of induction, we can apply the reduction in Lemma 5.59 repeatedly to ε_2 until we get ψ, where $\#\textbf{end}(\psi) = 0$, and $\#\textbf{comp}(\psi) = \#\textbf{comp}(\varepsilon_2)$. Since the ψ contains the **end** step for the first thread, this thread must be a single **accept** step. Since we never move a **RM** step left over an **ACC** step, this step is $First(\psi)$. Let

$$\begin{aligned} \psi &= \varepsilon_3; \psi' \quad \text{where } |\varepsilon_3| = 1 \\ \varepsilon_3 &= \psi'; \varepsilon_2' \end{aligned}$$

We have $(\varepsilon_3, \varepsilon_3')$, where

$$\begin{aligned}
&\varepsilon_3 \in Z_a \;\wedge\; |\varepsilon_3| = 1 \;\wedge\; NumA(\varepsilon_3) = 1 \;\wedge \\
&\varepsilon_3; \varepsilon_3' \in Complete(Z_m) \;\wedge\; \varepsilon \xrightarrow{*} \varepsilon_3; \varepsilon_3' \;\wedge\; \varepsilon_3; \varepsilon_3' \; resp \; \sim
\end{aligned}$$

Let $\varepsilon' = \varepsilon_3$ and $\varepsilon'' = \varepsilon_3'$, and we are done with the proof of Lemma 5.58.

Chapter 6

Fairness

6.1 Introduction

The last chapter showed a control relation $\sim$ that guarantees a reduction to an atomic execution for every complete execution respecting $\sim$. We showed a correspondence between configurations in the original execution and the reduced atomic execution, which allow us to infer properties of the concurrent execution from those of the atomic execution.

The purpose of a reduction theorem is to show that a set of concurrent executions can be represented by a set of atomic executions, such that the sequential executions cover all possible behaviours of the concurrent executions. In this way, we avoid having to reason directly about the behaviour of the concurrent executions, and can instead reason about the behaviour of just the atomic executions. For this inference to be valid, we must precisely define the set of atomic executions that is generated when we reduce a given set of concurrent executions.

In this chapter, we consider properties of atomic executions, and we show that certain types of properties, called *progress properties*, are not guaranteed in all atomic executions, but are guaranteed in executions satisfying certain fairness

conditions. We then consider the subset of the concurrent executions whose atomic reductions satisfy a given fairness condition.

Consider a program containing an action α that increments the value of integer variable x, and suppose no other action in the program affects the value of x. What can we say about the sequence of values taken by x during an execution? We know that it never decreases, since its value is only changed by increasing it. What we cannot show, for general atomic executions, is that the value of x increases in every execution. Consider an atomic execution that contains no steps for α. The value of x does not change during this execution.

Consider an infinite atomic execution of the program containing an infinite number of steps for α. In such an execution, we can see that the value of x increases without bound. That is, for any configuration in the program, there is a later configuration in which x has a greater value.

In Chapter 4, we showed that fairness in the choice of steps in an execution is required to ensure an execution where every thread terminates. In a similar way we define fairness conditions on the sequence of actions for the steps in an atomic execution that can ensure that executions have desirable properties. This is fairness at a higher level than thread fairness, which concerns the choice of queue semantics steps. We are now concerned with the choice of *actions*. As with thread-fairness, the fairness conditions in this chapter are of relevance only for infinite executions.

In an atomic execution, every configuration is quiescent, and so, for every $\alpha \in \mathbf{A}$, a step for α is enabled, either an **accept**(α) step or, if α is partial, an **reject**(α) step. Note that the queue semantics steps before the decision step are deterministic, so each action has exactly one step enabled in each configuration.

The first fairness condition for actions is *weak fairness*. An infinite atomic execution ε is weakly fair for an action α if ε contains an infinite number of steps for α. For the example action above, if ε is weakly fair for α, then the value of x

increases without bound in ε.

Weak fairness guarantees *absence of individual starvation* for an action. In the standard analogy used for this type of problem, an action is always hungry for a step. An action starves if it waits an infinite time to take a step, where time is measured in steps. Requiring an infinite number of steps for α is equivalent to requiring that every step for α be followed, after a finite number of steps, by another step for α. Thus, in an execution that is weakly fair for α, no action starves.

By design, a **reject** step does not change the configuration. So if **reject**(α) is enabled in configuration $\mathbf{C}$, it is still enabled after any **reject** step. If a execution reaches a configuration $\mathbf{C}$ in which only **reject** steps are enabled, the configuration does not change from this point on in the execution. We call $\mathbf{C}$ a *fixpoint* of the program.

A less stringent condition that weak fairness is *minimal fairness*. Execution ε is minimally fair if every configuration $\mathbf{C}$ in ε, where $\mathbf{C}$ is not a fixpoint is followed eventually by an **accept** step. An **accept** step is enabled in $\mathbf{C}$, by definition, and an **accept**(α) step remains enabled at least until an **accept** step is taken. In the example above, if there is a total action α' in the program, other than α, then the minimally fair executions include an execution consisting of nothing but **accept**(α') steps. In this execution, x does not increase at all.

Minimal fairness guarantees *absence of global starvation* for the program. In terms of the analogy, the program is hungry for an **accept** step in any nonfixpont configuration. In a minimally fair execution, there are an infinite number of **accept** steps or the execution reaches a fixpoint. Note that an execution that is weakly fair in all actions is a minimally fair execution.

We extend weak fairness and minimal fairness to concurrent executions, by mapping the sequence of decision steps in the execution onto **accept** and **reject** steps in the obvious way. We show that the reduction given in the last chapter can reduce

a concurrent execution that is weakly fair in all actions to an atomic execution that is minimally fair, but not necessarily weakly fair in all actions.

This means that to capture all behaviours of executions under weak compatibility, we need to consider minimally fair sequential executions that are weakly fair in a subset of the actions. Such executions satisfy fewer properties than executions that weakly fair in all actions.

We define a control relation, called *strong compatibility*, that allows less concurrency than weak compatibility. We prove a second reduction theorem that shows how we can use control relations combining weak and strong compatibility to guarantee a reduction of a weakly fair concurrent execution to an atomic execution that is weakly fair for a given subset of the actions.

In the example, we need only weak fairness for α to guarantee that x is increasing. We do not need weak fairness on any other action. This observation suggests a strategy for implementing *TCB* programs to guarantee a given progress property. We identify the actions whose weak fairness is required to establish that an atomic execution has the desired property, and we then use a control relation that ensures executions that can be reduced to atomic executions that are weakly fair in the appropriate actions.

In the final section of this chapter, we consider the implementation of fairness conditions and control relations, using a *scheduler*.

6.2 Fairness conditions for *TCB*

To apply fairness to the execution of *TCB* programs, we must decide on a sequence of events that gives a linear order to the threads. Since the reduction maintains the order of decision steps, we use these steps to represent the order of the threads. For the fairness conditions we define, we must record rejecting threads as well as accepting threads, so we define the following.

Definition 6.1 *If $\varepsilon \in Z_m$, and $0 \le i < |\varepsilon|$, then*

$$DLabel(\varepsilon, i) \triangleq \begin{array}{ll} \mathsf{accept}(Root(\varepsilon, i)) & \text{if } \varepsilon\langle i \rangle \subset \mathsf{ACC} \\ \mathsf{reject}(Root(\varepsilon, i)) & \text{if } \varepsilon\langle i \rangle \subset \mathsf{REJ} \end{array}$$

For p an ascending sequence containing every i, such that $\varepsilon\langle i \rangle \subseteq \mathsf{DEC}$,

$$DSeq(\varepsilon) \triangleq \langle i \ : \ 0 \le i < |p| \ : \ DLabel(\varepsilon, p[i]) \ \rangle$$

$$DS \triangleq \{ \varepsilon \ : \ \varepsilon \in Complete(Z_m) \wedge |\varepsilon| = \infty \ : \ DSeq(\varepsilon) \}$$

We call $DSeq(\varepsilon)$ the decision sequence *for ε. We use Δ for a typical element of DS.*

The set DS contains infinite sequences over Lab_a. Fairness are only relevant for infinite executions, so we confine our attention to infinite decision sequences. Note that, if an infinite execution is complete, it has an infinite decision sequence. If $\varepsilon \in Z_a$, then $DSeq(\varepsilon) = SSeq(\varepsilon)$. The decision sequence then, is an idealized representation of an infinite execution as an atomic execution. The following function returns the action for an label in a decision sequence.

Definition 6.2 *If $\mathbf{L} \in Lab_a$*

$$DAct(\mathbf{L}) \triangleq \alpha \quad \text{if } \mathbf{L} = \mathsf{accept}(\alpha) \vee \mathbf{L} = \mathsf{reject}(\alpha)$$

6.2.1 Weak fairness

An infinite execution is *weakly fair* for $\alpha \in \mathbf{A}$ if there is an infinite number of threads for α. We formulate this in terms of decision sequences.

Definition 6.3 (Weak fairness) *If $\Delta \in DS$, $\alpha \in \mathbf{A}$, and $\mathbf{A}' \subseteq \mathbf{A}$, then*

$$\mathbf{WF}(\Delta, \alpha) \triangleq \langle \# \, i \ : \ 0 \le i \ : \ DAct(\Delta[i]) = \alpha \ \rangle = \infty$$

$$\mathbf{WF}(\Delta, \mathbf{A}') \triangleq \langle \forall \, \alpha \ : \ \alpha \in \mathbf{A}' \ : \ \mathbf{WF}(\Delta, \alpha) \ \rangle$$

$$\mathbf{WF}(\Delta) \triangleq \mathbf{WF}(\Delta, \mathbf{A})$$

If $\mathbf{WF}(\Delta, x)$*, we say that* Δ *is* weakly fair *for* x*. If* $\mathbf{WF}(\Delta)$*, we say that* Δ *is* weakly fair.

If Δ is weakly fair for α then Δ contains an infinite number of labels for α. Note that only a finite number of these need be $\textsf{accept}(\alpha)$ labels, the rest can be $\textsf{reject}(\alpha)$ labels.

6.2.2 Minimal fairness

An infinite execution is minimally fair if there is an infinite number of accepting threads in it, or, it reaches a configuration where no thread can accept. This final condition is difficult to state in terms of decision sequences, so we use the following, slightly stronger formulation: an infinite execution is minimally fair if there is an infinite number of accepting threads in it, or, if after all the accepting threads, there is at least one rejecting thread for every action. We use the following function to help with the definition.

Definition 6.4 *If* $\Delta \in DS$*, then*

$$LastA(\Delta) \triangleq \langle \min k \; : \; 0 < k \; : \; \langle \forall\, i \; : \; k < i \; : \; \Delta[i] \subseteq \textsf{reject} \; \rangle \; \rangle$$

Note that if there is an infinite number of **accept** steps in Δ, then the range for the minimum is empty, and so $LastA(\Delta) = \infty$

Definition 6.5 (Minimal fairness) *If* $\Delta \in DS$*, then*

$$\begin{aligned}\mathbf{MF}(\Delta) \;\triangleq\;& LastA(\Delta) = \infty \;\vee \\ & \langle \forall\; \alpha \\ & \;\; : \; \alpha \in \mathbf{A} \\ & \;\; : \; \langle \exists\, i \; : \; LastA(\Delta) < i \; : \; \Delta[i] = \textsf{reject}(\alpha) \; \rangle \\ & \;\; \rangle\end{aligned}$$

If $\mathbf{MF}(\Delta)$*, we say that* Δ *is* minimally fair.

Theorem 6.6

$$\langle \forall\, \Delta \;:\; \Delta \in DS \;:\; \mathbf{WF}(\Delta) \;\Rightarrow\; \mathbf{MF}(\Delta) \;\rangle$$

6.3 Program properties

A *property* is a subset the infinite atomic executions of a program. We define properties to include only executions satisfying a condition on the sequence of persistent states. We use conditions such as "the value of x does not decrease", or "x increases without bound", for an integer box variable x. We identify a subset of executions with the condition defining it, using the name "property" for either.

Program properties can be classified as *safety* properties and *progress* (or *liveness*) properties [21]. A safety property says that "nothing bad happens". The first example above is a safety property, asserting that a decrease in the value of x does not occur. Safety properties can always be satisfied by a program that does nothing. A progress property says that "something good happens". The second example above is a progress property, asserting that there are an infinite number of steps increasing the value of x. A progress property requires some action be taken by the executing program.

Our aim is to investigate the properties satisfied by all executions, and properties satisfied by all executions satisfying one of the fairness conditions from the last section. We define a couple of types of property, one for safety, and one for progress. We show that fair executions satisfy more progress properties than all executions, but that both fair and unfair executions satisfy the same safety properties.

We extend the fairness operators to executions in the obvious way, writing $\mathbf{WF}(\varepsilon)$ instead of $\mathbf{WF}(DSeq(\varepsilon))$, for example. Using this, we define the following sets of executions.

Definition 6.7

$$\begin{aligned} Z_i &\triangleq \{\, \varepsilon \mid \varepsilon \in Z_a \;\wedge\; |\varepsilon| = \infty \,\} \\ W_i &\triangleq \{\, \varepsilon \mid \varepsilon \in Z_i \;\wedge\; \mathbf{WF}(\varepsilon) \,\} \\ M_i &\triangleq \{\, \varepsilon \mid \varepsilon \in Z_i \;\wedge\; \mathbf{MF}(\varepsilon) \,\} \end{aligned}$$

From Theorem 6.6, we have $W_i \subseteq M_i$.

Let $\mathbf{P}$ be a property, that is, a subset of Z_i. If $Z_i = \mathbf{P}$, then every execution of the program satisfies $\mathbf{P}$. If $W_i \subseteq \mathbf{P}$, then every weakly fair execution of the program satisfies $\mathbf{P}$. If $M_i \subseteq \mathbf{P}$, then every minimally fair execution of the program satisfies $\mathbf{P}$.

We define some simple properties.

Definition 6.8 *For $q \subseteq PC_q$,*

$$\begin{aligned} \mathbf{stab}(q) &\triangleq \{\, \varepsilon \mid \varepsilon \in Z_i \;\wedge\; \langle \forall\, i,j \;:\; 0 \le i < j \;:\; \varepsilon[i] \in q \;\Rightarrow\; \varepsilon[j] \in q \;\rangle \,\} \\ \mathbf{unav}(q) &\triangleq \{\, \varepsilon \mid \varepsilon \in \mathbf{stab}(q) \;\wedge\; \langle \exists\, k \;:\; 0 \le k \;:\; \varepsilon[i] \in q \;\rangle \,\} \end{aligned}$$

We call $\mathbf{stab}(q)$ *a* stable *property, and* $\mathbf{unav}(q)$ *an* unavoidable *property.*

We identify a boolean expression over the box variables of a program with the subset of PC_q for which it is the membership predicate. So, we write $D.x > 5$ to mean the set $\{\, \mathbf{C} \mid \mathbf{C} \in PC_q \wedge [\![D.x > 5]\!]\mathbf{C} \,\}$. If q is such an expression, we say "q holds in $\mathbf{C}$" to mean $\mathbf{C} \in q$.

For $\varepsilon \in \mathbf{stab}(q)$, if q holds at any configuration in ε, it holds in every configuration after that one. Note that $\varepsilon \in \mathbf{stab}(q)$ if q does not hold at any configuration in ε. Properties $\mathbf{stab}(q)$ are safety properties. For $\varepsilon \in \mathbf{unav}(q)$, q does not hold in a finite prefix of the configurations of ε, and it holds in the remainder. Properties $\mathbf{unav}(q)$ are progress properties.

The following theorem shows an important difference between stable properties and unavoidable properties: stable properties are oblivious to fairness.

Theorem 6.9

$$\langle \forall\ q \\ :\ q \subseteq PC_q \\ :\ (W_i \subseteq \mathbf{stab}(q)\ \equiv\ Z_i \subseteq \mathbf{stab}(q))\ \wedge\ (M_i \subseteq \mathbf{stab}(q)\ \equiv\ Z_i \subseteq \mathbf{stab}(q)) \\ \rangle$$

Proof

We prove the first conjunct of the term. The proof for the second is similar. Assume $q \subseteq PC_q$. We show

$$W_i \subseteq \mathbf{stab}(q) \quad \equiv \quad Z_i \subseteq \mathbf{stab}(q)$$

We prove the equivalence as two implications.

Case $\Leftarrow$:

An immediate consequence of $W_i \subseteq Z_i$.

