Bilateral Proofs of Safety and Progress Properties of Concurrent Programs

Jayadev Misra

University of Texas at Austin

Abstract

This paper suggests a theory of composable specification of concurrent programs that permits: (1) verification of program code for a given specification, and (2) composition of the specifications of the components to yield the specification of a program. The specification consists of both terminal properties that hold at the end of a program execution (if the execution terminates) and perpetual properties that hold throughout an execution. We devise (1) proof techniques for verification, and (2) composition rules to derive the specification of a program from those of its components. We employ terminal properties of components to derive perpetual properties of a program and conversely. Hence, this proof strategy is called bilateral. The compositional aspect of the theory is important in assembling a program out of components some of whose source code may not be available, as is increasingly the case with cross-vendor program integration.

Keywords: Program specification, concurrent programming, verification, composition, safety and progress properties.

1. Introduction

Four decades of intensive research has failed to yield a scalable solution to the problem of concurrent program design and verification. While there have been vast improvements in our understanding, the theory and practice in this area lag considerably behind what has been achieved for sequential programs. Very small programs, say for synchronization, are proved manually, though the proof methods are mostly unscalable. Larger programs of practical significance, say cache coherence protocols, are typically proved using model checking, which imposes size limitations. Programs from different vendors are rarely assembled to run concurrently.

We believe that the problem stems from the lack of a theory of *composable specification* for concurrent programs. Sequential imperative programs enjoy such a theory, introduced by Hoare [7], in which a program is specified by a pair

Email address: `misra@utexas.edu` (Jayadev Misra) of predicates, called its pre- and postcondition. The theory successfully permits: (1) verification of program code for a given specification, and (2) composition of the specifications of the components to yield the specification of a program. A fundamental concept is *invariant* that holds at specific program points, though invariant is not typically part of the program specification. Termination of a program is proved separately.

A specification of a concurrent program typically includes not just the pre- and postconditions but properties that hold of an entire execution, similar to invariants. A typical specification of a thread that requests a resource, for example, may state that: (1) once it requests a resource the thread waits until the resource is granted, and (2) once the resource is granted the thread will eventually release it. The first property is an instance of a *safety* property and the second of a *progress* property, see Lamport [10] and Owicki and Lamport [14].

Call the postcondition of a program for a given precondition to be a *terminal* property. And a property that holds throughout an execution a *perpetual* property. Terminal properties compose only for sequential programs, though not for concurrent programs, and they constitute the essence of the assertional proof method of Hoare. Safety and progress are typical perpetual properties.

This paper suggests a theory of composable specification of concurrent programs with similar goals as for sequential programs. The specification consists of both terminal and perpetual properties. We devise (1) proof techniques to verify that program code meets a given specification, and (2) composition rules to derive the specification of a program from those of its components. We employ terminal properties of components to derive perpetual properties of a program and conversely. Hence, this proof strategy is called *bilateral*. The compositional aspect of the theory is important in assembling a program out of components some of whose source code may not be available, as is increasingly the case with cross-vendor program integration.

The Hoare-proof theory for sequential programs is known to be sound and relatively-complete. A sound and relatively-complete proof theory for concurrent programs that use a very limited set of program constructs, known as *Unity*, appears in Chandy and Misra [2] and Misra [11]. This paper combines both approaches to yield a theory applicable to concurrent programs written in conventional languages. (The soundness and relatively-completeness of this theory has not been proved yet.)

We treat three examples of varying complexity in detail in this paper. First, a program that implements a distributed counter is used to illustrate the various concepts of the theory in stages through out the paper. Appendix Appendix B includes a proof of a mutual exclusion program, a system of tightly-coupled components. Unlike traditional proofs, it is based on composing the specifications of the components. Appendix Appendix C includes proof of a recursively-defined concurrent program, where code of one component is only implicit, again using composable specifications. We know of no other proof technique that can be used to prove this example program.

2. Program and Execution Model

2.1. Program Structure

The syntax for programs and its components is given below.

$$\begin{array}{rlll} & \textit{action} & ::= \text{guard} & \rightarrow \quad \text{body} \\ f,\ g :: & \textit{component} & ::= \textit{action} \mid f \ [] \ g & \mid \text{seq}(f_0,\ f_1, \cdots f_n) \\ & \textit{program} & ::= f & \end{array}$$

An action has a guard, which is a predicate, and a body which is a piece of code. Execution of the body in a state in which the guard holds is guaranteed to terminate; we assume that this guarantee is independently available. Details of execution semantics is given Section 2.2. A guard that is *true* is often dropped. An action without a guard is *non-blocking* and a guarded action *blocking*.

A structured component is either: (1) a *join* of the form $f \ [] \ g$ where f and g are its *direct subcomponents*, or (2) $\text{seq}(f_0,\ f_1, \cdots f_n)$ where the direct subcomponents, f_i, are combined using some sequential language construct. A join models concurrent execution, as we describe in Section 2.2. And seq denotes any sequential language construct for which proof rules are available. Thus, the typical constructs of sequencing, if-then-else and do-while are seq constructs. A subcomponent of any component is either its direct subcomponent or a subcomponent of some direct subcomponent. Note that a component is never empty.

Join construct is commutative and associative. The syntax permits arbitrary nesting of sequential and concurrency constructs, so, "$(f \ [] \ g)$ **;** $(f' \ [] \ g')$" is a program with $f,\ f',\ g$ and g' as components.

A *program* is a component that is meant to be executed alone.

Access rights to variables. A variable named in a component is *accessible* to it. Variable x is *local* to f if f has exclusive write-access to x during any of its executions. Therefore, any accessible variable of a component in a sequential program is local to it. However, a local variable of $f \ [] \ g$ may not be local to either f or g. An accessible non-local variable x of f is *shared*; x is shared with g if x is also accessible to g. Note that x is local to f and shared in g if f has exclusive write-access to x and g can only read x.

A *program* is executed alone, so, all its accessible variables are local to it.

A *local predicate* of f is a predicate in which every variable is a local variable of f. Therefore, *true* and *false* are local to all components. It follows that the value of a local predicate of f remains unchanged in any execution as long as f does not take a step.

2.2. Program execution

A *step* is an instance of the execution of an action. A step of action $b \rightarrow \alpha$ is executed as follows: evaluate b without preemption and if it is true, immediately execute α to completion, again without preemption —this is called an *effective execution* of the action— else the program state (and its control point) are unaltered —this is called an *ineffective execution.* Thus, in an effective execution of $b \rightarrow \alpha$, b holds when the execution of α begins. Ineffective execution models busy-waiting whereby the action execution is retried at some future moment.

Execution of a join starts simultaneously for all its direct subcomponents, and terminates when they all do. At any moment during the execution, the program control may reside simultaneously in many of its subcomponents. Execution rules for a seq are the traditional ones from sequential programming, which ensures that the program control is within one direct subcomponent at any moment. The subcomponent itself may be a join which may have multiple control points of its own.