Case $\Rightarrow$:

We prove the contrapositive form.

$$Z_i \not\subseteq \mathbf{stab}(q) \quad \Rightarrow \quad W_i \not\subseteq \mathbf{stab}(q)$$

Assume $Z_i \not\subseteq \mathbf{stab}(q)$. Choose $\varepsilon \in Z_i$ such that $\varepsilon \notin \mathbf{stab}(q)$. From Definition 6.8, we can find i and j, where $0 \leq i < j$, $\varepsilon[i] \in q$, and $\varepsilon[j] \notin q$. Let ε' be an infinite execution, where $\mathit{Start}(\varepsilon') = \varepsilon[j]$, and $\mathit{SSeq}(\varepsilon')$ is an infinite sequence of atomic steps for the actions of the program in cyclic order. This execution is well-formed, since any action can be started in a quiescent configuration, and it contains an infinite

```
box D
   var b : boolean
         c : boolean
   action r  ::  b := false
   action s  ::  ¬b ⟶ c := false
end
```

Figure 6.1: Program with fairness-dependent unavoidability properties

number of decision steps for each action, so $\varepsilon' \in W_i$. Let $\varepsilon'' = \varepsilon\langle 0 \ldots (j-1)\rangle ; \varepsilon'$. The composition is defined by construction, and $\varepsilon'' \in W_i$. Since $\varepsilon''[i] = \varepsilon[i]$, and $\varepsilon''[j] = \varepsilon[j]$, we have $\varepsilon'' \notin \mathbf{stab}(q)$. Thus $W_i \not\subseteq \mathbf{stab}(q)$.
(End of proof)

Theorem 6.9 brings out an important property of stability: if an execution violates stability, there is a finite prefix of the execution that shows the violation. The result here is a general one: restricting executions to those satisfying a fairness condition does not widen the class of valid safety properties.

We do not have a result such as Theorem 6.9 for unavoidable properties, and other progress properties. We show this using the program in Figure 6.1. The program contains a single box D, so we drop the box component of identifiers, and write b for $D.b$. We show the following

$$
\begin{array}{lll}
Z_i \not\subseteq \mathbf{unav}(\neg b) & M_i \not\subseteq \mathbf{unav}(\neg b) & W_i \subseteq \mathbf{unav}(\neg b) \\
Z_i \not\subseteq \mathbf{unav}(\neg c) & M_i \subseteq \mathbf{unav}(\neg c) & W_i \subseteq \mathbf{unav}(\neg c)
\end{array}
$$

Since the program text contains no statement that assigns *true* to any variable, we have the following.

$$
\begin{array}{l}
Z_i \subseteq \mathbf{stab}(\neg b) \\
Z_i \subseteq \mathbf{stab}(\neg c)
\end{array}
$$

To show the results for Z_i, let $\mathbf{C}$ be a configuration satisfying $b \wedge c$. Note that if a thread for action s is started from $\mathbf{C}$, the thread rejects. Let

$$\varepsilon \quad = \quad (\; \mathbf{C}^{\infty} \; , \; \textsf{\textbf{reject}}(s)^{\infty} \;)$$

Execution ε consists of an infinite sequence of rejecting threads for s. All configurations are $\mathbf{C}$. Thus $\varepsilon \notin \mathbf{unav}(\neg b)$, and $\varepsilon \notin \mathbf{unav}(\neg c)$. This gives us

$$\begin{aligned} Z_i &\not\subseteq \mathbf{unav}(\neg b) \\ Z_i &\not\subseteq \mathbf{unav}(\neg c) \end{aligned}$$

For the case of M_i, we note that, since action r is total, any thread for it accepts. Thus any $\varepsilon' \in M_i$ contains an infinite number of **accept** steps. The first of these is an $\textsf{\textbf{accept}}(r)$, since a thread for s rejects from $\mathbf{C}$. If $\varepsilon'\langle i\rangle$ is the first $\textsf{\textbf{accept}}(r)$ step, then $\neg c$ holds in $\varepsilon'[i+1]$. Thus $\varepsilon' \in \mathbf{unav}(\neg c)$. If $\mathbf{C}, \mathbf{C}' \in PC_q$, b holds in $\mathbf{C}$, and $(\mathbf{C}, \mathbf{C}') \in \textsf{\textbf{accept}}(r)$, then $b \wedge \neg c$ holds in $\mathbf{C}'$. Thus $(\mathbf{C}', \mathbf{C}') \in \textsf{\textbf{accept}}(r)$, so we can define

$$\varepsilon \quad = \quad (\; \mathbf{C} \circ (\mathbf{C}')^{\infty} \; , \; \textsf{\textbf{accept}}(r)^{\infty} \;)$$

Then $\varepsilon \in M_i$, and $\varepsilon \notin \mathbf{unav}(\neg b)$. Thus

$$\begin{aligned} M_i &\not\subseteq \mathbf{unav}(\neg b) \\ M_i &\subseteq \mathbf{unav}(\neg c) \end{aligned}$$

For the case of W_i, we note that $W_i \subseteq \mathbf{unav}(\neg c)$ follows from $W_i \subseteq M_i$. To show $W_i \subseteq \mathbf{unav}(\neg b)$, assume $\varepsilon \in W_i$. Since ε is weakly fair, it contains a step for r. If the first such step is $\varepsilon\langle i\rangle$, $\neg c$ holds in all $\varepsilon[j]$ for $i < j$. Again, since ε is weakly fair, it contains a step for s after step i. Let the first such step be $\varepsilon\langle k\rangle$.

box *Semk*
  var n : *integer*
    b : *boolean*
  action K :: $n > 0 \longrightarrow b := \mathit{false}$
  method P :: $n > 0 \wedge b \longrightarrow n := n - 1$
  method V :: $n := n + 1$
end

box D
  var x : *integer*
  action a :: true & $\mathit{Semk.P} \longrightarrow x := x + 1$; $\mathit{Semk.V}$
end

Figure 6.2: The semaphore with a kill action

Then $\neg b$ holds in $\varepsilon[k+1]$. Thus

$$W_i \subseteq \mathbf{unav}(\neg b)$$

$$W_i \subseteq \mathbf{unav}(\neg c)$$

6.4 Fairness for weak compatibility

We consider the issue of fairness in relation to programs run under the weak compatibility control relation. Consider the program in Figure 6.2. The program contains a box *Semk*, which is a variant of box *sem*. Box *Semk* has an action K, in addition to the P and V methods from *sem*. The box has an additional variable, a boolean b, which is initially *true*. Method P is amended so that a call to it accepts, and grants the semaphore to the source of the call, only if the semaphore is available (that is, $n > 0$), and b holds. A call to action K accepts only if the semaphore is available, and, if it is, b is set to *false*. Box D contains an integer variable x, and an action a. A call to a accepts, and increments x, if the semaphore is successfully granted. Action a releases the semaphore before terminating.

From the text of the program, we can see that it has the following stability property.

$$Z_i \subseteq \mathbf{stab}(\mathit{Semk}.n \leq 0 \ \vee \ \neg\mathit{Semk}.b)$$

To show this, we note that, if $\mathit{Semk}.n \leq 0$ holds at any point in an execution, then no thread accepts, and the configuration is the same for the remainder of the execution.

If $\mathit{Semk}.n > 0 \wedge \neg\mathit{Semk}.b$ holds in any configuration, then a thread for action $D.a$ rejects, and a thread for action $\mathit{Semk}.K$ accepts, leaving the configuration unchanged.

For the progress properties we have

$$W_i \subseteq \mathbf{unav}(\mathit{Semk}.n \leq 0 \ \vee \ \neg\mathit{Semk}.b)$$

$$M_i \not\subseteq \mathbf{unav}(\mathit{Semk}.n \leq 0 \ \vee \ \neg\mathit{Semk}.b)$$

To show the first of these, assume we have $\varepsilon \in W_i$. If $\mathit{Semk}.n \leq 0 \vee \neg\mathit{Semk}.b$ holds in $\mathit{Start}(\varepsilon)$, then we are done. Assume this is not the case, so $\mathit{Semk}.n > 0 \wedge \mathit{Semk}.b$ holds in $\mathit{Start}(\varepsilon)$. A thread for $D.a$ started from such a configuration leaves the configuration unchanged on these values. A thread for $\mathit{Semk}.K$ inverts the value of $\mathit{Semk}.b$. Since $\varepsilon \in W_i$, there is a thread for $\mathit{Semk}.K$ in ε, so there is a configuration where $\mathit{Semk}.n \leq 0 \vee \neg\mathit{Semk}.b$ holds, and so $W_i \subseteq \mathbf{unav}(\mathit{Semk}.n \leq 0 \vee \neg\mathit{Semk}.b)$.

To show the second, let ε be an execution where $\mathit{Semk}.n > 0 \wedge \mathit{Semk}.b$ holds in $\mathit{Start}(\varepsilon)$, and all steps in ε are for $D.a$. Each of these threads accepts (so $\varepsilon \in M_i$), and after each step the values of $\mathit{Semk}.n$ and $\mathit{Semk}.b$ are unchanged. Therefore, $\varepsilon \notin \mathbf{unav}(\mathit{Semk}.n \leq 0 \vee \neg\mathit{Semk}.b)$, and so $M_i \not\subseteq \mathbf{unav}(\mathit{Semk}.n \leq 0 \vee \neg\mathit{Semk}.b)$.

Consider now the concurrent executions of the program. We first check the

actions for weak compatibility. We have

$$Semk.K \quad \sim \quad D.a$$

Using Definition 5.54, we get just the following condition to check.

$$\textsf{\textbf{accept}}(Semk.K) \quad \curvearrowleft \quad \textsf{\textbf{tm}}(D, Semk.V)$$

We can see that this holds by checking the code.

Suppose we start from a configuration satisfying $Semk.n > 0 \wedge Semk.b$. We can construct the following execution of a thread for $Semk.K$ executing concurrently with a thread for $D.a$.

$$\varepsilon_0 \ ; \ \varepsilon' \ ; \ \varepsilon_1$$

The execution of the thread for $D.a$ is split between exections ε_0 and ε_1. This thread accepts. The first contains all the steps up to and including the assignment to $D.n$, and the second contains the rest. Execution ε' is a compact execution for $Semk.K$. This must be a rejecting thread, since $Semk.b$ is *false* after ε_0. The final configuration is quiescent, and $Semk.n > 0 \wedge Semk.b$ holds

We can repeat the above execution infinitely many times to generate a concurrent execution ε containing an infinite number of threads for $D.a$, and an infinite number for $Semk.K$. Thus $\mathbf{WF}(\varepsilon)$. Note that ε has a quiescent configuration between each interleaved execution of two threads, and $Semk.n > 0 \wedge Semk.b$ holds in each of these configurations. As we have seen, there is no weakly fair atomic execution of the program in which $Semk.n > 0 \wedge Semk.b$ holds in every configuration. Thus the atomic reduction of ε cannot be in W_i.

If we follow through the reduction process in Chapter 5 for execution ε, we see that each of the rejecting threads for $Semk.K$ is removed during the reject removal

step. So the atomic reduction of ε is an execution containing only accepting steps to $D.a$. Therefore, the reduction is not weakly fair for $Semk.K$.

Note that, in the reduction in Chapter 5, every accepting thread from the original execution appears in the atomic reduction. From this, we conclude that if ε is a weakly fair execution that respects weak compatibility, then the atomic reduction of ε may not be weakly fair in any action that has only a finite number of accepting threads in ε.

The reason for this loss of fairness is our choice to discard **reject** steps during reduction. The justification given for discarding steps is that they are computationally insignificant. But, as we have seen, the fact that there was a rejected thread for an action is significant from the standpoint of fairness.

We next show a stronger control relation such that we can guarantee, for some set $\mathbf{A}' \in \mathbf{A}$ every thread in a concurrent execution for an action in $\mathbf{A}'$ appears in the atomic reduction. Using this control relation, if the concurrent execution is weakly fair in α, and $\alpha \in \mathbf{A}'$, then the atomic reduction is weakly fair in α.

6.5 Strong compatibility

We define a control relation, called *strong compatibility*, that allows us to maintain weak fairness for a given action during reduction. That is, we define a control relation $\approx$ such that if in any concurrent execution ε, if α is only executed concurrently with actions α' such that $\alpha \approx \alpha'$, then we can reduce the execution to an atomic execution, where every decision step for threads for α, both accepting and rejecting, appears in the atomic reduction.

To see how to do this, we look at two aspects of the proof of the first reduction theorem. Consider the step that removes **reject** steps. This is necessary because the weak compatibility rule does not allow a **tm** step to move left over a **REJ** step. In the proof of Lemma 5.59, we used the weak compatibility rule to move a **tm** step

left over a sequence of steps from other threads. To apply this rule, we must show that the sequence of steps contains no **REJ** steps. Thus, removing the **reject** steps is required for the left-mover reduction to succeed.

We can certainly argue that our choice to remove *all* **reject** steps is overzealous. We could certainly organize reduction so that we remove a **reject** step only if we reach a position where a left-moving **tm** step is to its right. This ensures that we remove the only those **reject** steps that absolutely must be removed for the reduction to go through.

In the proof of the first reduction theorem, we also ignore any part of the original execution after the final accepting thread. The reduced sequence without this suffix meets the requirements to be a reduction of the original sequence, so this does not cause problems with the proof. It would not be hard to extend the reduction from Chapter 5 to reduce a suffix with no **ACC** steps (that is, an execution containing only steps for rejecting threads) to an atomic execution containing the same sequence of rejecting decision steps as the original. As we have seen, rejecting threads are two-phase, so the reduction is straightforward.

But even with these improvements, we still cannot preserve weak fairness. Take the example from Figure 6.2. There cannot be a weakly fair execution that is a reduction of the particular concurrent execution that we show, because in the concurrent execution every thread for $Semk.K$ rejects, but a step for this action in an atomic execution with the same starting configuration always accepts. So the threads for $Semk.K$ cannot appear in an atomic execution.

We defined weak compatibility to have the minimum conditions that allowed the reduction theorem to go through. One choice we made was to discard all rejected threads. To reduce an execution, and not remove rejecting threads, we need a control relation that allows **tm** steps to move left over a **reject** or **pm-reject** step.

The format of the definition of $\approx$ is similar to that for $\sim$. We first define a

commutativity condition on procedures.

Definition 6.10 *For $\mu \in \mathbf{M}_t$, and $\pi \in \mathbf{P}$,*

$$
\begin{array}{rcl}
\pi \ \mathit{slc}\ \mu & \triangleq & \langle \forall\ D, E \\
& & \quad : D, E \in \mathbf{B} \ \wedge\ D \neq E \\
& & \quad : (\ \pi \in \mathbf{A} \ \Rightarrow \ \textsf{\textbf{accept}}(\pi) \curvearrowleft \textsf{\textbf{tm}}(D, \mu) \ \wedge \\
& & \qquad\qquad\qquad\qquad \textsf{\textbf{reject}}(\pi) \curvearrowleft \textsf{\textbf{tm}}(D, \mu) \ ) \ \wedge \\
& & \qquad (\ \pi \in \mathbf{M}_p \ \Rightarrow \ \textsf{\textbf{pm-accept}}(E, \pi) \curvearrowleft \textsf{\textbf{tm}}(D, \mu) \ \wedge \\
& & \qquad\qquad\qquad\qquad \textsf{\textbf{pm-reject}}(E, \pi) \curvearrowleft \textsf{\textbf{tm}}(D, \mu) \ ) \ \wedge \\
& & \qquad (\ \pi \in \mathbf{M}_t \ \Rightarrow \ \textsf{\textbf{tm}}(E, \pi) \curvearrowleft \textsf{\textbf{tm}}(D, \mu) \ ) \\
& & \quad \rangle
\end{array}
$$

The definition of strong left commutativity differs from the definition of weak left commutativity in that it requires commutativity with both accepting and rejecting **atomic** steps, not just with accepting steps. If $\pi\ \mathit{slc}\ \mu$, then a **tm** step for μ can move left over any atomic step for π, accepting or rejecting. We have a couple of properties of *slc* that are immediate from this definition.

Theorem 6.11

$$
\begin{array}{l}
\langle \forall\ \mu \\
\quad : \mu \in \mathbf{M}_t \\
\quad : \langle \forall\, \pi \ : \ \pi \in \mathbf{P}_p \ : \ \pi\ \mathit{slc}\ \mu \Rightarrow \pi\ \mathit{wlc}\ \mu \ \rangle \ \wedge \\
\qquad \langle \forall\, \pi \ : \ \pi \in \mathbf{P}_t \ : \ \pi\ \mathit{slc}\ \mu \equiv \pi\ \mathit{wlc}\ \mu \ \rangle \\
\quad \rangle
\end{array}
$$

Below we repeat the examples used for weak left commutativity. The first examples are for the box *Sem* from Figure 2.3. The first line shows that strong left

commutativity is truly a stronger relation than weak left commutativity.

$$\neg(\ Sem.P \quad slc \quad Sem.V\ )$$

$$Sem.V \quad slc \quad Sem.V$$

For the first of these, the condition that fails is, as may be expected,

$$\textsf{\textbf{pm-reject}}(Sem.P) \quad \curvearrowleft \quad \textsf{\textbf{tm}}(Sem.V)$$

This fails because a call to $Sem.P$ immediately after a call to $Sem.V$ alway accepts, so

$$TwoStep(\ \textsf{\textbf{tm}}(Sem.V)\ ,\ \textsf{\textbf{pm-reject}}(Sem.P)\ ) \quad = \quad \emptyset$$

Also, a $\textsf{\textbf{pm-reject}}(Sem.P)$ is enabled in a configuration in which $Sem.n = 0$, and a $\textsf{\textbf{tm}}(Sem.V)$ is enabled in any configuration, so we have so

$$TwoStep(\ \textsf{\textbf{pm-reject}}(Sem.P)\ ,\ \textsf{\textbf{tm}}(Sem.V)\ ) \quad \neq \quad \emptyset$$

So for any $\varepsilon \in TwoStep(\ \textsf{\textbf{pm-reject}}(Sem.P)\ ,\ \textsf{\textbf{tm}}(Sem.V)\ )$, we have $Swap(\varepsilon) = \bot$. The commutativity of $Sem.V$ with itself follows from the corresponding result with wlc, and Theorem 6.11.