Initially the program control is at the entry point of the program. In any given state the program takes a step by choosing a control point before an action and executing, effectively or ineffectively, the action. If there is no such control point, the program has terminated. The choice of control point for a step is arbitrary, but subject to the following fairness constraint: every control point is chosen eventually for execution, so that no component is ignored forever. This is the same constraint advocated by Dijkstra [5] in his classic paper. In contrast to a terminated execution a deadlocked execution attempts executions of certain actions ineffectively forever, and the control resides permanently at the points preceding each of these actions.

An execution is a sequence of steps that can not be extended. Infinite executions do not have an end state. A finite execution either terminates or is deadlocked, and it has an end state. It simplifies the proof theory considerably to imagine that every execution is infinite, by extending each finite execution by an infinite number of stutter steps that repeat the end state forever.

2.3. Example: Distributed counter

The following example is inspired by a protocol developed by Blumofe [1] in his Ph.D. thesis. We abstract one aspect of it that implements a distributed counter. The original proof of the protocol is due to Rajeev Joshi. The proof in this paper, which is based on Joshi's proof, closely follows the one given in Chapter 6 of Misra [11].

The protocol f that implements counter ctr is the join of a finite number of threads, f_j, given below. Each f_j has local variables old_j and new_j. Below, each assignment statement and guarded statement is an action. The following form of the **if** statement was introduced by Dijkstra [4]; in each execution of the statement an alternative whose guard holds is executed without preemption.

```
initially  $ctr = 0$
$f_j$ ::
  initially  $old_j,\ new_j = 0,\ 0$
  loop
      $new_j := old_j + 1$;
      if [  $ctr = old_j \ \rightarrow\ ctr := new_j$
          | $ctr \neq old_j \ \rightarrow\ old_j := ctr$ ]
  forever
```

It is required to show that *ctr* behaves as a counter, that is: (1) *ctr* never decreases and it increases only by 1, and (2) *ctr* increases eventually. Both of these are perpetual properties. There is no terminal property of interest since the program never terminates.

3. Introduction to the proof theory

3.1. Specification

A specification of component f is of the form $\{r\}\ f\ \{Q \ \mid\ s\}$ where r and s are the *pre- and postconditions*, and Q is a set of *perpetual* properties. The meaning of such a specification is as follows. For any execution of f starting in an r-state,

1. if the execution terminates, the end state is an s-state, and
2. every property in Q holds for the execution.

We give proof rules for pre- and postconditions in the following section. Proof rules for perpetual properties appear in subsequent sections, for safety properties in Section 4 and for progress properties in Section 5.

Terminology. Write $\{r\}\ f\ \{s\}$ when Q is irrelevant in the discussion, and $\{r\}\ f\ \{Q \ \mid\ \}$ when s is irrelevant. Further for $q \in Q$, write "q in f" when r is understood from the context, and just "q" when both f and r are understood. An inference rule without explicit mention of f or r denotes that the rule applies for any f and r.

Variables named in properties. A property of component f includes predicates that name accessible variables of f, and other variables, called *free* variables. A property is implicitly universally quantified over its free variables. Any inaccessible variable named in a property denotes a free variable, and, being a bound variable, may be renamed.

3.2. Local Annotation

Local Annotation of component f associates assertions with program points such that each assertion holds whenever program control reaches the associated point in every execution of *any program in which f is a component.* Thus, a local annotation yields precondition for the execution of each action of f, and valid pre- and postcondition of f in any environment. Since the execution environment of f is arbitrary, only the predicates that are local to f are unaffected by executions of other components. Therefore, a local annotation associates predicates local to *each point* of f, as explained below.

Local annotation is defined by the program structure. First, the proof rule for an action is as follows:

$$\frac{\{p \wedge b\}\ \alpha\ \{q\}}{\{p\}\ b \rightarrow \alpha\ \{q\}}$$

To construct a local annotation of $f = \text{seq}(f_0,\ f_1, \cdots f_n)$, construct local annotation of each f_i using only the local variables of f_i as well as those of f. Then construct an annotation of f using the proof rules for seq from the sequential program proof theory. Observe that the local variables of f are local to each f_i because in a sequential execution among the direct subcomponents of f each f_i has exclusive write-access to these variables during its execution.

To construct a local annotation of $f = g\ []\ h$, construct local annotations of each of g and h using only their local variables. Then construct an annotation of f using the proof rule given below. Note that the proof rule is valid only because the assertions in g and h are local to those components.

$$\frac{\begin{array}{c}\{r\}\ g\ \{s\}\\ \{r'\}\ h\ \{s'\}\end{array}}{\{r \wedge r'\}\ g\ []\ h\ \{s \wedge s'\}}$$

Observe that a local variable of f is not necessarily local to g or h unless they have exclusive write-access to it. Henceforth, all annotations in this paper are local annotations.

A shortcoming of local annotation is that a variable that is local to $f\ []\ g$ but shared by both f and g can not appear in a local annotation by the application of these rules alone. The invariance meta-rule, given in Section 3.4, overcomes this problem.

3.3. Example: Distributed Counter, contd.

Construct a local annotation of f_j for the example of Section 2.3. Below, we have labeled the actions to refer to them in subsequent proofs.

```
$f_j$ ::
  **initially** $old_j,\ new_j = 0,\ 0$
  $\{true\}$
  **loop**
    $\{true\}$
      $\alpha_j$ :: $new_j := old_j + 1$;
    $\{new_j = old_j + 1\}$
      **if** [ $\beta_j$ :: $\{new_j = old_j + 1\}$ $ctr = old_j \rightarrow ctr := new_j$  $\{true\}$
           | $\gamma_j$ :: $\{new_j = old_j + 1\}$ $ctr \neq old_j \rightarrow old_j := ctr$   $\{true\}$
         ]
    $\{true\}$
  **forever**
```

3.4. Meta-rules

The following general rules apply for specifications.

- (lhs strengthening; rhs weakening)

$$\frac{\begin{array}{c}\{r\}\ f\ \{Q\ \mid\ s\}\\ r' \Rightarrow r,\ \ s \Rightarrow s',\ \ Q' \subseteq Q,\ \ r' \text{ and } s' \text{ are local to } f\end{array}}{\{r'\}\ f\ \{Q'\ \mid\ s'\}}$$

- (Conjunction; Disjunction)

$$\frac{\{r\}\ f\ \{Q\ \mid\ s\} \qquad \{r'\}\ f\ \{Q'\ \mid\ s'\}}{\begin{array}{c}\{r \wedge r'\}\ f\ \{Q \cup Q'\ \mid\ s \wedge s'\} \\ \{r \vee r'\}\ f\ \{Q \cap Q'\ \mid\ s \vee s'\}\end{array}}$$

Justifications for the meta-rules. The lhs strengthening and rhs weakening rules are inspired by similar rules for Hoare-triples. Additionally, since the properties in Q are independent, any number of them may be removed.