For the box $Semk$ in Figure 6.2, we have

$$\neg(\ Semk.P \quad slc \quad Semk.V\ )$$

$$\neg(\ Semk.K \quad slc \quad Semk.V\ )$$

The arguments for these are similar to the above argument for box Sem.

For the *Buff* example from Figure 1.2, we have

$$\neg(\ Buff.get \quad slc \quad Buff.put\ )$$
$$\neg(\ Buff.put \quad slc \quad Buff.put\ )$$

The first of these is shown using a similar argument to that for the first case for *Sem*, above. The second follows from the negative result for *wlc* and Theorem 6.11.

We now define the strong compatibility relation.

Definition 6.12 (Strong compatibility) *For* $\alpha, \alpha' \in \mathbf{A}$,

$$\begin{aligned} \alpha \ \approx \ \alpha' \quad \triangleq \quad & \langle \forall\ \pi, \pi' \\ & \quad : \pi \in Range(\alpha) \ \wedge \ \pi' \in Range(\alpha') \\ & \quad : (\pi \in \mathbf{M}_t \ \Rightarrow \ \pi' \ slc \ \pi) \ \wedge \ (\pi' \in \mathbf{M}_t \ \Rightarrow \ \pi \ slc \ \pi') \\ & \quad \rangle \end{aligned}$$

If $\alpha \approx \alpha'$, *we say that* α *is* strongly compatible *with* α'.

For the actions in Figure 2.3, we have

$$\neg(\ D.act \quad \approx \quad E.acq\ )$$
$$\neg(\ D.act \quad \approx \quad E.rel\ )$$

For the actions in Figure 6.2, we have

$$\neg(\ Semk.K \quad \approx \quad D.a\ )$$

For the actions in Figure 1.3, we have

$$\neg(\ Prod.make \quad \approx \quad Cons.use\ )$$

From these we see that strong compatibility allows less concurrency that weak com-

patibility. In particular, for the *Semk* example, running the program under strong compatibility excludes the execution in which all threads for $Semk.K$ reject. There is no magic involved here. We have not found a way to run actions $Semk.K$ and $D.a$ concurrently while preserving fairness in reduction. We are simply noticing where the problems arise, and defining sufficient conditions to exclude the problematic cases.

The reduction rule for $\approx$ allows a **tm** step to move left over **REJ** steps in addition to the steps allowed by the weak compatibility rule.

Theorem 6.13 (Strong compatibility rule)

$$
\begin{array}{l}
\langle \forall\ \mathbf{L}, \mathbf{L}', \varepsilon \\
\quad :\ \mathbf{L}, \mathbf{L}' \in Lab_m\ \wedge\ \mathbf{L} \not\subseteq \textbf{end}\ \wedge\ \mathbf{L}' \subseteq \textbf{tm}\ \wedge \\
\qquad \varepsilon \in TwoStep(\mathbf{L}, \mathbf{L}')\ \wedge\ \varepsilon\ resp\ \approx \\
\quad :\ \varepsilon \overset{*}{\rightarrow} Swap(\varepsilon) \\
\quad \rangle
\end{array}
$$

6.6 The second reduction theorem

We now have the machinery to show a reduction that preserves weak fairness. First we define a rather more general control relation, one that is part way between weak and strong compatibility.

It is often the case that a program has a progress property because of the presence of a single action. An example is the property $\mathbf{unav}(\neg Semk.b)$ in the program in Figure 6.2. To show that $\varepsilon \in \mathbf{unav}(\neg Semk.b)$, it is sufficient to have $\mathbf{WF}(\varepsilon, Semk.K)$. That is, we do not use $\mathbf{WF}(\varepsilon, D.a)$ in the proof. The argument given is valid, even if there are no threads for $D.a$ in ε.

For the example in Figure 6.1, to show that $\varepsilon \in \mathbf{unav}(\neg b)$, we show that there is an accepting thread in ε for action $D.r$, and that there is an accepting

thread for action $D.s$ occurring later in ε. For the first of these, $\mathbf{MF}(\varepsilon)$ is enough. So again, for this example, we do not require $\mathbf{WF}(\varepsilon, D.r)$. We can show that if $\varepsilon \in Z_i$, and $\mathbf{MF}(\varepsilon)$ and $\mathbf{WF}(\varepsilon, D.s)$, then $\varepsilon \in \mathbf{unav}(\neg b)$.

As we saw above, strong compatibility is a stringent requirement on actions, so an execution running under this control relation has restricted concurrency. The above observations suggest that we tailor the control relation to the properties that we require of the program. That is, we find a subset of actions $\mathbf{A}'$ such that for any $\varepsilon \in Z_i$ where $\mathbf{MF}(\varepsilon)$, and $\mathbf{WF}(\varepsilon, \mathbf{A}')$, we can show that the ε has the desired properties. The aim is to make the set $\mathbf{A}'$ as small as possible. Having done this, we can relax the control relation on the actions in $\mathbf{A} \setminus \mathbf{A}'$. We define a control relation that ensures that every concurrent execution ε such that $\mathbf{WF}(\varepsilon)$, there is an atomic execution ε' containing the same accept sequence as ε, and for every $\alpha \in \mathbf{A}'$, ε' contains every decision step for α that is in ε. Then we have $\mathbf{MF}(\varepsilon')$, and $\mathbf{WF}(\varepsilon', \mathbf{A}')$, as required.

We show the reduction for this more general case. The reduction for executions respecting $\approx$ as the control relation is a special case, when $\mathbf{A}' = \mathbf{A}$. We first define a control relation relative to a set of actions.

Definition 6.14 *For* $\mathbf{A}' \subseteq \mathbf{A}$,

$$\mathbf{U}(\mathbf{A}') \;\; = \;\; \approx \; \cup \; \{ \; (\alpha, \alpha') \mid \alpha \; \sim \; \alpha' \; \wedge \; \{\alpha, \alpha'\} \, \mathrm{disj} \, (\mathbf{A}' \cap \mathbf{A}_p) \; \}$$

The control relation $\mathbf{U}(\mathbf{A}')$ is $\approx$ extended with pairs of weakly compatible actions (α, α') where neither action is a partial action in $\mathbf{A}'$. The following theorem gives some properties of $\mathbf{U}(\mathbf{A}')$ that follow from this definition.

Theorem 6.15

$$\langle \forall\ \mathbf{A}', \mathbf{A}''$$
$$:\ \mathbf{A}'' \subseteq \mathbf{A}$$
$$:\ \mathbf{A}' \subseteq \mathbf{A}'' \ \Rightarrow\ \mathbf{U}(\mathbf{A}') \supseteq \mathbf{U}(\mathbf{A}'')$$
$$\rangle$$
$$\mathbf{U}(\mathbf{A}) \ = \ \approx$$
$$\mathbf{U}(\mathbf{A}_t) \ = \ \sim$$

The first part of the theorem shows that $\mathbf{U}(\mathbf{A}')$ is antimonotonic in it argument. The other two parts show that the extreme cases of this relation.

Below we give the second reduction theorem. This theorem states that a concurrent execution that is weakly fair and respects $\mathbf{U}(\mathbf{A}')$ can be reduced to an atomic execution that is minimally fair, and weakly fair in $\mathbf{A}'$.

Theorem 6.16 (Second reduction theorem)

$$\langle \forall\ \varepsilon, \mathbf{A}'$$
$$:\ \varepsilon \in \mathit{Complete}(Z)\ \wedge\ \mathbf{WF}(\varepsilon)\ \wedge\ \mathbf{A}' \subseteq \mathbf{A}\ \wedge\ \varepsilon\ \mathit{resp}\ \mathbf{U}(\mathbf{A}')$$
$$:\ \langle \exists\ \varepsilon'\ :\ \varepsilon' \in Z_a\ :\ \varepsilon \leadsto \varepsilon'\ \wedge\ \mathbf{MF}(\varepsilon')\ \wedge\ \mathbf{WF}(\varepsilon', \mathbf{A}')\ \rangle$$
$$\rangle$$

Proof

The proof proceeds much as in the proof of the first reduction theorem. We do not repeat the details here, but we note the relevant changes. We use the following lemma.

Lemma 6.17

$$
\begin{array}{l}
\langle\forall\ \mathbf{C}, \mathbf{A}' \\
\quad : \mathbf{C} \in PC \ \wedge\ \mathbf{A}' \subseteq \mathbf{A} \ \wedge\ \mathit{Actives}(\mathbf{C}) \cap \mathbf{A}' \cap \mathbf{A}_p \ \wedge\ \mathbf{C}\ \mathit{resp}\ \mathbf{U}(\mathbf{A}') \\
\quad : \mathbf{C}\ \mathit{resp}\ \approx \\
\quad \rangle
\end{array}
$$

The lemma says that in any configuration where a partial action in $\mathbf{A}'$ is active, if the configuration respects $\mathbf{U}(\mathbf{A}')$, then it also respects $\approx$.

We state and prove a variant of Lemma 5.58 that says we can reduce the beginning of ε to an execution starting with an **accept** or **reject** step, rather than just the former. The proof of this proceeds much as with Lemma 5.58. After the rendezvous reduction and the right-mover reduction, we do not use the reject removal step. Instead we proceed with the left-mover reduction, this time reducing accepting and rejecting threads at the same time. Whenever there is a **tm** step to the right of a **REJ** step, and the **tm** step must move left, there are two cases. If the **REJ** step is from a thread for an action in $\mathbf{A}'$, then, using Lemma 6.17, we have that the strong compatibility rule is applicable, so we apply it to move the **tm** step over the **REJ** step; otherwise, the **REJ** step is from a thread for an action not in $\mathbf{A}'$, so we move any noncontiguous steps for the rejecting thread left, using the left-mover rule, and we remove these steps, using the atomic rule, and the reject removal rule.

This gives us the necessary reduction. To show that the reduced exection has the required fairness properties, we first note that the only time a **REJ** step is removed from the execution during the reduction is when a **tm** step must move left. In this case, the thread for the **REJ** is running concurrently with a committed thread. Also, the only **REJ** steps removed are **reject**(α) steps for $\alpha \notin \mathbf{A}'$. Let ε' be the atomic reduction of ε produced by the procedure described above. We have $\mathit{NumA}(\varepsilon) = \mathit{NumA}(\varepsilon')$, so if $\mathit{NumA}(\varepsilon) = \infty$, then $\mathbf{MF}(\varepsilon')$. Otherwise, if

$NumA(\varepsilon) < \infty$, then none of the rejecting threads in ε with decision steps appearing after the last **action-end** step is removed by the above process. Since $\mathbf{WF}(\varepsilon)$, there is at least one thread for each action among these, so $\mathbf{MF}(\varepsilon')$. If $\alpha \in \mathbf{A}'$, then no **DEC** step for α is removed during the reduction. Since $\mathbf{WF}(\varepsilon)$, there is an infinite number of them in ε, and thus an infinite number in ε', giving $\mathbf{WF}(\varepsilon', \alpha)$.
(End of proof)

6.7 Scheduling *TCB* programs

The *scheduler* is the major component of an implementation of a *TCB* program. The scheduler is a control program that decides when to start a thread. The scheduler has two tasks: to ensure that the execution of the program respects the given control relation, and to ensure that a threads for different actions are started often enough to satisfy the given fairness condition.

The control relation, as we have seen, is chosen to ensure that all treads are terminating, and that all concurrent executions are reducible to atomic executions. The fairness condition, together with the control relation, determines the fairness condition satisfied by the reduced executions.

For concurrent execution, we restrict our attention to weak fairness. Minimally fair sequential executions have very weak progress properties. We have seen that, with a weakly fair concurrent execution, and an appropriate control relation, we can guarantee a reduction to an atomic execution that is weakly fair for any subset of the actions. For a minimally fair concurrent execution, we can show only redution to a minimally fair atomic execution.

The requirements of the control relation and the fairness condition must be handled in conjunction. To ensure that a thread for an action α is started infinitely often, we must ensure that infinitely often there is no thread running for an action α' such that (α, α') is not in the control relation.

We assume that the scheduler only interacts with the program to start threads and end threads. To start a thread for action $D.a$, the scheduler puts an entry for a in D's call queue, according to rule **(action-start)**. There are no **action-start** steps in the execution, other than those taken by the scheduler. To ensure that the control relation is respected, the scheduler keeps a record of the actions with active threads at each point in the execution. Once a thread is started, the scheduler has no further influence over its execution. The boxes execute independently according to the semantics. When the thread completes, the scheduler updates its record of active threads. The completion of a thread for action α may make it possible to start a thread for an action α', where (α, α') is not in the control relation.

We can abstract the problem of scheduling as follows. A scheduler is a program that interacts with an executing *TCB* program. The scheduler's task is to ensure that every execution respects a given control relation and fairness condition. The scheduler and the program interact using messages $S(\alpha)$, $A(\alpha)$, and $R(\alpha)$, for $\alpha \in \mathbf{A}$. The scheduler sends $S(\alpha)$ messages to the executing program to start a thread for action α, and the executing program sends $A(\alpha)$ and $R(\alpha)$ messages to the scheduler when a thread completes. The first is sent for an accepting thread and the second for a rejecting thread. The program executes according to the queue semantics, with the following additions. An **action-start** step is taken for action α whenever an $S(\alpha)$ message is received from the scheduler, and when an **action-end** or **action-reject** step is taken for this thread, an $A(\alpha)$ or $R(\alpha)$ message is sent to the scheduler.

For the scheduler's purposes, a thread is active from the time the $S(\alpha)$ message is sent, to the time the corresponding $A(\alpha)$ or $R(\alpha)$ step is received from the executing program. The first of these is sent at a time decided by the scheduler. When the others are sent is outside of the scheduler's control. We assume only that

for every $S(\alpha)$ message sent, an $A(\alpha)$ or $R(\alpha)$ is received at some later time, that is, that every thread terminates.

We defined fairness conditions in terms of the sequence of decision steps. The scheduler has imprecise information about the exact sequence of decision steps. We assume only that the decision step for a thread happens at some time between the time the scheduler sends the **start** message for the thread, and when it receives the corresponding reply. But note that we can recast the weak fairness condition in terms of the scheduler messages as follows.

$$\mathbf{WF}(\varepsilon, \alpha) \triangleq \text{ the number of } S(\alpha) \text{ messages sent is infinite}$$

This is clearly equivalent to the earlier definition of $\mathbf{WF}(\varepsilon, \alpha)$, assuming that every thread terminates, and thus every thread that is started takes a decision step before it ends.

This problem of scheduling an execution that is weakly fair, and respects a given control relation, is an instance of a well-known problem in distibuted computing, the problem of the dining philosophers [5]. This problem concerns a set of philosophers, each of which can be in one of three phases, *thinking*, *hungry*, or *eating*. We start with all philosophers in the *thinking* phase. A philosopher becomes *hungry* when it wishes to eat. There is a control program that decides when a hungry philosphers can start eating. We assume that every eating philosopher eventually stops eating, an enters phase *thinking*. To eat again, the philosopher enters phase *hungry* and waits for the control program to let it start eating.

We restrict the behaviour of the control program using an *incompatibility graph*. The graph contains a node for each philospher, and an edge between two nodes if the corresponding philosophers are not allowed to eat at the same time. The problem is then to write a control program such that ensures that every execution respects the incompatibility graph, and every hungry philosopher eventually eats.

The actions in a *TCB* program are the philosophers. For this case, the *eating* phase corresponds to the execution of a thread for an action. There is no need for a *thinking* phase, since when a thread for an action α completes, the system is ready to start another thread for α as soon as it is able. That is, a philosopher that is not eating is hungry.

The incompatibility graph is the complement of the control relation. There is an edge from the node for α to the node for α' if (α, α') is not in the control relation.

There are several solutions to this problem, some defined in terms a single control program, some defined in terms of a network of coöperating programs. In [25], Misra gives an algorithm for a Seuss scheduler that ensures weak fairness.

Chapter 7

Conclusions

7.1 Summary of the main results

The three main theorems of this work, the complete execution theorem, and the two reduction theorems, give a framework for using the Seuss model.

The complete execution theorem gives a set of necessary and sufficient conditions for an execution to be complete. The discussion following the theorem in Chapter 4 gives guidelines for implementing a *TCB* system so that all executions are complete.

The reduction theorems show that a concurrent execution respecting a given control relations can be reduced to an atomic execution, where the executions have the same sequence of accepting actions, and their configuration sequences are closely related. The first reduction theorem shows a reduction without regard to fairness, and the second theorem shows that we can trade concurrency for fairness conditions in the reduction.

7.2 Future work

Below we present a number of areas for future research stemming from the present work.

7.2.1 Reasoning about Seuss programs

We argued in the introduction that reasoning about atomic executions of Seuss programs is simpler than reasoning directly about concurrent executions. This is the motivation for the reduction theorems. Given the theorems, the proof of correctness of a Seuss program is separated into two parts. One is proving properties of atomic executions of a program, and the other is deriving properties of the concurrent executions from the atomic execution properties.

We also argued in the introduction that the structure of a Seuss program is designed to allow the application of several common techniques for building correct programs — hierarchical reasoning, compositional reasoning, and program refinement. These can all be applied to reasoning about atomic executions.

A logic for atomic executions is under development. The intention is to design a logic that allows the specification of a procedure to be derived from the procedure's code, and the specifications of any methods called in the code.

Before Seuss is a usable system for writing and proving programs, the relationship between the properties of a concurrent execution and the properties of its atomic reduction must be clearly established. The discussion in Section 5.5 shows some ideas for this.

The properties of an atomic execution can be expressed in terms of the values of program variables. Properties such as "$x \geq y$ is invariant" have a clear interpretation for atomic executions. This property is true of an atomic execution if $x \geq y$ holds in every configuration.

Consider now a concurrent execution of a program whose atomic execution

satisfies the above property. We cannot claim that $x \geq y$ holds in every configuration. From the discussion in Section 5.5, we know only that the predicate holds in every quiescent configuration. This tells us nothing about an execution that contains no quiescent configurations.

The example using the probe action suggests that properties of concurrent executions may be best expressed not only in terms of the values of program variables, but also in terms of the acceptance or rejection of a thread for a given action. If we add action α to the program, where α is a probe for $x \geq y$, and execute the program so that α does not execute concurrently with actions that change the value of $x \geq y$, then the property "every thread for action α accepts" is a consequence of the invariance of $x \geq y$ in the atomic executions.

7.2.2 Concurrent termination

The problem of identifying a control relation that ensures that all threads in a *TCB* program are terminating is nontrivial. One approach is to define "noninterference" conditions between actions, just as we use commutativity conditions to ensure that actions do not interfere with each other's execution for the purposes of reduction.

7.2.3 Negative alternatives

As noted in Section 1.4, we choose not to implement negative alternatives in *TCB*. Extending the syntax and semantics of the language to include negative alternatives is fairly straightforward. The reject removal rule is not valid if a rejecting thread can change the values of the program variables, so this suggests that we must use strong compatibility to ensure reduction for actions that may call negative alternatives.

Appendix A

Semantics for TCB languages

A.1 Rendezvous semantics for $TCBtot$

(t-action-start-rdv)

$$\frac{\begin{array}{r@{\;}l} \mathbf{C}.D &= (\textsc{idle}, \sigma, \bot) \\ a &\in \mathit{TotActs}(D) \\ \gamma' &= (a) \\ \theta' &= \mathit{Code}(D.a) \end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma, \gamma', \theta')]}$$

(total-call-rdv)

$$\frac{\begin{array}{r@{\;}l} \mathbf{C}.D &= (\textsc{accept}, \sigma_0, \gamma_0, \theta_0) \\ \theta_0 &= E.m(\tilde{e}; \tilde{x}); \hat{\theta} \\ \mathbf{C}.E &= (\textsc{idle}, \sigma_1, \bot) \\ \tilde{v} &= [\![\tilde{e}]\!]\sigma_0 \\ \tilde{y} &= \mathit{InParam}(E.m) \\ \tilde{z} &= \mathit{OutParam}(E.m) \\ \sigma'_1 &= \sigma\,[\tilde{z} \mapsto \tilde{\bot}]\,[\tilde{y} \mapsto \tilde{v}] \\ \gamma'_1 &= (m, D) \\ \theta'_1 &= \mathit{Code}(E.m) \end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\textsc{wait}, \sigma_0, \gamma_0, \theta_0) \\ E \mapsto (\textsc{accept}, \sigma'_1, \gamma'_1, \theta'_1)) \end{bmatrix}}$$

(local-step)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\textsc{accept}, \sigma, \gamma, \theta) \\ (\sigma, \theta) \longrightarrow (\sigma', \theta')\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma', \gamma, \theta')]}$$

(proc-term-rdv)

$$\frac{\begin{array}{r@{\;}l}\mathbf{C}.D &= (\textsc{accept}, \sigma, \gamma, \bot) \\ \gamma &= (p, Q) \\ \tilde{z} &= OutParam(D.p) \\ \sigma' &= \sigma \restriction BoxVars(D) \\ \tilde{v} &= [\![\tilde{z}]\!]\sigma \\ \gamma' &= (p, Q, \tilde{v})\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{return}, \sigma', \gamma')]}$$

(action-end-rdv)

$$\frac{\begin{array}{r@{\;}l}\mathbf{C}.D &= (\textsc{return}, \sigma, \gamma) \\ \gamma &= (a)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{idle}, \sigma, \bot)]}$$

(total-return-rdv)

$$\frac{\begin{array}{r@{\;}l}\mathbf{C}.D &= (\textsc{wait}, \sigma_0, \gamma_0, \theta_0) \\ \theta_0 &= E.m(\tilde{e}; \tilde{x}); \theta'_0 \\ \mathbf{C}.E &= (\textsc{return}, \sigma_1, \gamma_1) \\ \gamma_1 &= (m, D, \tilde{v}) \\ \sigma'_0 &= \sigma_0\,[\tilde{x} \mapsto \tilde{v}]\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\textsc{accept}, \sigma'_0, \gamma_0, \theta'_0) \\ E \mapsto (\textsc{idle}, \sigma_1, \bot)\end{bmatrix}}$$

A.2 Rendezvous semantics for *TCB*

(p-action-start-rdv)

$$\frac{\mathbf{C}.D = (\text{IDLE}, \sigma, \bot) \quad a \in \mathit{PartActs}(D) \quad \gamma' = (a)}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\text{GUARD}, \sigma, \gamma')]}$$

(t-action-start-rdv)

$$\frac{\mathbf{C}.D = (\text{IDLE}, \sigma, \bot) \quad a \in \mathit{TotActs}(D) \quad \gamma' = (a) \quad \theta' = \mathit{Code}(D.a)}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\text{ACCEPT}, \sigma, \gamma', \theta')]}$$

(guard-test-rdv)

$$\frac{\begin{array}{r}\mathbf{C}.D = (\text{GUARD}, \sigma_0, \gamma_0) \\ p = \mathit{Proc}(\gamma) \\ \mathit{Alt}(D.p, \sigma_0) = (E.m(\tilde{e}; \tilde{x}), \theta) \\ \theta'_0 = E.m(\tilde{e}; \tilde{x}); \hat{\theta} \\ \mathbf{C}.E = (\text{IDLE}, \sigma_1, \bot) \\ \tilde{v} = [\![\tilde{e}]\!]\sigma_0 \\ \tilde{y} = \mathit{InParam}(E.m) \\ \tilde{z} = \mathit{OutParam}(E.m) \\ \sigma'_1 = \sigma_1\,[\tilde{z} \mapsto \tilde{\bot}]\,[\tilde{y} \mapsto \tilde{v}] \\ \gamma'_1 = (m, D)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\text{PWAIT}, \sigma_0, \gamma_0, \theta'_0) \\ E \mapsto (\text{GUARD}, \sigma'_1, \gamma'_1) \end{bmatrix}}$$

(total-call-rdv)

$$\frac{\begin{array}{r}\mathbf{C}.D = (\text{ACCEPT}, \sigma_0, \gamma_0, \theta_0) \\ \theta_0 = E.m(\tilde{e}; \tilde{x}); \hat{\theta} \\ \mathbf{C}.E = (\text{IDLE}, \sigma_1, \bot) \\ \tilde{v} = [\![\tilde{e}]\!]\sigma_0 \\ \tilde{y} = \mathit{InParam}(E.m) \\ \tilde{z} = \mathit{OutParam}(E.m) \\ \sigma'_1 = \sigma\,[\tilde{z} \mapsto \tilde{\bot}]\,[\tilde{y} \mapsto \tilde{v}] \\ \gamma'_1 = (m, D) \\ \theta'_1 = \mathit{Code}(E.m)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\text{WAIT}, \sigma_0, \gamma_0, \theta_0) \\ E \mapsto (\text{ACCEPT}, \sigma'_1, \gamma'_1, \theta'_1)) \end{bmatrix}}$$

(guard-accept)

$$\frac{\mathbf{C}.D = (\textsc{guard}, \sigma, \gamma) \quad p = \mathit{Proc}(\gamma) \quad \mathit{Alt}(D.p, \sigma) = (\bot, \theta')}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma, \gamma, \theta')]}$$

(guard-reject)

$$\frac{\mathbf{C}.D = (\textsc{guard}, \sigma, \gamma) \quad p = \mathit{Proc}(\gamma) \quad \mathit{Alt}(D.p, \sigma) = \bot \quad \sigma' = \sigma \upharpoonright \mathit{BoxVars}(D)}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{reject}, \sigma', \gamma)]}$$

(local-step)

$$\frac{\mathbf{C}.D = (\textsc{accept}, \sigma, \gamma, \theta) \quad (\sigma, \theta) \longrightarrow (\sigma', \theta')}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma', \gamma, \theta')]}$$

(proc-term-rdv)

$$\frac{\mathbf{C}.D = (\textsc{accept}, \sigma, \gamma, \bot) \quad \gamma = (p, Q) \quad \tilde{z} = \mathit{OutParam}(D.p) \quad \sigma' = \sigma \upharpoonright \mathit{BoxVars}(D) \quad \tilde{v} = [\![\tilde{z}]\!]\sigma \quad \gamma' = (p, Q, \tilde{v})}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{return}, \sigma', \gamma')]}$$

(action-end-rdv)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\textsc{return}, \sigma, \gamma) \\ \gamma = (a)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\left[D \mapsto (\textsc{idle}, \sigma, \bot)\right]}$$

(action-reject-rdv)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\textsc{reject}, \sigma, \gamma) \\ \gamma = (a)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\left[D \mapsto (\textsc{idle}, \sigma, \bot)\right]}$$

(test-accept-rdv)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\textsc{pwait}, \sigma_0, \gamma_0, \theta_0) \\ \theta_0 = E.m(\tilde{e}; \tilde{x}); \theta'_0 \\ \mathbf{C}.E = (\textsc{return}, \sigma_1, \gamma_1) \\ \gamma_1 = (m, D, \tilde{v}) \\ \sigma'_0 = \sigma_0\left[\tilde{x} \mapsto \tilde{v}\right]\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix}D \mapsto (\textsc{accept}, \sigma'_0, \gamma_0, \theta'_0) \\ E \mapsto (\textsc{idle}, \sigma_1, \bot)\end{bmatrix}}$$

(test-reject-rdv)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\textsc{pwait}, \sigma_0, \gamma_0, \theta_0) \\ \theta_0 = E.m(\tilde{e}; \tilde{x}); \hat{\theta} \\ \mathbf{C}.E = (\textsc{reject}, \sigma_1, \gamma_1) \\ \gamma_1 = (m, D) \\ \sigma'_0 = \sigma_0 \upharpoonright \mathit{BoxVars}(D)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix}D \mapsto (\textsc{reject}, \sigma'_0, \gamma_0) \\ E \mapsto (\textsc{idle}, \sigma_1, \bot)\end{bmatrix}}$$

(total-return-rdv)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\textsc{wait}, \sigma_0, \gamma_0, \theta_0) \\ \theta_0 = E.m(\tilde{e}; \tilde{x}); \theta'_0 \\ \mathbf{C}.E = (\textsc{return}, \sigma_1, \gamma_1) \\ \gamma_1 = (m, D, \tilde{v}) \\ \sigma'_0 = \sigma_0\left[\tilde{x} \mapsto \tilde{v}\right]\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix}D \mapsto (\textsc{accept}, \sigma'_0, \gamma_0, \theta'_0) \\ E \mapsto (\textsc{idle}, \sigma_1, \bot)\end{bmatrix}}$$

A.3 Queue semantics for *TCB*

(action-start)

$$\frac{\begin{array}{r@{}l} \mathbf{C}.D &= (\phi, \sigma, \gamma, \theta) \\ a &\in \mathit{Actions}(D) \\ \gamma' &= \gamma \lhd (a) \end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\phi, \sigma, \gamma', \theta)]}$$

(guard-test)

$$\frac{\begin{array}{r@{}l} \mathbf{C}.D &= (\textsc{guard}, \sigma_0, \gamma_0) \\ p &= \mathit{Proc}(\gamma) \\ \mathit{Alt}(D.p, \sigma_0) &= (E.m(\tilde{e}; \tilde{x}), \theta) \\ \theta_0' &= E.m(\tilde{e}; \tilde{x}); \hat{\theta} \\ \mathbf{C}.E &= (\phi_1, \sigma_1, \gamma_1, \theta_1) \\ \tilde{v} &= [\![\tilde{e}]\!]\sigma_0 \\ \gamma_1' &= \gamma_1 \lhd (m, D, \tilde{v}) \end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\textsc{pwait}, \sigma_0, \gamma_0, \theta_0') \\ E \mapsto (\phi_1, \sigma_1, \gamma_1', \theta_1) \end{bmatrix}}$$

(total-call)

$$\frac{\begin{array}{r@{}l} \mathbf{C}.D &= (\textsc{accept}, \sigma_0, \gamma_0, \theta_0) \\ \theta_0 &= E.m(\tilde{e}; \tilde{x}); \hat{\theta} \\ \mathbf{C}.E &= (\phi_1, \sigma_1, \gamma_1, \theta_1) \\ \tilde{v} &= [\![\tilde{e}]\!]\sigma_0 \\ \gamma_1' &= \gamma_1 \lhd (m, D, \tilde{v}) \end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\textsc{wait}, \sigma_0, \gamma_0, \theta_0) \\ E \mapsto (\phi_1, \sigma_1, \gamma_1', \theta_1) \end{bmatrix}}$$

(p-action-init)

$$\frac{\begin{array}{r}\mathbf{C}.D = (\textsc{idle}, \sigma, \gamma) \\ p = \mathit{Proc}(\gamma) \\ p \in \mathit{PartActs}(D)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{guard}, \sigma, \gamma)]}$$

(p-method-init)

$$\frac{\begin{array}{r}\mathbf{C}.D = (\textsc{idle}, \sigma, \gamma) \\ \gamma = (p, Q, \tilde{v}) \triangleright \hat{\gamma} \\ p \in \mathit{PartMeths}(D) \\ \tilde{y} = \mathit{InParam}(D.p) \\ \tilde{z} = \mathit{OutParam}(D.p) \\ \sigma' = \sigma\,[\tilde{z} \mapsto \tilde{\bot}]\,[\tilde{y} \mapsto \tilde{v}] \\ \gamma' = (p, Q) \triangleright \hat{\gamma}\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{guard}, \sigma', \gamma')]}$$

(t-action-init)

$$\frac{\begin{array}{r}\mathbf{C}.D = (\textsc{idle}, \sigma, \gamma) \\ p = \mathit{Proc}(\gamma) \\ p \in \mathit{TotActs}(D) \\ \theta' = \mathit{Code}(D.p)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma, \gamma, \theta')]}$$

(t-method-init)

$$\frac{\begin{array}{r}\mathbf{C}.D = (\textsc{idle}, \sigma, \gamma) \\ \gamma = (p, Q, \tilde{v}) \triangleright \hat{\gamma} \\ p \in \mathit{TotMeths}(D) \\ \tilde{y} = \mathit{InParam}(D.p) \\ \tilde{z} = \mathit{OutParam}(D.p) \\ \sigma' = \sigma\,[\tilde{z} \mapsto \tilde{\bot}]\,[\tilde{y} \mapsto \tilde{v}] \\ \gamma' = (p, Q) \triangleright \hat{\gamma} \\ \theta' = \mathit{Code}(D.p)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma', \gamma', \theta')]}$$

(guard-accept)

$$\frac{\mathbf{C}.D = (\textsc{guard}, \sigma, \gamma) \quad p = Proc(\gamma) \quad Alt(D.p, \sigma) = (\bot, \theta')}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma, \gamma, \theta')]}$$

(guard-reject)

$$\frac{\mathbf{C}.D = (\textsc{guard}, \sigma, \gamma) \quad p = Proc(\gamma) \quad Alt(D.p, \sigma) = \bot \quad \sigma' = \sigma \restriction BoxVars(D)}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{reject}, \sigma', \gamma)]}$$

(local-step)

$$\frac{\mathbf{C}.D = (\textsc{accept}, \sigma, \gamma, \theta) \quad (\sigma, \theta) \longrightarrow (\sigma', \theta')}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{accept}, \sigma', \gamma, \theta')]}$$