For the conjunction rule, let the set of executions of f starting in r-state be r-executions, and, similarly r'-executions. The $r \wedge r'$-executions is the intersection of r-executions and r'-executions. Therefore, the postcondition of any execution in $r \wedge r'$-executions satisfies $s \wedge s'$ and every property in Q or Q', justifying the conjunction rule. The arguments for the disjunction rule are similar.

4. Safety Properties

A safety property is perpetual. We develop a safety property, **co**, and its special cases, taken from Misra [11]. Safety properties are derived from local annotations and/or safety properties of the subcomponents of a component. Conversely, safety properties permit establishing stronger annotations and terminal properties.

4.1. Safety Property **co**

Write p **co** q in component f, for predicates p and q that may not be local to f, to mean that *effective execution* of any action of f in a p-state establishes a q-state. Observe that an ineffective execution preserves p. Thus, given p **co** q: (1) in any execution of f once p holds it continues to hold until q is established, though q may never be established, and (2) as a composition rule, p **co** q holds in component iff it holds in every subcomponent of it.

For an annotated component, **co** is defined by the following proof rule.

$$\frac{\begin{array}{c}\{r\}\ f\ \{s\} \\ \text{For every action } b \rightarrow \alpha \text{ with precondition } pre \text{ in the annotation :} \\ \{pre \wedge b \wedge p\}\ \alpha\ \{q\}\end{array}}{\{r\}\ f\ \{p\ \mathbf{co}\ q\ \mid\ s\}}$$

As an example, the statement "every change in integer variable *ctr* can only increment its value" may be formalized as $ctr = m$ **co** $ctr = m\ \vee\ ctr = m + 1$, for all integer m.

4.2. Special cases of **co**

Define special cases of **co** for component f: **stable**, **constant** and **invariant**. Given predicate p and expression e, in any execution of f: (1) **stable** p means that p remains *true* once it becomes *true*, (2) **constant** e that the value of e never changes, and (3) **invariant** p that p is always *true*, including after termination, if the program terminates. Formally, in f

$$\begin{array}{lll}
\textbf{stable } p & \equiv & p \ \textbf{co} \ p \\
\textbf{constant } e & \equiv & (\forall c :: \ \textbf{stable } e = c) \\
\textbf{invariant } p & \equiv & \textbf{initially } p \text{ and } \textbf{stable } p
\end{array}$$

Observe that **invariant** *true* (hence, **stable** *true*) and **stable** *false* are properties of every component. A variable for which f has no write-access is constant in f, and so is any expression constructed out of such variables.

Derived rules for **co** and some of its special cases, which are heavily used in actual proofs, are given in Appendix Appendix A.1. It follows from the derived rules that a safety property of a program is a property of all its components, and conversely, as given by the inheritance rule below.

4.3. Meta-rules

1. (Inheritance) If any safety property (**co** or any of its special cases) holds in all subcomponents of f then it holds in f. More formally, for safety properties σ,

$$\text{Given: } \frac{(\forall i :: \ \{r_i\} \ f_i \ \{s_i\})}{\{r\} \ f \ \{s\}} \qquad \text{Assert: } \frac{(\forall i :: \ \{r_i\} \ f_i \ \{\sigma \ \mid \ s_i\})}{\{r\} \ f \ \{\sigma \ \mid \ s\}}$$

2. (Invariance) A local invariant of a component, i.e., a local predicate that is invariant in the component, can be substituted for *true*, and vice versa, in any predicate in an annotation or property of the component. In particular, for a program all variables are local, so any invariant can be conjoined to an assertion including the postcondition.

Justifications for the meta-rules. Inheritance rule is based on the fact that if a safety property holds for all components of f it holds for f as well. Given the proof rule at left the inheritance proof rule at right can be asserted for any set of safety properties σ.

The invariance rule is from Chandy and Misra [2] where it is called the "substitution axiom". One consequence of the rule is that a local invariant of $f \ [] \ g$, that may not be a local predicate of either f or g, could be conjoined to predicates in an annotation of $f \ [] \ g$. Additionally, all variables in a program are local; so, any invariant can be substituted for *true* in a program.

4.4. Example: Distributed Counter, contd.

For the example of Section 2.3 we prove that *ctr* behaves as a counter in that its value can only be incremented, i.e., $ctr = m$ **co** $ctr = m \vee ctr = m + 1$ in f. Using the inheritance rule, it is sufficient to prove this property in every component f_j. In f_j, only β_j may change the value of *ctr*; so we need only show the following whose proof is immediate:

$$\{ctr = m \wedge new_j = old_j + 1 \wedge ctr = old_j\} \ ctr := new_j \ \{ctr = m \vee ctr = m + 1\}$$

5. Progress Properties

We are mostly interested in progress properties of the form "if predicate p holds at any point during the execution of a component, predicate q holds eventually". Here "eventually" includes the current and all future moments in the execution. This property, called *leads-to*, is defined in Section 5.3 (page 12). First, we introduce two simpler progress properties, *transient* and *ensures*. Transient is a fundamental progress property, the counterpart of the safety property **stable**. It is introduced because its proof rules are easy to justify and it can be used to define ensures. However, it is rarely used in program specification because ensures is far more useful in practice. Ensures is used to define *leads-to*.

5.1. Progress Property: transient

In contrast to a stable predicate that remains true once it becomes true, a transient predicate is guaranteed to be falsified eventually. That is, predicate p is transient in component f implies that if p holds at any point during an execution of f, $\neg p$ holds then or eventually in that execution. In temporal logic notation p is transient is written as $\Box\Diamond\neg p$. Note that $\neg p$ may not continue to hold after p has been falsified. Predicate *false* is transient because $\mathit{false} \Rightarrow \mathit{true}$, and, hence $\neg\mathit{false}$ holds whenever *false* holds. Note that given p transient in f, $\neg p$ holds at the termination point of f because, otherwise, f can take no further steps to falsify p.

The proof rules are given in Figure 5.1. Below, post_f is a local predicate of f that is initially *false* and becomes *true* only on the termination of f. Such a predicate always exists, say, by encoding the termination control point into it. For a non-terminating program, post_f is *false*. Proof of post_f is a safety proof.

Justifications. The formal justification is based on induction on the program structure: show that the basis rule is justified and then inductively prove the remaining rules. We give informal justifications below.

In the basis rule the hypotheses guarantee that each action of f is effectively executed whenever p holds, and that the execution establishes $\neg p$. If no action can be executed effectively, because precondition of no action holds, the program has terminated and post_f holds. Hence, $\neg p \vee \mathit{post}_f$, i.e.$\neg(p \wedge \neg \mathit{post}_f)$, hold eventually in all cases. Note that if $\mathit{pre} \Rightarrow \neg p$ then $\mathit{pre} \wedge p$ is *false* and the hypotheses are vacuously satisfied. If f never terminates, $\neg\mathit{post}_f$ always holds and $\neg p$ is guaranteed eventually.

The next two rules, for sequential and concurrent composition, have weaker hypotheses. The sequencing rule is based on an observation about a sequence of actions, α; β. To prove **transient** p it is sufficient that α establish $\neg p$ *or that it execute effectively, thus establishing* post_α, and that β establish $\neg p$. The sequencing rule generalizes this observation to components. Being a local predicate, post_f can not be falsified by any concurrently executing component, so it holds as long as the control remains at the termination point of f.

In the concurrency rule, as a tautology g either establishes $\neg p$ eventually, thus guaranteeing the desired result, or preserves p forever. In the latter case, f establishes $\neg p$ since **transient** p holds in f.

- (Basis)

$$\frac{\begin{array}{c}\{r\}\ f\ \{s\}\\ \text{For every action } b \rightarrow \alpha \text{ of } f \text{ with precondition } pre:\\ pre \wedge p \Rightarrow b\\ \{pre \wedge p\}\ \alpha\ \{\neg p\}\end{array}}{\{r\}\ f\ \{\textbf{transient } p \wedge \neg post_f \ \mid\ s\}}$$

- (Sequencing)

$$\frac{\begin{array}{c}\{r\}\ f\ \{\textbf{transient } p \wedge \neg post_f \ \mid\ post_f\}\\ \{post_f\}\ g\ \{\textbf{transient } p \ \mid\ \}\end{array}}{\{r\}\ f\ \textbf{;}\ g\ \{\textbf{transient } p \ \mid\ \}}$$

- (Concurrency)

$$\frac{\{r\}\ f\ \{\textbf{transient } p \ \mid\ \}}{\{r\}\ f\ []\ g\ \{\textbf{transient } p \ \mid\ \}}$$

- (Inheritance)

$$\text{Given: } \frac{(\forall i ::\ \{r_i\}\ f_i\ \{s_i\})}{\{r\}\ f\ \{s\}} \qquad \text{Assert: } \frac{(\forall i ::\ \{r_i\}\ f_i\ \{\textbf{transient } p \ \mid\ s_i\})}{\{r\}\ f\ \{\textbf{transient } p \ \mid\ s\}}$$

Figure 1: Definition of **transient**

The inheritance rule applies to a program with multiple components. It asserts that if the property holds in each component f_i then it holds in program f. To see this consider two cases: f is seq or join, and argue by induction on the program structure.

For seq f: if p holds at some point before termination of f it is within exactly one direct subcomponent f_i, or will do so without changing any variable value. For example, if control precedes execution of "**if**$_b$ **then** f_0 **else** f_1" then it will move to a point preceding f_0 or f_1 after evaluation of b without changing the state. Note that f_i may be a join, so there may be multiple program points within it where control may reside simultaneously, but all controls reside within one direct subcomponent of seq f at any moment. From the hypothesis, that component, and hence, the program establishes $\neg p$ eventually.

For a join, say $f\ []\ g$: Consider an execution in which, say, f has not terminated when p holds. From the arguments for the concurrency rule, using that **transient** p in f, eventually $\neg p$ is established in that execution. Similar remarks apply for all executions in which g has not terminated. And, if both f and g have terminated, then $\neg p$ holds from the definition of **transient** for each component. □

Notes.

1. The basis rule by itself is sufficient to define an elementary form of transient. However, the transient predicate then has to be extremely elaborate, typically encoding control point of the program, so that it is falsified by

every action of the component. The other rules permit simpler predicates to be proven transient.

2. Basis rule is the only rule that needs program code for its application, others are derived from properties of the components, and hence, permit specification composition.
3. It is possible that p is eventually falsified in every execution of a component though there is no proof for **transient** p. To see this consider the program $f \ [] \ g$ in which every action of f falsifies p only if for some predicate q, $p \wedge q$ holds as a precondition, and every action of g falsifies p only if $p \wedge \neg q$ holds as a precondition, and neither component modifies q. Clearly, p will be falsified eventually, but this fact can not be proved as a transient property; only $p \wedge q$ and $p \wedge \neg q$ can be shown transient. As we show later, p *leads-to* $\neg p$.

5.2. Progress Property: ensures

Property ensures for component f, written as p **en** q with predicates p and q, says that if p holds at any moment in an execution of f then it continues to hold until q holds, and q holds eventually. This claim applies even if p holds after the termination of f. For initial state predicate r, it is written formally as $\{r\}\ f\ \{p\ \mathbf{en}\ q \ \mid \ \}$ and defined as follows:

$$\frac{\{r\}\ f\ \{p \wedge \neg q\ \ \mathbf{co}\ \ p \vee q,\ \mathbf{transient}\ p \wedge \neg q\ \mid\ \}}{\{r\}\ f\ \{p\ \mathbf{en}\ q\ \mid\ \}}$$

We see from the safety property in the hypothesis that once p holds it continues until q holds, and from the transient property that eventually q holds.

Corresponding to each proof rule for transient, there is a similar rule for ensures. These rules and additional derived rules for **en** are given in Appendix Appendix A.3 (page 16).

Example: Distributed counter, contd.. We prove a progress property of the annotated program from Section 3.3, reproduced below.

```
$f_j$ ::
  initially  $old_j,\ new_j = 0,\ 0$
  $\{true\}$
  loop
    $\{true\}$
      $\alpha_j$ :: $new_j := old_j + 1$;
    $\{new_j = old_j + 1\}$
      if [ $\beta_j$ :: $\{new_j = old_j + 1\}\ ctr = old_j \ \rightarrow\ ctr := new_j \ \ \{true\}$
         | $\gamma_j$ :: $\{new_j = old_j + 1\}\ ctr \neq old_j \ \rightarrow\ old_j := ctr \ \ \{true\}$
         ]
    $\{true\}$
  forever
```

Our ultimate goal is to prove that for any integer m if $ctr = m$ at some point during an execution of f, eventually $ctr > m$. To this end let auxiliary variable nb be the number of threads f_j for which $ctr \neq old_j$. We prove the following ensures property, (E), that says that every step of f either increases ctr or decreases nb while preserving ctr's value. Proof uses the inheritance rule from Appendix Appendix A.3 (page 16). For every f_j and any m and N:

$$ctr = m \wedge nb = N \ \mathbf{en} \ \ ctr = m \wedge nb < N \vee ctr > m \text{ in } f_j \tag{E}$$

We use the rules for **en** given in Appendix Appendix A.3 (page 16). First, to prove (E) in g; h, for any g and h, it is sufficient to show that g terminates and (E) in h. Hence, it is sufficient to show that (E) holds only for the loop in f_j, because initialization always terminates. Next, using the inheritance rule, it is sufficient to show that (E) holds only for the body of the loop in f_j. Further, since α_j always terminates, (E) needs to be shown only for the **if** statement. Using inheritance, prove (E) for β_j and γ_j. In each case, assume the precondition $ctr = m \wedge nb = N$ of **if** and the preconditions of β_j and γ_j. The postcondition $ctr = m \wedge nb < N \vee ctr > m$ is easy to see in each of the following cases:

$$\begin{aligned}
\beta_j ::\ & \{ctr = m \wedge nb = N \wedge new_j = old_j + 1 \wedge ctr = old_j\} \\
& \qquad ctr := new_j \\
& \{ctr = m \wedge nb < N \vee ctr > m\}
\end{aligned}$$

$$\begin{aligned}
\gamma_j ::\ & \{ctr = m \wedge nb = N \wedge new_j = old_j + 1 \wedge ctr \neq old_j\} \\
& \qquad old_j := ctr \\
& \{ctr = m \wedge nb < N \vee ctr > m\}
\end{aligned}$$

5.3. Progress Property: Leads-to

The informal meaning of $p \mapsto q$ (read: p *leads-to* q) is "if p holds at any point during an execution, q holds eventually". Unlike **en**, p is not required to hold until q holds.