(proc-term)

$$\frac{\mathbf{C}.D = (\textsc{accept}, \sigma, \gamma, \bot) \quad \gamma = (p, Q) \triangleright \hat{\gamma} \quad \tilde{z} = OutParam(D.p) \quad \sigma' = \sigma \restriction BoxVars(D) \quad \tilde{v} = [\![\tilde{z}]\!]\sigma \quad \gamma' = (p, Q, \tilde{v}) \triangleright \hat{\gamma}}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{return}, \sigma', \gamma')]}$$

(action-end)

$$\frac{\mathbf{C}.D = (\textsc{return}, \sigma, \gamma) \quad \gamma = (a) \triangleright \gamma'}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{idle}, \sigma, \gamma')]}$$

(action-reject)

$$\frac{\mathbf{C}.D = (\textsc{reject}, \sigma, \gamma) \quad \gamma = (a) \triangleright \gamma'}{\mathbf{C} \Longrightarrow \mathbf{C}\,[D \mapsto (\textsc{idle}, \sigma, \gamma')]}$$

(test-accept)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\textsc{pwait}, \sigma_0, \gamma_0, \theta_0) \\ \theta_0 = E.m(\tilde{e}; \tilde{x}); \theta_0' \\ \mathbf{C}.E = (\textsc{return}, \sigma_1, \gamma_1) \\ \gamma_1 = (m, D, \tilde{v}) \triangleright \gamma_1' \\ \sigma_0' = \sigma_0\,[\tilde{x} \mapsto \tilde{v}]\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\textsc{accept}, \sigma_0', \gamma_0, \theta_0') \\ E \mapsto (\textsc{idle}, \sigma_1, \gamma_1') \end{bmatrix}}$$

(test-reject)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\textsc{pwait}, \sigma_0, \gamma_0, \theta_0) \\ \theta_0 = E.m(\tilde{e}; \tilde{x}); \hat{\theta} \\ \mathbf{C}.E = (\textsc{reject}, \sigma_1, \gamma_1) \\ \gamma_1 = (m, D) \triangleright \gamma_1' \\ \sigma_0' = \sigma_0 \upharpoonright \mathit{BoxVars}(D)\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\textsc{reject}, \sigma_0', \gamma_0) \\ E \mapsto (\textsc{idle}, \sigma_1, \gamma_1') \end{bmatrix}}$$

(total-return)

$$\frac{\begin{array}{c}\mathbf{C}.D = (\textsc{wait}, \sigma_0, \gamma_0, \theta_0) \\ \theta_0 = E.m(\tilde{e}; \tilde{x}); \theta_0' \\ \mathbf{C}.E = (\textsc{return}, \sigma_1, \gamma_1) \\ \gamma_1 = (m, D, \tilde{v}) \triangleright \gamma_1' \\ \sigma_0' = \sigma_0\,[\tilde{x} \mapsto \tilde{v}]\end{array}}{\mathbf{C} \Longrightarrow \mathbf{C}\begin{bmatrix} D \mapsto (\textsc{accept}, \sigma_0', \gamma_0, \theta_0') \\ E \mapsto (\textsc{idle}, \sigma_1, \gamma_1') \end{bmatrix}}$$

Appendix B

Additional proofs

B.1 Proofs for Chapter 3

B.1.1 Proof of Theorem 3.33

We show

$$\langle \forall\, \mathbf{C}, \mathbf{C}' \;:\; \mathbf{C} \in PC \;\wedge\; \mathbf{C} \Longrightarrow \mathbf{C}' \;:\; \mathbf{C}' \in PC \;\rangle$$

Assume that $\mathbf{C} \Longrightarrow \mathbf{C}'$ and that $\mathbf{C}$ is well-formed. We show that $\mathbf{C}'$ is locally well-formed, call correct, and well-founded.

$\mathbf{C}'$ is locally well-formed

By assumption, $\mathbf{C}$ is locally well-formed. From Theorem 3.32 all nonlocus boxes have the same configuration in $\mathbf{C}'$ as they do in $\mathbf{C}$, and they are thus well-formed. For the loci of the rule, we check that each of the conditions of Definition 3.5 is satisfied by the locus box configuration(s) in $\mathbf{C}'$.

For Condition 1, we note that, for this condition to be violated, the step from $\mathbf{C}$ to $\mathbf{C}'$ must leave a locus D with a phase other than IDLE, and an empty call queue. Since $\mathbf{C}.D$ is well-formed before the step, the step must involve a transition

from a nonempty queue to an empty queue, or a transition from IDLE to some other phase. The only rules that can change a nonempty call queue to an empty one are

(action-end)
(action-reject)
(test-accept)
(test-reject)
(total-return)

In each of these rules, the source is left in phase IDLE. The only rules that move a box out of the idle phase are

(p-action-init)
(p-method-init)
(t-action-init)
(t-method-init)

In each of these, the call queue must be nonempty before the rule, and it is not changed by the rule.

For Condition 2, we note that every new box configuration introduced by the rules obeys this condition.

For Condition 3, we note that the only rules that leave a box in phase PWAIT or WAIT are

(total-call)
(guard-test)

and that both of these rules leave the call with a method call at the head of its code.

For Condition 4, we let

$$P = \{\textsc{guard}, \textsc{pwait}, \textsc{reject}\}$$

We note that the rules that leave a box's phase in P are

(p-proc-init)

(guard-reject)

(guard-test)

(test-reject)

We note that none of these changes the procedure in the first entry in the call queue. Each of the rules, other than the first, requires that its locus's phase be in P before the step, and the first rule is only applicable if the procedure in the first entry in the call queue is partial.

For Condition 5, we note that the rules that leave a box in phase RETURN or REJECT are

(proc-term)

(guard-reject)

(test-reject)

and that these set the domain of the local state for the box to the required value. The rules that leave a box in phase IDLE are

(action-end)

(action-reject)

(test-accept)

(test-reject)

(total-return)

In each of these rules, the agent is in phase RETURN or IDLE prior to the step.

For Condition 6, We note that the only changes to the domain of the local state are to extend it with parameters, or to reset it to the box variables.

For Condition 7, we note that the program is assumed to be well-formed. In particular, no calls to actions appear in the code for a box, and no box calls a method on itself. So if an entry is put on a call queue with a source other than $\perp$, the call is for an action on the agent. Rule **(action-start)** is the only rule that can put a call for an action in a call queue, and this rule correctly gives the source as $\perp$.

For Condition 8, we note that no rule increases the number of call queue entries by more than one. So if box $\mathbf{C}'.D.\gamma$ has more than one entry for box E, there is an entry for E in $\mathbf{C}.D.\gamma$. Since $\mathbf{C}$ is call correct, this means that $\mathbf{C}.E.\phi \in \{\text{PWAIT}, \text{WAIT}\}$. There is no rule that allows box E to place an entry in a call queue when it is one of these phases, so there can be at most one entry for E in D's call queue.

For Condition 9, We note that an entry in the call queue is not changed until it becomes the current procedure call, so we need only show that every entry obeys this condition when it is added to the queue. For actions, rule **(action-start)** sets the parameter values to $\perp$, which satisfies the condition. For a method call, the rules **(total-call)**, and **(guard-test)**, evaluate the parameter values from the expressions in the method call statement, and by the type rules, these are of the correct type.

For Condition 10, we note that the only rule that put a box in phase RETURN is **(proc-term)**, and this rule sets the parameter values to a list that matches the type of the output parameters.

$\mathbf{C}'$ is call correct

We note that no rule changes the procedure name or source component of any call queue entry, so we confine our attention to the rules that add and remove call queue

entries for methods. The rules that add method call entries are

(total-call)

(guard-test)

For both of these, the source is not in a waiting phase in $\mathbf{C}$, but it is in $\mathbf{C}'$, and there is a new entry in the agent's call queue in $\mathbf{C}'$. We note that this is the correct type of procedure, since **(total-call)** is applicable only when a method call statement is encountered in the body code of the procedure, and, by the type rules, this must be a total method, and **(guard-test)** is applicable only when the alternative function returns a test, and, again by the type rules, this must be a partial method.

$\mathbf{C}'$ is well-founded

We note from the rules that $\mathrm{dom}(K.\mathbf{C}')$ is either the same as $\mathrm{dom}(K.\mathbf{C})$, or it has one member more or less, and that if $D \in \mathrm{dom}(K.\mathbf{C}')$, then $D \notin \mathrm{dom}(K.\mathbf{C})$, or $K.\mathbf{C}'.D = K.\mathbf{C}.D$. That is, no rule changes the "is executing for" relation other than to add or remove an entry. Thus we only need check rules that change the domain of this function.

From the above, we have that $\mathbf{C}'$ is locally well-formed and call correct, so Theorem 3.15 gives us that a step that removes an entry in $K.\mathbf{C}$, removes an entry for a D such that $D \notin \mathrm{rng}(K.\mathbf{C})$. Thus, if $\mathrm{dom}(K.\mathbf{C}')$ has one fewer entry than $\mathrm{dom}(K.\mathbf{C})$, the entry removed is at the beginning of a call stack, and thus all values used to reach $\perp$ for all the boxes in $\mathrm{dom}(K.\mathbf{C}')$ are still present.

The rules that add a member to $\mathrm{dom}(K.\mathbf{C})$ are

(p-proc-init)

(t-action-init)

(t-method-init)

Suppose one of these rules is applied with locus D. We have

$$
\begin{array}{ll}
 & \mathit{true} \\
\equiv & \quad \{ \text{ Theorem 3.15 } \} \\
 & K.\mathbf{C}'.D = \bot \ \lor \ K.\mathbf{C}'.D \in \mathrm{dom}(K.\mathbf{C}') \\
\equiv & \quad \{ \ \mathbf{C}'.D \text{ is well-formed, so } K.\mathbf{C}'.D \neq D \ \} \\
 & K.\mathbf{C}'.D = \bot \ \lor \ K.\mathbf{C}'.D \in \mathrm{dom}(K.\mathbf{C}) \\
\equiv & \quad \{ \ \mathbf{C} \text{ is well-founded } \} \\
 & K.\mathbf{C}'.D = \bot \ \lor \ \langle \exists\, k \ : \ k > 0 \ : \ (K.\mathbf{C})^k.(K.\mathbf{C}'.D) = \bot \ \rangle \\
\Rightarrow & \quad \{ \ K.\mathbf{C}.E = K.\mathbf{C}'.E \text{ for } E \in \mathrm{dom}(K.\mathbf{C}) \ \} \\
 & K.\mathbf{C}'.D = \bot \ \lor \ \langle \exists\, k \ : \ k > 0 \ : \ (K.\mathbf{C}')^k.(K.\mathbf{C}'.D) = \bot \ \rangle \\
\equiv & \quad \{ \text{ Definition 3.17, twice } \} \\
 & (K.\mathbf{C}')^1.D = \bot \ \lor \ \langle \exists\, k \ : \ k > 0 \ : \ (K.\mathbf{C}')^{k+1}.D = \bot \ \rangle \\
\equiv & \quad \{ \text{ one-point rule; change range } \} \\
 & \langle \exists\, k \ : \ k = 1 \ : \ (K.\mathbf{C}')^k.D = \bot \ \rangle \ \lor \\
 & \langle \exists\, k \ : \ k > 1 \ : \ (K.\mathbf{C}')^k.D = \bot \ \rangle \\
\equiv & \quad \{ \text{ combine ranges } \} \\
 & \langle \exists\, k \ : \ k > 0 \ : \ (K.\mathbf{C}')^k.D = \bot \ \rangle
\end{array}
$$

Thus $\mathbf{C}'$ is well-founded.

B.1.2 Proof of Theorem 3.51

We use the following lemmas.

Lemma B.1

$$
\begin{array}{l}
\langle \forall \ \mathbf{C}, \mathbf{L}, \mathbf{L}' \\
\quad : \ \mathbf{C} \in PC \ \land \ \mathbf{L}, \mathbf{L}' \in \mathit{Lab} \ \land \ \mathbf{L} \neq \mathbf{L}' \ \land \ \mathbf{L},\ \mathbf{L}' \ \mathit{enabled\ in}\ \mathbf{C} \\
\quad : \ \mathit{CLoci}(\mathbf{L}) \ \mathrm{disj} \ \mathit{CLoci}(\mathbf{L}') \\
\quad \rangle
\end{array}
$$

Proof

Assume we have $\mathbf{C} \in PC$, and $\mathbf{L}, \mathbf{L}' \in Lab$, where $\mathbf{L} \neq \mathbf{L}'$, and $\mathbf{L}$, $\mathbf{L}'$ are enabled in $\mathbf{C}$. We have

$$
\begin{array}{ll}
 & D \in CLoci(\mathbf{L}) \\
\equiv & \quad \{ \text{ Definition 3.45 } \} \\
 & \mathbf{L} \in Cond(D) \\
\Rightarrow & \quad \{ \ \mathbf{L}, \mathbf{L}' \text{ enabled in } \mathbf{C}, \mathbf{L} \neq \mathbf{L}', \text{ Theorem 3.47 } \} \\
 & \mathbf{L}' \notin Cond(D) \\
\equiv & \quad \{ \text{ Definition 3.45 } \} \\
 & D \notin CLoci(\mathbf{L}')
\end{array}
$$

Thus $CLoci(\mathbf{L})$ disj $CLoci(\mathbf{L}')$, as required.

(End of proof)

Lemma B.2

$$
\begin{array}{l}
\langle \forall \ \mathbf{L}, \mathbf{C}, \mathbf{C}' \\
\ : \mathbf{L} \in Lab \ \wedge \ \mathbf{C}, \mathbf{C}' \in PC \ \wedge \ \langle \forall \ D \ : \ D \in CLoci(\mathbf{L}) \ : \ \mathbf{C}.D \sqsubseteq \mathbf{C}'.D \ \rangle \\
\ : \mathbf{L} \ \textit{is enabled in} \ \mathbf{C} \ \Rightarrow \ \mathbf{L} \ \textit{is enabled in} \ \mathbf{C}' \\
\ \rangle
\end{array}
$$

Proof

Suppose $\mathbf{L} \in Lab$, $\mathbf{C} \in PC$. From Theorem 3.46, only the configurations of its conditional loci are relevant for determining if $\mathbf{L}$ is enabled in $\mathbf{C}$. Suppose $D \in CLoci(\mathbf{L})$, and let $\mathbf{C}.D = (\phi, \sigma, \gamma, \theta)$, and $\gamma \sqsubseteq \gamma'$. We have

$$
\begin{array}{ll}
 & \mathbf{L} \text{ is enabled in } \mathbf{C} \\
\Rightarrow & \quad \{ \text{ Theorem 3.50 } \} \\
 & \neg(\mathbf{C}.D.\phi = \text{IDLE} \ \wedge \ \mathbf{C}.D.\gamma = \bot) \\
\equiv & \quad \{ \ \mathbf{C}.D \in BC(D), \text{ Definition 3.5 } \}
\end{array}
$$

$$\begin{array}{ll} & \mathbf{C}.D.\gamma \neq \bot \\ \Rightarrow & \{ \ \gamma \sqsubseteq \gamma' \ \} \\ & \text{first}(\gamma) = \text{first}(\gamma') \end{array}$$

Thus, if $\mathbf{L}$ is enabled in $\mathbf{C}$, then $\text{first}(\gamma) = \text{first}(\gamma')$. We observe from Table 3.3 that in each case where γ is used in the condition, $\text{first}(\gamma)$ is the only element of γ that is used, either directly, or in $Proc(\gamma)$, or via functions $K.\mathbf{C}$ and $W.\mathbf{C}$, or their subfunctions. Thus, if $\mathbf{L}$ is enabled in $\mathbf{C}$, and

$$\mathbf{C}' \quad = \quad \mathbf{C}\,[D \mapsto (\phi, \sigma, \gamma', \theta)]$$

then $\mathbf{L}$ is enabled in $\mathbf{C}'$. If $\mathbf{L}$ has a single conditional locus we are done. If not, we apply this argument twice, once for each conditional locus.
(End of proof)

Lemma B.3

$$\langle \forall\, \mathbf{L}, \mathbf{C}, \mathbf{C}', D \ : \ \mathbf{L} \in Lab \ \wedge \ \mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}' \ \wedge \ D \in ULoci(\mathbf{L}) \ : \ \mathbf{C}.D \sqsubseteq \mathbf{C}'.D \ \rangle$$

Proof

From the rules, we see that in each of the cases where D is an unconditional locus for step $\mathbf{L}$, the only change to D's configuration after the step is that an entry has been added to the end of its call queue, so the value of the call queue before the step is a prefix of its value after the step.
(End of proof)

Lemma B.4

$$\langle \forall\, \mathbf{L}, \mathbf{C}, \mathbf{C}', D \ : \ \mathbf{L} \in Lab \ \wedge \ \mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}' \ \wedge \ D \notin CLoci(L) \ : \ \mathbf{C}.D \sqsubseteq \mathbf{C}'.D \ \rangle$$

Proof

Assume $\mathbf{L} \in \mathit{Lab}$, and $\mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}'$. We have

$$
\begin{array}{ll}
 & D \notin \mathit{CLoci}(\mathbf{L}) \\
\equiv & \quad \{ \text{ Definition 3.45 } \} \\
 & D \notin \mathit{Loci}(\mathbf{L}) \;\vee\; D \in \mathit{ULoci}(\mathbf{L}) \\
\Rightarrow & \quad \{ \; \mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}', \text{ Theorem 3.32, Lemma B.3 } \} \\
 & \mathbf{C}.D = \mathbf{C}'.D \;\vee\; \mathbf{C}.D \sqsubseteq \mathbf{C}'.D \\
\equiv & \quad \{ \sqsubseteq \text{ is reflexive } \} \\
 & \mathbf{C}.D \sqsubseteq \mathbf{C}'.D
\end{array}
$$

(End of proof)

For the proof of Theorem 3.51, we show

$$
\begin{array}{l}
\langle \forall \; \mathbf{L}, \mathbf{C}, \mathbf{C}', D \\
\quad : \mathbf{L} \in \mathit{Lab} \;\wedge\; \mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}' \;\wedge\; D \in \mathbf{B} \;\wedge\; \mathit{Enabled}(\mathbf{C}, D) \notin \{\bot, \mathbf{L}\} \\
\quad : \mathit{Enabled}(\mathbf{C}, D) = \mathit{Enabled}(\mathbf{C}', D) \\
\quad \rangle
\end{array}
$$

Assume we have $\mathbf{L} \in \mathit{Lab}$, $\mathbf{C}$ and $\mathbf{C}'$ such that $\mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}'$, and $D \in \mathbf{B}$, and let $\mathbf{L}' = \mathit{Enabled}(\mathbf{C}, D)$, where $\mathbf{L}' \notin \{\bot, \mathbf{L}\}$. Note that the result will follow if we can show that $\mathbf{L}'$ is enabled in $\mathbf{C}'$, since at most one conditional step for D is enabled in $\mathbf{C}'$.

$$
\begin{array}{ll}
 & D \in \mathit{CLoci}(\mathbf{L}') \\
\Rightarrow & \quad \{ \; \mathbf{L} \neq \mathbf{L}', \mathbf{L} \text{ and } \mathbf{L}' \text{ enabled in } \mathbf{C}, \text{ Lemma B.1 } \} \\
 & D \notin \mathit{CLoci}(\mathbf{L}) \\
\Rightarrow & \quad \{ \text{ Lemma B.4 } \} \\
 & \mathbf{C}.D \sqsubseteq \mathbf{C}'.D
\end{array}
$$

Thus we have $\langle \forall \; D \;:\; D \in \mathit{CLoci}(\mathbf{L}') \;:\; \mathbf{C}.D \sqsubseteq \mathbf{C}'.D \;\rangle$. Since $\mathbf{L}'$ is enabled in $\mathbf{C}$, Lemma B.2 implies $\mathbf{L}'$ is enabled in $\mathbf{C}'$, as required.

B.1.3 Proof of Theorem 3.14

We show

$$
\begin{array}{l}
\langle \forall\ \mathbf{C}, D, E \\
\quad :\ \mathbf{C} \in \mathit{ProgConfig}\ \wedge\ \mathbf{C} \text{ is locally well-formed and call correct } \wedge \\
\qquad D, E \in \mathrm{dom}(K.\mathbf{C})\ \wedge\ K.\mathbf{C}.D = K.\mathbf{C}.E \\
\quad :\ K.\mathbf{C}.D = \bot\ \vee\ D = E \\
\quad \rangle
\end{array}
$$

Assume $\mathbf{C} \in \mathit{ProgConfig}$ is locally well-formed and call correct, $D, E \in \mathrm{dom}(K.\mathbf{C})$, and $K.\mathbf{C}.D = K.\mathbf{C}.E$. We have

$$
\begin{array}{ll}
 & K.\mathbf{C}.D \neq \bot \\
\Rightarrow & \qquad \{ \text{ Let } D' = \mathit{Source}(\mathbf{C}.D.\gamma);\ K.\mathbf{C}.D = K.\mathbf{C}.E \ \} \\
 & \mathit{Source}(\mathbf{C}.D.\gamma) = D'\ \wedge\ \mathit{Source}(\mathbf{C}.E.\gamma) = D' \\
\Rightarrow & \qquad \{ \text{ Definition 2.23 } \} \\
 & \langle \exists\ m\ ::\ (m, D') \in \mathbf{C}.D.\gamma\ \rangle\ \wedge\ \langle \exists\ m'\ ::\ (m', D') \in \mathbf{C}.E.\gamma\ \rangle \\
\Rightarrow & \qquad \{ \text{ Theorem 3.12, } D' \in \mathbf{B} \ \} \\
 & D = E
\end{array}
$$

Thus

$$K.\mathbf{C}.D \neq \bot \ \Rightarrow\ D = E$$

which is equivalent to

$$K.\mathbf{C}.D = \bot\ \vee\ D = E$$

as required.

B.1.4 Proof of Theorem 3.15

We use the following lemmas.

Lemma B.5

$$
\begin{aligned}
&\langle \forall\ \mathbf{C}, D \\
&\quad :\ \mathbf{C} \in \mathit{ProgConfig}\ \ \wedge\ \ D \in \mathrm{dom}(K.\mathbf{C}) \\
&\quad :\ K.\mathbf{C}.D = \mathit{Source}(\mathbf{C}.D.\gamma) \\
&\quad \rangle
\end{aligned}
$$

Proof

Immediate from the definitions.

(End of proof)

Lemma B.6

$$
\begin{aligned}
&\langle \forall\ \mathbf{C}, D \\
&\quad :\ \mathbf{C} \in \mathit{ProgConfig}\ \ \wedge\ \ \mathbf{C}\ \textit{is locally well-formed}\ \ \wedge\ \ D \in \mathbf{B} \\
&\quad :\ D \in \mathrm{dom}(K.\mathbf{C})\ \ \not\equiv\ \ \mathbf{C}.D.\phi = \textsc{idle} \\
&\quad \rangle
\end{aligned}
$$

Proof

Assume $\mathbf{C} \in \mathit{ProgConfig}$, $\mathbf{C}$ is locally well-formed, and $D \in \mathbf{B}$.

$$
\begin{aligned}
& \mathbf{C}.D.\phi \neq \textsc{idle} \\
\equiv \quad & \qquad \{\ \mathbf{C} \text{ is locally well-formed, so } \mathbf{C}.D \text{ is well-formed}\ \} \\
& \mathbf{C}.D.\phi \neq \textsc{idle}\ \ \wedge\ \ \mathbf{C}.D.\gamma \neq \bot \\
\equiv \quad & \qquad \{\ \text{Definitions 2.23, 2.19 3.5, and Assumption 2.18}\ \} \\
& \mathbf{C}.D.\phi \neq \textsc{idle}\ \wedge \\
& (\mathit{Proc}(\mathbf{C}.D.\gamma) \in \mathit{PartMeths}(D)\ \vee\ \mathit{Proc}(\mathbf{C}.D.\gamma) \in \mathit{TotMeths}(D))
\end{aligned}
$$

$\equiv$ { distribute }

$(\mathbf{C}.D.\phi \neq \text{IDLE} \ \wedge \ \mathit{Proc}(\mathbf{C}.D.\gamma) \in \mathit{PartMeths}(D)) \ \vee$

$(\mathbf{C}.D.\phi \neq \text{IDLE} \ \wedge \ \mathit{Proc}(\mathbf{C}.D.\gamma) \in \mathit{TotMeths}(D))$

$\equiv$ { Definition 3.13 }

$D \in \text{dom}(Kp.\mathbf{C}) \ \vee \ D \in \text{dom}(Kt.\mathbf{C})$

$\equiv$ { Definition 3.13 }

$D \in \text{dom}(K.\mathbf{C})$

(End of proof)

For the proof of Theorem 3.15, we show

$\langle \forall \ \mathbf{C}, D$

$: \mathbf{C} \in \mathit{ProgConfig} \ \wedge$ $\mathbf{C}$ is locally well-formed and call correct $\wedge$

$D \in \text{dom}(K.\mathbf{C})$

$: K.\mathbf{C}.D = \bot \ \vee \ K.\mathbf{C}.D \in \text{dom}(K.\mathbf{C})$

$\rangle$

Assume $\mathbf{C} \in \mathit{ProgConfig}$ is locally well-formed and call correct, and $D \in \text{dom}(K.\mathbf{C})$. As with Theorem 3.14, we show

$$K.\mathbf{C}.D \neq \bot \ \Rightarrow \ K.\mathbf{C}.D \in \text{dom}(K.\mathbf{C})$$

Assume $K.\mathbf{C}.D \neq \bot$, and let $K.\mathbf{C}.D = E$, for some $E \in \mathbf{B}$. We have

true

$\equiv$ { $K.\mathbf{C}.D = E$, Lemma B.5 }

$E = \mathit{Source}(\mathbf{C}.D.\gamma)$

$\Rightarrow$ { Definition 2.23 }

$\langle \exists \ m \ :: \ (m, E) \in \mathbf{C}.D.\gamma \ \rangle$

$\Rightarrow$ { $\mathbf{C}$ is call correct, $E \in \mathbf{B}$ }

$\mathbf{C}.E.\phi \in \{\text{PWAIT}, \text{WAIT}\}$

$\Rightarrow$ $\{$ $\mathbf{C}$ is locally well-formed, Lemma B.6 $\}$

$E \in \text{dom}(K.\mathbf{C})$

B.1.5 Proof of Theorem 3.16

We use the following lemmas.

Lemma B.7

$$\langle \forall\ \mathbf{C}\ :\ \mathbf{C} \in \mathit{ProgConfig}\ :\ \text{dom}(Kp.\mathbf{C}) \cap \text{dom}(Kt.\mathbf{C}) = \emptyset\ \rangle$$

Proof

Immediate from the definitions, since $\mathit{PartMeths}(D) \cap \mathit{TotMeths}(D) = \emptyset$.

(End of proof)

Lemma B.8

$$\begin{aligned}
&\langle \forall\ \mathbf{C}, D \\
&\quad :\ \mathbf{C} \in \mathit{ProgConfig}\ \wedge\ \mathbf{C}\ \textit{is locally well-formed and call correct} \\
&\quad\quad \wedge\ D \in \mathbf{B} \\
&\quad :\ (D \in \text{rng}(Kp.\mathbf{C})\ \Rightarrow\ \mathbf{C}.D.\phi = \text{PWAIT})\ \wedge \\
&\quad\quad (D \in \text{rng}(Kt.\mathbf{C})\ \Rightarrow\ \mathbf{C}.D.\phi = \text{WAIT}) \\
&\quad \rangle
\end{aligned}$$

Proof

Assume $\mathbf{C} \in \mathit{ProgConfig}$ is locally well-formed and call correct, and $D \in \mathbf{B}$. We prove the two conjuncts of the term separately. For the first part, assume $D = Kp.\mathbf{C}.E$, for some $E \in \mathbf{B}$.

true

$\equiv$ { $Kp.\mathbf{C}.E = D$, Definition 3.13 }

$\langle \exists \ m, \tilde{v}$

$: m \in \mathit{PartMeths}(E) \ \wedge \ \tilde{v} \in \mathit{Val}^*$

$: \text{first}(\mathbf{C}.E.\gamma) = (m, D, \tilde{v})$

$\rangle$

$\Rightarrow$ { Definition 3.1 }

$\langle \exists \ m \ : \ m \in \mathit{PartMeths}(E) \ : \ (m, D) \in \mathbf{C}.E.\gamma \ \rangle$

$\Rightarrow$ { $\mathbf{C}$ is call correct }

$\mathbf{C}.D.\phi = \text{PWAIT}$

The proof for the second conjunct is similar.

(End of proof)

For the proof of Theorem 3.16, we show

$\langle \forall \ \mathbf{C}$

$: \mathbf{C} \in \mathit{ProgConfig} \ \wedge \ \mathbf{C}$ is locally well-formed and call correct

$: \text{rng}(Kp.\mathbf{C}) \ \text{ disj } \ \text{dom}(Kt.\mathbf{C})$

$\rangle$

Assume $\mathbf{C} \in \mathit{ProgConfig}$ is locally well-formed and call correct. Since

$$\text{dom}(Kt.\mathbf{C}) \ \subseteq \ \mathbf{B}$$

it is sufficient to show that, for any $D \in \mathbf{B}$,

$$D \in \text{rng}(Kp.\mathbf{C}) \ \Rightarrow \ D \notin \text{dom}(Kt.\mathbf{C})$$

Assume $D \in \mathbf{B}$. We have

$D \in \text{rng}(Kp.\mathbf{C})$

$\Rightarrow$ { Lemma B.8 }

$$
\begin{aligned}
& \mathbf{C}.D.\phi = \text{PWAIT} \\
\equiv \quad & \quad \{\ \mathbf{C}.D \text{ is well-formed }\} \\
& \mathbf{C}.D.\phi = \text{PWAIT} \ \wedge \ \mathit{Proc}(\mathbf{C}.D.\gamma) \in \mathit{Partials}(D) \\
\Rightarrow \quad & \quad \{\ \text{Definition 3.13 }\} \\
& D \in \text{dom}(Kp.\mathbf{C}) \\
\Rightarrow \quad & \quad \{\ \text{Lemma B.7 }\} \\
& D \notin \text{dom}(Kt.\mathbf{C})
\end{aligned}
$$

B.2 Proofs for Chapter 4

B.2.1 Proof of Theorem 4.7

We show

$$
\begin{aligned}
& \langle \forall\ \varepsilon \\
& \quad :\ \varepsilon \in \mathit{Complete}(Z) \\
& \quad :\ \langle \forall\ D, i \\
& \qquad :\ D \in \mathbf{B} \ \wedge \ 0 \le i \le |\varepsilon| \\
& \qquad :\ \langle \exists\ j \ :\ i \le j \le |\varepsilon| \ :\ \varepsilon[j].D.\phi = \text{IDLE} \ \rangle \\
& \qquad \rangle \\
& \quad \rangle
\end{aligned}
$$

Assume $\varepsilon \in \mathit{Complete}(Z)$, $D \in \mathbf{B}$, and $0 \le i < |\varepsilon|$. If $\varepsilon[i].D.\phi = \text{IDLE}$, then we can choose $j = i$, and we are done. Assume $\varepsilon[i].D.\phi \neq \text{IDLE}$. If E is the root box for the current call in D in $\varepsilon[i]$, then E is currently executing an action call. Since ε is complete, this action call completes at some point. Before it does so, the current procedure call in D must complete. If the last step for this procedure call is step k, then $\varepsilon[k+1].D.\phi = \text{IDLE}$. Choose $j = k + 1$.

B.2.2 Proof of Theorem 4.8

We use the following lemmas.

Lemma B.9

$$\langle \forall\, \mathbf{C} \;:\; \mathbf{C} \in PC \;:\; \mathit{Actives}(\mathbf{C}) = \emptyset \;\equiv\; qt(\mathbf{C}) \;\rangle$$

Proof

Assume $\mathbf{C} \in PC$. If $qt(\mathbf{C})$, then all call queues are empty, so there are no active action calls. If there are no active action calls, since $\mathbf{C}$ is well-founded, there can be no active method calls. Thus all call queues are empty, and $qt(\mathbf{C})$.
(End of proof)

Definition B.10 *For $D \in \mathbf{B}$,*

$$\mathbf{A}(D) \;\triangleq\; \{\, \alpha \mid \alpha \in \mathbf{A} \;\wedge\; \mathit{Box}(\alpha) = D \,\}$$

Lemma B.11

$$\begin{array}{l} \langle \forall\, \varepsilon, D \\ \quad : \varepsilon \in Z \;\wedge\; |\varepsilon| < \infty \;\wedge\; qt(\mathit{Start}(\varepsilon)) \;\wedge\; D \in \mathbf{B} \\ \quad : \mathit{NumStarts}(\varepsilon, D) = \mathit{NumEnds}(\varepsilon, D) \;\equiv\; \mathit{Actives}(\mathit{Final}(\varepsilon)) \;\mathrm{disj}\; \mathbf{A}(D) \\ \quad \rangle \end{array}$$

Proof

Assume $\varepsilon \in Z$, $|\varepsilon| < \infty$, $qt(\mathit{Start}(\varepsilon))$, and $D \in \mathbf{B}$. Each **action-start**(D) step in ε adds a call for $\alpha \in \mathbf{A}(D)$ to D's call queue, and each **action-end**(D) or **action-reject**(D) step removes a call for an $\alpha \in \mathbf{A}(D)$ from D's call queue. No other steps add call for actions to D's call queue. Thus $\mathit{NumStarts}(\varepsilon, D) - \mathit{NumEnds}(\varepsilon, D)$ is the number of action calls in D's call queue at the end of ε. The result follows

from this.