Leads-to is defined by the following three rules, taken from Chandy and Misra [2]. The rules are easy to justify intuitively.

- (basis) $\dfrac{p \ \mathbf{en} \ q}{p \ \mapsto \ q}$
- (transitivity) $\dfrac{p \ \mapsto \ q \ , \ q \ \mapsto \ r}{p \ \mapsto \ r}$
- (disjunction) For any (finite or infinite) set of predicates S

$$\frac{(\forall p:\ p \in S:\ p \ \mapsto \ q)}{(\vee p:\ p \in S:\ p) \ \mapsto \ q}$$

Derived rules for $\mapsto$ are given in Appendix Appendix A.4 (page 18). *leads-to* is not conjunctive, nor does it obey the inheritance rule, so even if $p \mapsto q$ holds in both f and g it may not hold in f [] g.

Example: Distributed counter, contd.. We show that for the example of Section 2.3 the counter *ctr* increases without bound. The proof is actually quite simple. We use the induction rule for leads-to given in Appendix Section Appendix A.4.2.

The goal is to show that for any integer C, $\mathit{true} \mapsto \mathit{ctr} > C$. Below, all properties are in f.

$\mathit{ctr} = m \land \mathit{nb} = N$ **en** $\mathit{ctr} = m \land \mathit{nb} < N \lor \mathit{ctr} > m$
  proven in Section 5.2
$\mathit{ctr} = m \land \mathit{nb} = N \;\mapsto\; \mathit{ctr} = m \land \mathit{nb} < N \lor \mathit{ctr} > m$
  Applying the basis rule of *leads-to*
$\mathit{ctr} = m \;\mapsto\; \mathit{ctr} > m$
  Induction rule, use the well-founded order $<$ over natural numbers
$\mathit{true} \;\mapsto\; \mathit{ctr} > C$, for any integer C
  Induction rule, use the well-founded order $<$ over natural numbers.

6. Related Work

The earliest proof method for concurrent programs appears in Owicki and Gries [13]. The method works well for small programs, but does not scale up for large ones. Further it is limited to proving safety properties only. There is no notion of component specification and their composition. Lamport [10] first identified *leads-to* for concurrent programs as the appropriate generalization of termination for sequential programs ("progress" is called *liveness* in that paper). Owicki and Lamport [14] is a pioneering paper.

The first method to suggest proof rules in the style of Hoare [7], and thus a specification technique, is due to Jones [8, 9]. Each component is annotated assuming that its environment preserves certain predicates. Then the assumptions are discharged using the annotations of the various components. The method though is restricted to safety properties only. A similar technique for message communicating programs was proposed in Misra and Chandy [12].

The approaches above are all based on specifying allowed interface behaviors using compositional temporal logics. Since 2000, a number of proposals[6, 3] have instead used traditional Hoare triples and resource/object invariants, but extending state predicates to include permissions describing how locations can be used, and suitably generalizing the Hoare rule for disjoint parallel composition. However, these proposals typically address only global safety and local termination properties, not global progress properties (as this paper does).

A completely different approach is suggested in the UNITY theory of Chandy and Misra [2], and extended in Misra [11]. A restricted language for describing programs is prescribed. There is no notion of associating assertions with program points. Instead, the safety and progress specification of each component is given by a set of properties that are proved directly. The specifications of components of a program can be composed to derive program properties. The current paper extends this approach by removing the syntactic constraints on programs, though the safety and progress properties of UNITY are the ones used in this paper.

One of the essential questions in these proof methods is to propose the appropriate preconditions for actions. In Owicki and Gries [13] theory it is postulated and proved. In UNITY the programmer supplies the preconditions, which are often easily available for event-driven systems. Here, we derive preconditions that remain valid in any environment; so there can be no assertion about shared variables. The theory separates local precondition (obtained through annotation) from global properties that may mention shared variables.

The proof strategy described in the paper is bilateral in the following sense. An invariant, a perpetual property, may be used to strengthen a postcondition, a terminal property, using the invariance rule. Conversely, a terminal property, postcondition $post_f$ of f, may be employed to deduce a transient predicate, a perpetual property.

Separation logic [16] has been effective in reasonong about concurrently accessed data structures. We are studying its relationship to the work described here.

Acknowledgment. I am truly grateful to Ernie Cohen of Amazon and José Meseguer of the Univ. of Illinois at Urbana, for reading the manuscript thoroughly and several times, endless discussions and substantive comments that have shaped the paper. Presenting this work at the IFIPS Working Group 2.3 and the Programming Languages lunch at the University of Texas at Austin has sharpened my understanding; I am grateful to the students and the other attendees, in particular to Dhananjay Raju.

Appendix A. Appendix: Derived Rules

Appendix A.1. Derived Rules for **co** *and its special cases*

The derived rules for **co** are given in Figure A.2 and for the special cases in Figure A.3. The rules are easy to derive; see Chapter 5 of Misra [11].

$$\mathit{false} \ \ \mathbf{co} \ \ q \qquad\qquad p \ \ \mathbf{co} \ \ \mathit{true}$$

$$\frac{p \ \mathbf{co} \ q \ , \ p' \ \mathbf{co} \ q'}{p \wedge p' \ \mathbf{co} \ q \wedge q'} \ (\textsc{conjunction}) \qquad \frac{p \ \mathbf{co} \ q \ , \ p' \ \mathbf{co} \ q'}{p \vee p' \ \mathbf{co} \ q \vee q'} \ (\textsc{disjunction})$$

$$\frac{p \ \mathbf{co} \ q}{p \wedge p' \ \mathbf{co} \ q} \ (\textsc{lhs strengthening}) \qquad \frac{p \ \mathbf{co} \ q}{p \ \mathbf{co} \ q \vee q'} \ (\textsc{rhs weakening})$$

Figure A.2: Derived rules for **co**

The top two rules in Figure A.2 are simple properties of Hoare triples. The conjunction and disjunction rules follow from the conjunctivity, disjunctivity and monotonicity properties of the weakest precondition, see Dijkstra [4] and of logical implication. These rules generalize in the obvious manner to any set —finite or infinite— of **co**-properties, because weakest precondition and logical implication are universally conjunctive and disjunctive.