(End of proof)

For the proof of Theorem 4.8, we show

$$\langle \forall\, \varepsilon \;:\; \varepsilon \in Z \,\wedge\, |\varepsilon| < \infty \;:\; \mathit{Complete}(\varepsilon) \;\equiv\; qt(\mathit{Start}(\varepsilon)) \,\wedge\, qt(\mathit{Final}(\varepsilon)) \;\rangle$$

Assume $\varepsilon \in Z$, and $|\varepsilon| < \infty$. We have

$$\begin{array}{ll}
& \mathit{Complete}(\varepsilon) \\
\equiv & \quad \{ \text{ Definition 4.6 } \} \\
& \mathit{Proper}(\varepsilon) \;\wedge\; \langle \forall\, D \;:\; D \in \mathbf{B} \;:\; \mathit{NumStarts}(\varepsilon, D) = \mathit{NumEnds}(\varepsilon, D) \;\rangle \\
\equiv & \quad \{ \; \mathit{Proper}(\varepsilon) \Rightarrow qt(\mathit{Start}(\varepsilon)),\ |\varepsilon| < \infty, \text{ Lemma B.11 } \} \\
& \mathit{Proper}(\varepsilon) \;\wedge\; \langle \forall\, D \;:\; D \in \mathbf{B} \;:\; \mathit{Actives}(\mathit{Final}(\varepsilon)) \;\text{ disj }\; \mathbf{A}(D) \;\rangle \\
\equiv & \quad \{ \text{ Property of disj } \} \\
& \mathit{Proper}(\varepsilon) \;\wedge\; \mathit{Actives}(\mathit{Final}(\varepsilon)) \;\text{ disj }\; \langle \cup\, D \;:\; D \in \mathbf{B} \;:\; \mathbf{A}(D) \;\rangle \\
\equiv & \quad \{ \text{ Definition B.10 } \} \\
& \mathit{Proper}(\varepsilon) \;\wedge\; \mathit{Actives}(\mathit{Final}(\varepsilon)) \;\text{ disj }\; \mathbf{A} \\
\equiv & \quad \{ \; \mathit{Actives}(\mathit{Final}(\varepsilon)) \subseteq \mathbf{A} \; \} \\
& \mathit{Proper}(\varepsilon) \;\wedge\; \mathit{Actives}(\mathit{Final}(\varepsilon)) = \emptyset \\
\equiv & \quad \{ \text{ Lemma B.9 } \} \\
& \mathit{Proper}(\varepsilon) \;\wedge\; qt(\mathit{Final}(\varepsilon)) \\
\equiv & \quad \{ \text{ Definition 4.4, } |\varepsilon| < infty \; \} \\
& qt(\mathit{Start}(\varepsilon)) \;\wedge\; qt(\mathit{Final}(\varepsilon)) \;\wedge \\
& \langle \forall\, D \;:\; D \in \mathbf{B} \;:\; \mathit{Enabled}(\mathit{Final}(\varepsilon), D) = \bot \;\rangle \\
\equiv & \quad \{ \text{ Second conjunct implies third, by Theorem 3.50 } \} \\
& qt(\mathit{Start}(\varepsilon)) \;\wedge\; qt(\mathit{Final}(\varepsilon))
\end{array}$$

B.2.3 Proof of Theorem 4.12

We use the following lemmas.

Lemma B.12

$$
\begin{aligned}
&\langle \forall\ \mathbf{C}, \mathbf{C}', D, E \\
&\quad :\ \mathbf{C}, \mathbf{C}' \in PC \ \wedge\ \mathbf{C} \Longrightarrow \mathbf{C}' \ \wedge\ W.\mathbf{C}.D = E \ \wedge\ E \in \mathrm{dom}(W.\mathbf{C}) \\
&\quad :\ W.\mathbf{C}'.D = E \\
&\quad \rangle
\end{aligned}
$$

Proof

Assume $\mathbf{C}, \mathbf{C}' \in PC$, and $\mathbf{L} \in \mathit{Lab}$, such that $\mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}'$. Also assume $W.\mathbf{C}.D = E$, and $E \in \mathrm{dom}(W.\mathbf{C})$. We have

$$
\begin{aligned}
& \mathit{true} \\
\equiv & \qquad \{ \text{ assumption } \} \\
& D, E \in \mathrm{dom}(W.\mathbf{C}) \\
\Rightarrow & \qquad \{ \text{ Definition 3.27 } \} \\
& \mathbf{C}.D.\phi, \mathbf{C}.E.\phi \in \{\textsc{pwait}, \textsc{wait}\} \\
\equiv & \qquad \{ \text{ assumption } \} \\
& W.\mathbf{C}.D = E \ \wedge\ \mathbf{C}.D.\phi, \mathbf{C}.E.\phi \in \{\textsc{pwait}, \textsc{wait}\} \\
\Rightarrow & \qquad \{ \text{ Theorem 3.50, Definition 3.48 } \} \\
& \mathit{Enabled}(\mathbf{C}, D) = \bot \\
\Rightarrow & \qquad \{ \ \mathbf{L} \text{ enabled in } \mathbf{C} \ \} \\
& D \notin \mathit{CLoci}(\mathbf{L}) \\
\Rightarrow & \qquad \{ \ \mathbf{C}\langle\mathbf{L}\rangle\mathbf{C}', \text{ Lemma B.4 } \} \\
& \mathbf{C}.D \sqsubseteq \mathbf{C}'.D \\
\Rightarrow & \qquad \{ \text{ Definition 3.6 } \} \\
& \mathbf{C}.D.\phi = \mathbf{C}'.D.\phi \ \wedge\ \mathbf{C}.D.\gamma = \mathbf{C}'.D.\gamma
\end{aligned}
$$

$$\Rightarrow \qquad \{ \text{ Definition 3.27 } \}$$

$$W.\mathbf{C}'.D = W.\mathbf{C}.D$$

$$\equiv \qquad \{ \ W.\mathbf{C}.D = E \ \}$$

$$W.\mathbf{C}'.D = E$$

(End of proof)

Lemma B.13

$$\begin{array}{l} \langle \forall \ \varepsilon, T, k \\ \quad : \varepsilon \in Z \ \wedge \ T \in \mathbf{B}^{+} \ \wedge \ 0 \leq k \leq |\varepsilon| \ \wedge \ T \ \textit{is a knot in} \ \varepsilon[k] \\ \quad : \langle \forall \, i \ : \ k \leq i \leq |\varepsilon| \ : \ T \ \textit{is a knot in} \ \varepsilon[i] \ \rangle \\ \quad \rangle \end{array}$$

Proof

Assume $\varepsilon \in Z$, $T \in \mathbf{B}^{+}$, and T is a knot in $\varepsilon[k]$. We prove the result by induction on i. For the basis, $i = k$, we have the result by assumption. For the induction step, we have, for $k \leq i < |\varepsilon|$,

$$T \text{ is a knot in } \varepsilon[i]$$

$$\equiv \qquad \{ \text{ Definition 4.11 } \}$$

$$\langle \forall \, n \ : \ 0 \leq n < |T| \ : \ W.\varepsilon[i].T[n] = T[n \oplus 1] \ \rangle$$

$$\equiv \qquad \{ \text{ definition of domain } \}$$

$$\begin{array}{l} \langle \forall \ n \\ \quad : 0 \leq n < |T| \\ \quad : W.\varepsilon[i].T[n] = T[n \oplus 1] \ \wedge \ T[n] \in \mathrm{dom}(W.\varepsilon[i]) \\ \quad \rangle \end{array}$$

$$\equiv \qquad \{ \ \forall \text{ over } \wedge \ \}$$

$$\langle \forall \, n \ : \ 0 \leq n < |T| \ : \ W.\varepsilon[i].T[n] = T[n \oplus 1] \ \rangle \ \wedge$$
$$\langle \forall \, n \ : \ 0 \leq n < |T| \ : \ T[n] \in \mathrm{dom}(W.\varepsilon[i]) \ \rangle$$

$\equiv$ { range is not empty }

$$\langle \forall\ n \\ :\ 0 \le n < |T| \\ :\ W.\varepsilon[i].T[n] = T[n \oplus 1]\ \wedge \\ \quad \langle \forall\ n\ :\ 0 \le n < |T|\ :\ T[n] \in \mathrm{dom}(W.\varepsilon[i])\ \rangle \\ \rangle$$

$\Rightarrow$ { instantiate }

$$\langle \forall\ n \\ :\ 0 \le n < |T| \\ :\ W.\varepsilon[i].T[n] = T[n \oplus 1]\ \wedge\ T[n \oplus 1] \in \mathrm{dom}(W.\varepsilon[i]) \\ \rangle$$

$\Rightarrow$ { Lemma B.12 }

$$\langle \forall\ n\ :\ 0 \le n < |T|\ :\ W.\varepsilon[i+1].T[n] = T[n \oplus 1]\ \rangle$$

$\equiv$ { Definition 4.11 }

T is a knot in $\varepsilon[i+1]$

(End of proof)

Lemma B.14

$$\langle \forall\ \varepsilon, k, D, E \\ :\ \varepsilon \in Z\ \wedge\ 0 \le k \le |\varepsilon|\ \wedge\ D \in \mathbf{B}\ \wedge\ W.\varepsilon[k].D = E\ \wedge \\ \quad \langle \forall\ i\ :\ k \le i \le |\varepsilon|\ :\ E \in \mathrm{dom}(W.\varepsilon[i])\ \rangle \\ :\ \langle \forall\ i\ :\ k \le i \le |\varepsilon|\ :\ W.\varepsilon[i].D = E\ \rangle \\ \rangle$$

Proof

Assume $\varepsilon \in Z$, $0 \le k \le |\varepsilon|$, $D \in \mathbf{B}$, $W.\varepsilon[k].D = E$ and

$$\langle \forall\ i\ :\ k \le i \le |\varepsilon|\ :\ E \in \mathrm{dom}(W.\varepsilon[i])\ \rangle$$

We have, for $k \le i < |\varepsilon|$,

$$
\begin{array}{ll}
 & W.\varepsilon[i+1].D = E \\
\Leftarrow & \quad \{ \text{ Lemma B.12 } \} \\
 & W.\varepsilon[i].D = E \;\wedge\; E \in \mathrm{dom}(W.\varepsilon[i]) \\
\equiv & \quad \{ \; E \in \mathrm{dom}(W.\varepsilon[i]) \text{ by assumption } \} \\
 & W.\varepsilon[i].D = E
\end{array}
$$

Thus we have $W.\varepsilon[k].D = E$, and $W.\varepsilon[i].D = E \Rightarrow W.\varepsilon[i+1].D = E$, for $k \le i < |\varepsilon|$, so the result follows by induction.

(End of proof)

Lemma B.15

$$
\begin{array}{l}
\langle \forall\ \varepsilon, D, k, n \\
\quad : \varepsilon \in Z \;\wedge\; D \in \mathbf{B} \;\wedge\; 0 \le k \le |\varepsilon| \;\wedge \\
\quad\quad 0 \le n \;\wedge\; (W.\varepsilon[k])^n.D \textit{ is in a knot in } \varepsilon[k] \\
\quad : \langle \forall\, i \;:\; k \le i < |\varepsilon| \;:\; (W.\varepsilon[i])^n.D \textit{ is in a knot in } \varepsilon[i] \;\rangle \\
\quad \rangle
\end{array}
$$

Proof

Assume $\varepsilon \in Z$, and $D \in \mathbf{B}$. We prove the result by induction on n.

Basis: $n = 0$

For $0 \le k \le |\varepsilon|$, we have

$$
\begin{array}{ll}
 & (W.\varepsilon[i])^0.D \text{ is in a knot in } \varepsilon[k] \\
\equiv & \quad \{ \text{ definition } \} \\
 & D \text{ is in a knot in } \varepsilon[k] \\
\Rightarrow & \quad \{ \text{ Lemma B.13 } \} \\
 & \langle \forall\, i \;:\; k \le i \le |\varepsilon| \;:\; D \text{ is in a knot in } \varepsilon[i] \;\rangle
\end{array}
$$

$\equiv$ { definition }

$\langle \forall\, i \ : \ k \le i \le |\varepsilon| \ : \ (W.\varepsilon[i])^0.D$ is in a knot in $\varepsilon[i]\ \rangle$

Induction step

Assume as the induction hypothesis, for $0 \le n$,

$$
\begin{aligned}
&\langle \forall\ D, k \\
&\ : D \in \mathbf{B} \ \wedge\ 0 \le k \le |\varepsilon| \ \wedge\ (W.\varepsilon[k])^n.D \text{ is in a knot in } \varepsilon[k] \\
&\ : \langle \forall\, i \ : \ k \le i \le |\varepsilon| \ : \ (W.\varepsilon[i])^n.D \text{ is in a knot in } \varepsilon[i]\ \rangle \\
&\ \rangle
\end{aligned}
$$

Suppose $W.\varepsilon[k].D = E$, for $0 \le k \le |\varepsilon|$. We have

$(W.\varepsilon[k])^{n+1}.D$ is in a knot in $\varepsilon[k]$

$\equiv$ { definition }

$(W.\varepsilon[k])^n.E$ is in a knot in $\varepsilon[k]$

$\Rightarrow$ { induction hypothesis }

$\langle \forall\, i \ : \ k \le i \le |\varepsilon| \ : \ (W.\varepsilon[i])^n.E$ is in a knot in $\varepsilon[i]\ \rangle$

$\equiv$ { definition of domain }

$$
\begin{aligned}
&\langle \forall\ i \\
&\ : k \le i \le |\varepsilon| \\
&\ : (W.\varepsilon[i])^n.E \text{ is in a knot in } \varepsilon[i] \ \wedge\ E \in \mathrm{dom}(W.\varepsilon[i]) \\
&\ \rangle
\end{aligned}
$$

$\equiv$ { $\forall$ over $\wedge$ }

$\langle \forall\, i \ : \ k \le i \le |\varepsilon| \ : \ (W.\varepsilon[i])^n.E$ is in a knot in $\varepsilon[i]\ \rangle\ \wedge$

$\langle \forall\, i \ : \ k \le i \le |\varepsilon| \ : \ E \in \mathrm{dom}(W.\varepsilon[i])\ \rangle$

$\Rightarrow$ { Lemma B.14, $W.\varepsilon[k].D = E$ }

$\langle \forall\, i \ : \ k \le i \le |\varepsilon| \ : \ (W.\varepsilon[i])^n.E$ is in a knot in $\varepsilon[i]\ \rangle\ \wedge$

$\langle \forall\, i \ : \ k \le i \le |\varepsilon| \ : \ W.\varepsilon[i].D = E\ \rangle$

$\Rightarrow$ { $\forall$ over $\wedge$ }

$\langle\forall\ i$

$:\ k \leq i \leq |\varepsilon|$

$:\ W.\varepsilon[i].D = E\ \ \wedge\ \ (W.\varepsilon[i])^n.E$ is in a knot in $\varepsilon[i]$

$\rangle$

$\Rightarrow$ { definition }

$\langle\forall\ i\ :\ k \leq i \leq |\varepsilon|\ :\ (W.\varepsilon[i])^{n+1}.D$ is in a knot in $\varepsilon[i]\ \rangle$

(End of proof)

For the proof of Theorem 4.12, we show

$$\langle\forall\ \varepsilon, k, D$$
$$:\ \varepsilon \in Z\ \ \wedge\ \ 0 \leq k \leq |\varepsilon|\ \ \wedge\ \ D \in \mathbf{B}\ \ \wedge\ \ dl(\varepsilon[k], D)$$
$$:\ \langle\forall\ i\ :\ k \leq i \leq |\varepsilon|\ :\ dl(\varepsilon[i], D)\ \rangle$$
$$\rangle$$

Assume $\varepsilon \in Z$, $0 \leq k \leq |\varepsilon|$, and $D \in \mathbf{B}$. We have

$dl(\varepsilon[k], D)$

$\equiv$ { Definition 4.11 }

$\langle\exists\ n\ :\ 0 \leq n\ :\ (W.\varepsilon[k])^n.D$ is in a knot in $\varepsilon[k]\ \rangle$

$\Rightarrow$ { Lemma B.15 }

$\langle\exists\ n$

$:\ 0 \leq n$

$:\ \langle\forall\ i\ :\ k \leq i \leq |\varepsilon|\ :\ (W.\varepsilon[k])^n.D$ is in a knot in $\varepsilon[i]\ \rangle$

$\rangle$

$\Rightarrow$ { interchange quantifications }

$\langle\forall\ i$

$:\ k \le i \le |\varepsilon|$

$:\ \langle\exists\ n\ :\ 0 \le n\ :\ (W.\varepsilon[k])^n.D$ is in a knot in $\varepsilon[i]\ \rangle$

$\rangle$

$\equiv$ { Definition 4.11 }

$\langle\forall\ i\ :\ k \le i \le |\varepsilon|\ :\ dl(\varepsilon[i], D)\ \rangle$

B.2.4 Proof of Theorem 4.14

We use the following lemmas.

Lemma B.16

$$\langle\forall\ \mathbf{C}, D$$
$$:\ \mathbf{C} \in PC\ \wedge\ D \in \mathbf{B}\ \wedge\ dl(\mathbf{C}, D)$$
$$:\ D \in \mathrm{dom}(W.\mathbf{C}.D)\ \wedge\ dl(\mathbf{C}, W.\mathbf{C}.D)$$
$$\rangle$$

Proof

Immediate from Definitions 3.27 and 4.11.