The following rules for the special cases are easy to derive from the definition of **stable** , **invariant** and **constant** . Special Cases of co

- (stable conjunction, stable disjunction)
$$\frac{p \ \mathbf{co} \ q \ , \ \mathbf{stable} \ r}{\begin{array}{c} p \wedge r \ \mathbf{co} \ q \wedge r \\ p \vee r \ \mathbf{co} \ q \vee r \end{array}}$$
- (Special case of the above) $\dfrac{\mathbf{stable} \ p \ , \ \mathbf{stable} \ q}{\begin{array}{c} \mathbf{stable} \ p \wedge q \\ \mathbf{stable} \ p \vee q \end{array}}$
- $$\frac{\mathbf{invariant} \ p \ , \ \mathbf{invariant} \ q}{\begin{array}{c} \mathbf{invariant} \ p \wedge q \\ \mathbf{invariant} \ p \vee q \end{array}}$$
- $$\frac{\{r\} \ f \ \{\mathbf{stable} \ p \ \mid \}}{\{r \wedge p\} \ f \ \{p\}} \qquad \frac{\{r\} \ f \ \{\mathbf{constant} \ e \ \mid \}}{\{r \wedge e = c\} \ f \ \{e = c\}}$$
- (constant formation) Any expression built out of constants is constant.

Figure A.3: Derived rules for the special cases of **co**

Appendix A.2. Derived Rules for **transient**

- **transient** *false*.
- (Strengthening) Given **transient** p, **transient** $p \wedge q$ for any q.

To prove **transient** *false* use the basis rule. The proof of the strengthening rule uses induction on the number of applications of the proof rules in deriving **transient** p. The proof is a template for proofs of many derived rules for ensures and leads-to. Consider the different ways by which **transient** p can be proved in a component. Basis rule gives the base case of induction.

1. (Basis) In component f, p is of the form $p' \wedge \neg post_f$ for some p'. Then in some annotation of f where action $b \rightarrow \alpha$ has the precondition pre:
(1) $pre \wedge p' \;\Rightarrow\; b$, and (2) $\{pre \wedge p'\}\ \alpha\ \{\neg p'\}$.
(1') From predicate calculus $pre \wedge p' \wedge q \;\Rightarrow\; b$, and
(2') from Hoare logic $\{pre \wedge p' \wedge q\}\ \alpha\ \{\neg p'\}$. Applying the basis rule, **transient** $p' \wedge q \wedge \neg post_f$, i.e., **transient** $p \wedge q$.
2. (Sequencing) In $f\ ;\ g$, **transient** $p \wedge \neg post_f$ in f and **transient** p in g. Inductively, **transient** $p \wedge q \wedge \neg post_f$ in f and **transient** $p \wedge q$ in g. Applying the sequencing rule, **transient** $p \wedge q$.
3. (Concurrency, Inheritance) Similar proofs.

Appendix A.3. Derived Rules for **en**

Appendix A.3.1. Counterparts of rules for transient

This set of derived rules correspond to the similar rules for **transient**. Their proofs are straight-forward using the definition of **en**.

- (Basis)

$$\frac{\begin{array}{c}\{r\}\ f\ \{s\}\\ \text{For every action } b \rightarrow \alpha \text{ with precondition } pre \text{ in the annotation :}\\ pre \wedge p \wedge \neg q \;\Rightarrow\; b\\ \{pre \wedge p \wedge \neg q\}\ \alpha\ \{q\}\end{array}}{\{r\}\ f\ \{p\ \mathbf{en}\ p \wedge s \vee q\ \mid\ s\}}$$

- (Sequencing)

$$\frac{\begin{array}{c}\{r\}\ f\ \{p\ \mathbf{en}\ p \wedge post_f \vee q\ \mid\ post_f\}\\ \{post_f\}\ g\ \{p\ \mathbf{en}\ q\ \mid\ \}\end{array}}{\{r\}\ f\ ;\ g\ \{p\ \mathbf{en}\ q\ \mid\ \}}$$

- (Concurrency)

$$\frac{\begin{array}{c}p\ \mathbf{en}\ q \text{ in } f\\ p \wedge \neg q\ \ \mathbf{co}\ \ p \vee q \text{ in } g\end{array}}{p\ \mathbf{en}\ q \text{ in } f\ [\,]\ g}$$

- (Inheritance) Assuming the proof rule at left the inheritance proof rule at right can be asserted.

Given: $\dfrac{(\forall i ::\ \{r_i\}\ f_i\ \{s_i\})}{\{r\}\ f\ \{s\}}$ Assert: $\dfrac{(\forall i ::\ \{r_i\}\ f_i\ \{p \textbf{ en } q \ \mid\ s_i\})}{\{r\}\ f\ \{p \textbf{ en } q \ \mid\ s\}}$

Appendix A.3.2. Additional derived rules

The following rules are easy to verify by expanding each ensures property by its definition, and using the derived rules for **transient** and **co**. We show one such proof, for the PSP rule. Observe that ensures is only partially conjunctive and not disjunctive, unlike **co**.

1. (implication) $\dfrac{p \Rightarrow q}{p \textbf{ en } q}$
   Consequently, *false* **en** q and p **en** *true* for any p and q.
2. (rhs weakening) $\dfrac{p \textbf{ en } q}{p \textbf{ en } q \vee q'}$
3. (partial conjunction) $\dfrac{p \textbf{ en } q \qquad p' \textbf{ en } q}{p \wedge p' \textbf{ en } q}$
4. (lhs manipulation) $\dfrac{p \wedge \neg q \ \Rightarrow\ p' \ \Rightarrow\ p \vee q}{p \textbf{ en } q \ \equiv\ p' \textbf{ en } q}$
   Observe that $p \wedge \neg q \ \equiv\ p' \wedge \neg q$ and $p \vee q \ \equiv\ p' \vee q$. So, p and q are interchangeable in all the proof rules. As special cases, $p \wedge \neg q \textbf{ en } q \ \equiv\ p \textbf{ en } q \equiv p \vee q \textbf{ en } q$.
5. (PSP) The general rule is at left, and a special case at right using **stable** r as r **co** r.

   (PSP) $\dfrac{p \textbf{ en } q \qquad r \textbf{ co } s}{p \wedge r \textbf{ en } (q \wedge r) \ \vee\ (\neg r \wedge s)}$ (Special case) $\dfrac{p \textbf{ en } q \qquad \textbf{stable } r}{p \wedge r \textbf{ en } q \wedge r}$
6. (Special case of Concurrency)
   $\dfrac{p \textbf{ en } q \text{ in } f \qquad \textbf{stable } p \text{ in } g}{p \textbf{ en } q \text{ in } f \ [\!]\ g}$

□

Proof of (PSP): From the hypotheses:

$$\textbf{transient } p \wedge \neg q \tag{1}$$
$$p \wedge \neg q \textbf{ co } p \vee q \tag{2}$$
$$r \textbf{ co } s \tag{3}$$

We have to show:

$$\textbf{transient } p \wedge r \wedge \neg(q \wedge r) \wedge \neg(\neg r \wedge s) \tag{4}$$
$$p \wedge r \wedge \neg(q \wedge r) \wedge \neg(\neg r \wedge s) \textbf{ co } p \wedge r \vee q \wedge r \vee \neg r \wedge s \tag{5}$$

First, simplify the term on the rhs of (4) and lhs of (5) to $p \wedge r \wedge \neg q \wedge s$. Proof of (4) is then immediate, as a strengthening of (1). For the proof of (5), apply conjunction to (2) and (3) to get:

$p \wedge r \wedge \neg q$ **co** $p \wedge s \vee q \wedge s$

$\equiv$ {expand both terms in rhs}

$p \wedge r \wedge \neg q$ **co** $p \wedge r \wedge s \vee p \wedge \neg r \wedge s \vee q \wedge r \wedge s \vee q \wedge \neg r \wedge s$

$\Rightarrow$ {lhs strengthening and rhs weakening}

$p \wedge r \wedge \neg q \wedge s$ **co** $p \wedge r \vee q \wedge r \vee \neg r \wedge s$

Appendix A.4. Derived Rules for leads-to

The rules are taken from Misra [11] where the proofs are given. The rules are divided into two classes, *lightweight* and *heavyweight*. The former includes rules whose validity are easily established; the latter rules are not entirely obvious. Each application of a heavyweight rule goes a long way toward completing a progress proof.

Appendix A.4.1. Lightweight rules

1. (implication) $\dfrac{p \;\Rightarrow\; q}{p \;\mapsto\; q}$
2. (lhs strengthening, rhs weakening)

$$\frac{p \;\mapsto\; q}{\begin{array}{c} p' \wedge p \;\mapsto\; q \\ p \;\mapsto\; q \vee q' \end{array}}$$

3. (disjunction) $\dfrac{(\forall i ::\; p_i \;\mapsto\; q_i)}{(\forall i ::\; p_i) \;\mapsto\; (\forall i ::\; q_i)}$

   where i is quantified over an arbitrary finite or infinite index set, and p_i, q_i are predicates.
4. (cancellation) $\dfrac{p \;\mapsto\; q \vee r \;,\; r \;\mapsto\; s}{p \;\mapsto\; q \vee s}$

Appendix A.4.2. Heavyweight rules

1. (impossibility) $\dfrac{p \;\mapsto\; \mathit{false}}{\neg p}$
2. (PSP) The general rule is at left, and a special case at right using **stable** r as r **co** r.

(PSP)

$$\frac{\begin{array}{c} p \;\mapsto\; q \\ r \;\textbf{co}\; s \end{array}}{p \wedge r \;\mapsto\; (q \wedge r) \;\vee\; (\neg r \wedge s)}$$

(Special case)

$$\frac{\begin{array}{c} p \;\mapsto\; q \\ \textbf{stable}\; r \end{array}}{p \wedge r \;\mapsto\; q \wedge r}$$

3. (induction) Let M be a total function from program states to a well-founded set $(W, \prec)$. Variable m in the following premise ranges over W. Predicates p and q do not contain free occurrences of variable m.

$$\frac{(\forall m ::\; p \;\wedge\; M = m \;\mapsto\; (p \;\wedge\; M \prec m) \;\vee\; q)}{p \;\mapsto\; q}$$

4. (completion) Let p_i and q_i be predicates where i ranges over a finite set.

$$\frac{\begin{array}{l} (\forall i :: \\ \quad p_i \;\mapsto\; q_i \vee b \\ \quad q_i \;\textbf{co}\; q_i \vee b \\ ) \end{array}}{(\forall i ::\; p_i) \;\mapsto\; (\forall i ::\; q_i) \;\vee\; b}$$

Appendix A.4.3. Lifting Rule

This rule permits lifting a *leads-to* property of f to $f\ []\ g$ with some modifications. Let x be a tuple of some accessible variables of f that includes all variables that f shares with g. Below, X is a free variable, therefore universally quantified. Predicates p and q name accessible variables of f and g. Clearly, any local variable of g named in p or q is treated as a constant in f.

$$\textit{(L).}\quad \frac{\begin{array}{c} p \mapsto q \text{ in } f \\ r \wedge x = X \ \mathbf{co}\ x = X \vee \neg r \text{ in } g \end{array}}{p \mapsto q \vee \neg r \text{ in } f\ []\ g}$$

An informal justification of this rule is as follows. Any p-state in which $\neg r$ holds, $q \vee \neg r$ holds. We show that in any execution of $f\ []\ g$ starting in a $(p \wedge r)$-state q or $\neg r$ holds eventually. If r is falsified by a step of f then $\neg r$ holds. Therefore, assume that every step of f preserves r. Now if any step of g changes the value of x then it falsifies r from the antecedent, i.e., $\neg r$ holds. So, assume that no step of g modifies x. Then g does not modify any accessible variable of f; so, f is oblivious to the presence of g, and it establishes q.

As a special case, we can show

$$\textit{(L')}.\quad \frac{p \mapsto q \text{ in } f}{p \wedge x = M \mapsto q \vee x \neq M \text{ in } f\ []\ g}$$

The formal proof of (L) is by induction on the structure of the proof of $p \mapsto q$ in f. See
`http://www.cs.utexas.edu/users/psp/unity/notes/UnionLiftingRule.pdf`
for details.

Appendix B. Example: Mutual exclusion

We prove a coarse-grained version of a 2-process mutual exclusion program due to Peterson [15]. The given program has a finite number of states, so it is amenable to model-checking. In fact, model-checking is a simpler alternative to an axiomatic proof. We consider this example primarily because this is an instance of a tightly-coupled system where the codes of all the components are typically considered together to construct a proof. In contrast, we construct a composable specification of each component and combine the specifications to derive a proof.

Appendix B.1. Program

The program has two processes M and M'. Process M has two local boolean variables, *try* and *cs* where *try* remains *true* as long as M is attempting to enter the critical section or in it and *cs* is *true* as long as it is in the critical section; M' has try' and cs'. They both have access to a shared boolean variable *turn*. It simplifies coding and proof to postulate an additional boolean variable $turn'$ that is the complement of *turn*, i.e., $turn' \equiv \neg turn$.

The global initialization and the code for M, along with a local annotation, is given below. The code of M' is the *dual* of M, obtained by replacing each variable in M by its primed counterpart. Henceforth, the primed and the unprimed versions of the same variable re duals of each other.