(End of proof)

Lemma B.17

$$\langle\forall\ \mathbf{C}, D$$
$$:\ \mathbf{C} \in PC\ \wedge\ D \in \mathbf{B}$$
$$:\ dl(\mathbf{C}, D)\ \equiv\ \langle\forall\, n\ :\ 0 \le n\ :\ (W.\mathbf{C})^n.D\ \textit{is defined}\ \rangle$$
$$\rangle$$

Proof

Assume $\mathbf{C} \in PC$, and $D \in \mathbf{B}$. We prove the equivalence by proving the implication in both directions.

Case ⇒:

We have

$$
\begin{array}{ll}
 & \mathit{true} \\
\equiv & \quad \{ \text{ Lemma B.16 } \} \\
 & dl(\mathbf{C}, D) \;\Rightarrow\; D \in \mathrm{dom}(W.\mathbf{C}) \;\wedge\; dl(\mathbf{C}, W.\mathbf{C}.D) \\
\equiv & \quad \{ \text{ definition } \} \\
 & dl(\mathbf{C}, D) \;\Rightarrow\; W.\mathbf{C}.D \text{ is defined } \wedge\; dl(\mathbf{C}, W.\mathbf{C}.D)
\end{array}
$$

Now we have, for any deadlocked D,

$$
\begin{array}{ll}
 & \mathit{true} \\
\equiv & \quad \{ \text{ above } \} \\
 & \langle \forall\; n \\
 & \quad : 0 \leq n \;\wedge\; dl(\mathbf{C}, (W.\mathbf{C})^n.D) \\
 & \quad : (W.\mathbf{C})^{n+1}.D \text{ is defined } \;\wedge\; dl(\mathbf{C}, (W.\mathbf{C})^{n+1}.D) \\
 & \quad \rangle \\
\Rightarrow & \quad \{ \forall \text{ antimonotonic in the range } \} \\
 & \langle \forall\; n \\
 & \quad : 0 \leq n \;\wedge\; (W.\mathbf{C})^n.D \text{ is defined } \;\wedge\; dl(\mathbf{C}, (W.\mathbf{C})^n.D) \\
 & \quad : (W.\mathbf{C})^{n+1}.D \text{ is defined } \;\wedge\; dl(\mathbf{C}, (W.\mathbf{C})^{n+1}.D) \\
 & \quad \rangle \\
\equiv & \quad \{ \text{ Definition 3.17, assumption } \} \\
 & (W.\mathbf{C})^0.D \text{ is defined } \;\wedge\; dl(\mathbf{C}, (W.\mathbf{C})^0.D) \;\wedge
\end{array}
$$

$$\langle \forall\ n$$
$$:\ 0 \leq n\ \wedge\ (W.\mathbf{C})^n.D \text{ is defined } \wedge\ dl(\mathbf{C}, (W.\mathbf{C})^n.D)$$
$$:\ (W.\mathbf{C})^{n+1}.D \text{ is defined } \wedge\ dl(\mathbf{C}, (W.\mathbf{C})^{n+1}.D)$$
$$\rangle$$

$\Rightarrow$ { induction }

$$\langle \forall\, n\ :\ 0 \leq n\ :\ (W.\mathbf{C})^n.D \text{ is defined } \wedge\ dl(\mathbf{C}, (W.\mathbf{C})^n.D)\ \rangle$$

$\Rightarrow$ { weaken term }

$$\langle \forall\, n\ :\ 0 \leq n\ :\ (W.\mathbf{C})^n.D \text{ is defined }\rangle$$

Case $\Leftarrow$:

Assume

$$\langle \forall\, n\ :\ 0 \leq n\ :\ (W.\mathbf{C})^n.D \text{ is defined }\rangle$$

Consider the infinite sequence $U \in \mathbf{B}^+$,

$$U\ =\ \langle n\ :\ 0 \leq n\ :\ (W.\mathbf{C})^n.D\ \rangle$$

Every element in this sequence is defined, by assumption. Since the sequence is infinite, and $\mathbf{B}$ is finite, the sequence contains repeated values. We choose s and t such that $s < t$, and $U[s] = U[t]$. Then the sequence $U[s \ldots t-1]$ is a knot in $\mathbf{C}$, and $(W.\mathbf{C})^s.D \in U[s \ldots t-1]$. Thus $dl(\mathbf{C}, D)$.

(End of proof)

Lemma B.18

$$\langle \forall\, \mathbf{C}, D\ :\ \mathbf{C} \in PC\ \wedge\ D \in \mathbf{B}\ :\ \neg(D \in \text{rng}(W.\mathbf{C})\ \wedge\ qt(\mathbf{C}.D))\ \rangle$$

Proof

Assume $\mathbf{C} \in PC$, and $D \in \mathbf{B}$. We show

$$D \in \text{rng}(W.\mathbf{C}) \quad \Rightarrow \quad \neg qt(\mathbf{C}.D) \tag{B.1}$$

which is equivalent to the term above.

$$\begin{array}{ll}
 & D \in \text{rng}(W.\mathbf{C}) \\
\equiv & \quad \{ \text{ definition } \} \\
 & \langle \exists\, E \;\; :: \;\; W.\mathbf{C}.E = D \;\rangle \\
\equiv & \quad \{ \text{ Definition 3.27 } \} \\
 & \langle \exists\, E \;\; :: \;\; \mathbf{C}.E.\phi \in \{\textsc{pwait}, \textsc{wait}\} \;\wedge\; Agent(\mathbf{C}.E.\theta) = D \;\rangle \\
\equiv & \quad \{ \text{ Definition 3.20 } \} \\
 & \langle \exists\, E, m \;\; :: \;\; (m, E) \in \mathbf{C}.D.\gamma \;\rangle \\
\Rightarrow & \quad \{ \text{ list property } \} \\
 & \mathbf{C}.D.\gamma \neq \bot \\
\equiv & \quad \{ \text{ Definition 3.3 } \} \\
 & \neg qt(\mathbf{C}.D)
\end{array}$$

(End of proof)

For the proof of Theorem 4.14, we show

$$\begin{array}{l}
\langle \forall\; \mathbf{C} \\
\quad : \mathbf{C} \in PC \\
\quad : \langle \forall\, D \;\; :: \;\; Enabled(\mathbf{C}, D) = \bot \;\rangle \;\equiv\; \langle \forall\, D \;\; :: \;\; qt(\mathbf{C}.D) \;\vee\; dl(\mathbf{C}, D) \;\rangle \\
\quad \rangle
\end{array}$$

Assume $\mathbf{C} \in PC$. We prove the equivalence as two implications.

Case $\Rightarrow$:

We have

$$Enabled(\mathbf{C}, D) = \bot$$

$\Rightarrow$ { Theorem 3.50, Definition 3.48 }

$$qt(\mathbf{C}.D) \ \vee \ \mathbf{C}.D.\phi \in \{\text{PWAIT}, \text{WAIT}\}$$

$\Rightarrow$ { Definition 3.27 }

$$qt(\mathbf{C}.D) \ \vee \ D \in \text{dom}(W.\mathbf{C})$$

$\equiv$ { predicate calculus }

$$\neg qt(\mathbf{C}.D) \ \Rightarrow \ D \in \text{dom}(W.\mathbf{C})$$

$\Rightarrow$ { Lemma B.18 }

$$D \in \text{rng}(W.\mathbf{C}) \ \Rightarrow \ D \in \text{dom}(W.\mathbf{C})$$

From the first and penultimate lines, we get

$$Enabled(\mathbf{C}, D) = \bot \wedge \neg qt(\mathbf{C}.D) \quad \Rightarrow \quad D \in \text{dom}(W.\mathbf{C})$$

and from the first and last lines, we get

$$Enabled(\mathbf{C}, D) = \bot \quad \Rightarrow \quad (E \in \text{rng}(W.\mathbf{C}) \ \Rightarrow \ E \in \text{dom}(W.\mathbf{C}))$$

We now show, for any D

$$\langle \forall\, E \ : \ E \in \mathbf{B} \ : \ Enabled(\mathbf{C}, E) = \bot \ \rangle$$
$$\Rightarrow \ qt(\mathbf{C}.D.) \ \vee \ dl(\mathbf{C}, D)$$

by showing the following, which is equivalent.

$$\neg qt(\mathbf{C}.D.) \wedge \langle \forall\, E \ : \ E \in \mathbf{B} \ : \ Enabled(\mathbf{C}, E) = \bot \ \rangle$$
$$\Rightarrow \ dl(\mathbf{C}, D)$$

We have

$$\neg qt(\mathbf{C}.D.) \ \wedge \ \langle \forall\, E \ :: \ Enabled(\mathbf{C}, E) = \bot \ \rangle$$

$\equiv$ { instantiate }

$\neg qt(\mathbf{C}.D.) \;\wedge\; \mathit{Enabled}(\mathbf{C}, D) = \bot \;\wedge$

$\langle \forall\, E \;::\; \mathit{Enabled}(\mathbf{C}, E) = \bot \;\rangle$

$\Rightarrow$ { above results }

$D \in \mathrm{dom}(W.\mathbf{C}) \;\wedge\; \langle \forall\, E \;:\; E \in \mathrm{rng}(W.\mathbf{C}) \;:\; E \in \mathrm{dom}(W.\mathbf{C}) \;\rangle$

$\Rightarrow$ { Definition 3.17 }

$\langle \forall\, n \;:\; 0 \leq n \;:\; (W.\mathbf{C})^n.D \text{ is defined } \rangle$

$\equiv$ { Lemma B.17 }

$dl(\mathbf{C}, D)$

Case $\Leftarrow$:

We show the following, stronger, result.

$$\langle \forall\, D \;::\; qt(\mathbf{C}.D) \vee dl(\mathbf{C}, D) \;\Rightarrow\; \mathit{Enabled}(\mathbf{C}, D) = \bot \;\rangle$$

Assume $D \in \mathbf{B}$. We have

$qt(\mathbf{C}.D) \;\vee\; dl(\mathbf{C}, D)$

$\equiv$ { Lemma B.16 }

$qt(\mathbf{C}.D) \;\vee\; (dl(\mathbf{C}, D) \;\wedge\; dl(\mathbf{C}, W.\mathbf{C}.D))$

$\Rightarrow$ { definition }

$qt(\mathbf{C}.D) \;\vee$

$(\mathbf{C}.D.\phi \in \{\textsc{pwait}, \textsc{wait}\} \;\wedge\; \mathbf{C}.(W.\mathbf{C}.D).\phi \in \{\textsc{pwait}, \textsc{wait}\})$

$\Rightarrow$ { Theorem 3.50, Definition 3.48 }

$\mathit{Enabled}(\mathbf{C}, D) = \bot$

B.3 Proofs for Chapter 5

B.3.1 Proof of Theorem 5.56

We show

$\langle \forall\ \varepsilon, \varepsilon', \mathbf{T}$

$\quad : \varepsilon, \varepsilon' \in Z_m \ \wedge\ \varepsilon\ \mathit{resp}\ \mathbf{T}\ \wedge$

$\quad\ \ \varepsilon \xrightarrow{*} \varepsilon'$ by Theorem 5.46, 5.47, 5.48, 5.49, 5.50, or 5.55

$\quad : \varepsilon'\ \mathit{resp}\ \mathbf{T}$

$\quad \rangle$

Assume $\varepsilon, \varepsilon' \in Z_m$, $\varepsilon\ \mathit{resp}\ \mathbf{T}$, and $\varepsilon \xrightarrow{*} \varepsilon'$ by Theorem 5.46, 5.47, 5.48, 5.49, 5.50, or 5.55. A thread is active for an action from the first step taken for the thread, which is one of

accept
reject
action-start
p-action-start-rdv
t-action-start-rdv

to the last step taken for the thread, which is one of

accept
reject
action-end
action-reject

Note that **accept** and **reject** steps are both the first and last for their threads. Two threads are active at the same time if one has its first step between the first and last step for the other. A transformation that gives $\neg(\varepsilon' \ \mathit{resp}\ \mathbf{T})$ involves moving a start step left over an end step. Note that the start steps are all right-movers or decision steps, and that the end steps are all left-movers or decision steps. Thus this step involves moving a right-mover or decision step left over a left-mover or decision step.

Theorem 5.46 removes a thread from the execution, so does not introduce a step that violates $\mathbf{T}$. Theorems 5.47, and 5.48 do not change the relative order of the first and last steps for any threads. If Theorems 5.49, 5.50, or 5.55 are used to show $\varepsilon \xrightarrow{*} \varepsilon'$, then either a right-mover is moved right, or a left-mover is moved left. None of the theorems allows a right-mover or a decision step to move left over a left-mover or a decision step. Thus $\varepsilon' \ \mathit{resp}\ \mathbf{T}$.

Bibliography

[1] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] R J R Back. A method for refining atomicity in parallel algorithms. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *PARLE '89: Parallel Architecture and Languages Europe, vol II: Parallel Languages*, number 366 in Lecture Notes in Computer Science, pages 199–216. Springer-Verlag, 1989.

[3] R J R Back and J von Wright. Reasoning algebraically about loops. *Acta Informatica*, 36:295–334, 1999.

[4] Rohit Chandra, Anoop Gupta, and John L Hennessy. Cool: an object-based language for parallel computing. *Computer*, 27(8):13–26, Aug 1994.

[5] K Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[6] Ernie Cohen. A guide to reduction. Unpublished manuscript, 1994.

[7] Ernie Cohen and Leslie Lamport. Reduction in TLA. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR '98: Concurrency Theory*, number 1466 in Lecture Notes in Computer Science, pages 317–331. Springer-Verlag, 1998.

[8] B A Davey and H A Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[9] Edsger W Dijkstra and Carel S Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[10] Thomas W Doeppner, Jr. Parallel program correctness through refinement. In *Fourth ACM Symposium on the Principles of Programming Languages*, pages 155–169, 1977.

[11] E Allen Emerson. Temporal and modal logic. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1071. Elsevier Science Publishers, 1990.

[12] 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, Nov 1976.

[13] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[14] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Series in Data Management Systems. Morgan Kaufmann, 1992.

[15] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[16] C A R Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct 1969.

[17] C A R Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–282, 1972.

[18] C A R Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, Oct 1974.

[19] Steve J Hodges and Cliff B Jones. Non-interference properties of a concurrent object-based language: proofs based on an operational semantics. In Burkhard Freitag, Cliff B Jones, Christian Lengauer, and Hans-Jörg Schek, editors, *Object Orientation with Parallelism and Persistence*, pages 1–22. Kluwer Academic Publishers, 1996.

[20] Rajeev Joshi and Jayadev Misra. On the impossibility of robust solutions for fair resource allocation. Technical Report Technical Report TR-99-14, University of Texas at Austin, Department of Computer Sciences, Apr 1999.

[21] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[22] Leslie Lamport. A theorem on atomicity in distributed algorithms. *Distributed Computing*, 4:59–68, 1990.

[23] Leslie Lamport and Fred B Schneider. Pretending atomicity. Technical Report Research Report 44, Digital Equipment Corporation, Systems Research Center, 1989.

[24] Richard J Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–724, Dec 1975.

[25] Jayadev Misra. *A Discipline of Multiprogramming.* Texts and Monographs in Computer Science. Springer-Verlag, 2000 (in preparation). Parts of the manuscript are available from `ftp://ftp.cs.utexas.edu/pub/psp/seuss/-discipline.ps.gz`.

[26] C Mohan, Donald Fussell, and Abraham Silberschatz. Compatibility and commutativity of lock modes. *Information and Control*, 61(1):38–64, April 1984.

[27] J Eliot B Moss. *Nested Transactions: an Approach to Reliable Distributed Computing.* The MIT Press, 1985.

[28] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6:319–340, 1976.

[29] Gordon D Plotkin. The structural approach to operational semantics. Technical Report Research Paper DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, Sep 1981.

[30] A M Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, 1936. Correction in [31].

[31] A M Turing. On computable numbers, with an application to the Entscheidungsproblem. a correction. *Proceedings of the London Mathematical Society, Series 2*, 43:544–546, 1937. Correction to [30].

[32] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug 1990.

[33] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* MIT Press, 1990.

Vita

William Edward Adams was born in Bristol, England on December 5, 1960, to John Edward Adams and Sheila Margaret Adams. He attended St Bede's Roman Catholic Comprehensive School in Bristol until 1979, when he entered Trinity College, Cambridge. In 1982, he received a Bachelor of Arts degree in Mathematics from Cambridge University. After leaving Trinity College he lived in London, working at various times as a pedal cycle mechanic, a photoprocess operator, and a computer programmer. He attended graduate classes at the City University, London, from 1987 to 1989. He also ran a branch of the Labour Party, organizing several election campaigns, and helped lead a campaign of public-sector tenants that successfully opposed privatization of public housing. In 1991, he entered the Graduate School of the University of Texas at Austin. He received a Masters of Science in Computer Sciences in 1993.

Permanent Address: 4605 Richmond Avenue
Austin TX 78745
USA

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.