The "unrelated computation" below refers to computation preceding the attempt to enter the critical section that does not access any of the relevant variables. This computation may or may not terminate in any iteration.

```
initially $cs, cs' = false, false$ — global initialization

$M$: initially $try = false$
$\{\neg try,\ \neg cs\}$
loop
  — unrelated computation that may not terminate;
  $\{\neg try,\ \neg cs\}$   $\alpha$:  $try,\ turn := true,\ true$;
  $\{try,\ \neg cs\}$        $\beta$:  $\neg try' \vee turn' \ \rightarrow\ cs := true$; — Enter critical section
  $\{try,\ cs\}$             $\gamma$:  $try,\ cs := false,\ false$ — Exit critical section
forever
```

Given that M' is the dual of M, from any property of M obtain its dual as a property of M'. And, for any property of M [] M' its dual is a property of M' [] M, i.e., M [] M', thus reducing the proof length by around half.

Remarks on the program. The given program is based on a simplification of an algorithm in Peterson [15]. In the original version the assignment in α may be decoupled to the sequence $try := true$; $turn := true$. The tests in β may be made separately for each disjunct in arbitrary order. Action γ may be written in sequential order $try := false;\ cs := false$. These changes can be easily accommodated within our proof theory by introducing auxiliary variables to record the program control.

Appendix B.2. Safety and progress properties

It is required to show in $M \; [] \; M'$ (1) the safety property: both M and M' are never simultaneously within their critical sections, i.e., **invariant** $\neg(cs \wedge cs')$, and (2) the progress property: any process attempting to enter its critical section will succeed eventually, i.e., $try \mapsto cs$ and $try' \mapsto cs'$; we prove just $try \mapsto cs$ since its dual also holds.

Appendix B.2.1. Safety proof: **invariant** $\neg(cs \wedge cs')$

We prove below:

invariant $cs \Rightarrow try$ in $M \; [] \; M'$ (S1)
invariant $cs' \wedge try \Rightarrow turn$ in $M \; [] \; M'$ (S2)

Mutual exclusion is immediate from (S1) and (S2), as follows.

$$\begin{array}{ll}
 & cs \wedge cs' \\
\Rightarrow & \{\text{from (S1) and its dual}\} \\
 & cs \wedge try \wedge cs' \wedge try' \\
\equiv & \{\text{rewriting}\} \\
 & (cs' \wedge try) \wedge (cs \wedge try') \\
\Rightarrow & \{\text{from (S2) and its dual}\} \\
 & turn \wedge turn' \\
\equiv & \{turn \text{ and } turn' \text{ are complements}\} \\
 & false
\end{array}$$

Proofs of (S1) and (S2). First, we show the following stable properties of M, which constitute its safety specification, from which (S1) and (S2) follow.

stable $cs \Rightarrow try$ in M (S3)
stable $try \Rightarrow turn$ in M (S4)
stable $cs \wedge try' \Rightarrow turn'$ in M (S5)

The proofs of (S3), (S4) and (S5) are entirely straight-forward, but each property has to be shown stable for each action. We show the proofs in Table B.1, where each row refers to one of the predicates in (S3 – S5) and each column to an action. An entry in the table is either: (1) "post: p" claiming that since p is a postcondition of this action the given predicate is stable, or (2) "unaff." meaning that the variables in the given predicate are unaffected by this action execution. The only entry that is left out is for the case that β preserves (S5): $cs \wedge try' \Rightarrow turn'$; this is shown as follows. The guard of β can be written as $try' \Rightarrow turn'$ and execution of β affects neither try' nor $turn'$, so $try' \Rightarrow turn'$ is a postcondition, and so is $cs \wedge try' \Rightarrow turn'$.

		α	β	γ
(S3)	$cs \Rightarrow try$	post: $\neg cs$	post: try	post: $\neg cs$
(S4)	$try \Rightarrow turn$	post: $turn$	unaff.: try, $turn$	post: $\neg try$
(S5)	$cs \wedge try' \Rightarrow turn'$	post: $\neg cs$	see text	post: $\neg cs$

Table B.1: Proofs of (S3), (S4) and (S5)

Now we are ready to prove (S1) and (S2). The predicates in (S1) and (S2) hold initially because **initially** $cs, cs' = \mathit{false}, \mathit{false}$. Next, we show these predicates to be stable. The proof is compositional, by proving each predicate to be stable in both M and M'. The proof is simplified by duality of the codes.

- (S1) **stable** $cs \Rightarrow try$ in M [] M':

stable $cs \Rightarrow try$ in M	, (S3)
stable $cs \Rightarrow try$ in M'	, cs and try are constant in M'
stable $cs \Rightarrow try$ in M [] M'	, Inheritance rule

- (S2) **stable** $cs' \wedge try \Rightarrow turn$ in M [] M':

stable $try \Rightarrow turn$ in M	, (S4)
stable $cs' \wedge try \Rightarrow turn$ in M	, cs' constant in M
stable $cs' \wedge try \Rightarrow turn$ in M'	, dual of (S5)
stable $cs' \wedge try \Rightarrow turn$ in M [] M'	, Inheritance rule

Appendix B.2.2. Progress proof: $try \mapsto cs$ in M [] M'

First, we prove a safety property:

$$try \wedge \neg cs \ \textbf{co} \ try \vee cs \text{ in } M \ [] \ M' \qquad \text{(S6)}$$

To prove (S6) first prove $try \wedge \neg cs$ **co** $try \vee cs$ in M, which is entirely straightforward. Now variables try and cs are local to M, therefore **stable** $try \wedge \neg cs$ in M', and through rhs weakening, $try \wedge \neg cs$ **co** $try \vee cs$ in M'. Using inheritance (S6) follows. □

Next we prove a progress property:

$$try \wedge (\neg try' \vee turn') \ \mapsto \ \neg try \text{ in } M \ [] \ M' \qquad \text{(P)}$$

First, prove $try \wedge (\neg try' \vee turn')$ **en** $\neg try$ in M, using the sequencing rule for **en** ; intuitively, in a $try \wedge (\neg try' \vee turn')$-state the program control is never at α, execution of β terminates while preserving $try \wedge (\neg try' \vee turn')$, and execution of γ establishes $\neg try$.

Using duality on (S4) get **stable** $try' \Rightarrow turn'$ in M', i.e., **stable** $\neg try' \vee turn'$ in M'. And try is local to M, so **stable** try in M'. Conjoining these two properties, **stable** $try \wedge (\neg try' \vee turn')$ in M'. Apply the concurrency rule for **en** with **stable** $try \wedge (\neg try' \vee turn')$ in M' and $try \wedge (\neg try' \vee turn')$ **en** $\neg try$ in M to conclude that $try \wedge (\neg try' \vee turn')$ **en** $\neg try$ in M [] M'.

Now apply the basis rule for $\mapsto$ to conclude the proof of (P). □

Proof of $try \mapsto cs$ **in** $M \; [] \; M'$**:** All the properties below are in $M \; [] \; M'$. Conclude from (P), using lhs strengthening,

$$try \wedge \neg try' \mapsto \neg try \tag{P1}$$
$$try \wedge turn' \mapsto \neg try \tag{P2}$$

The main proof:

$try' \wedge turn \mapsto \neg try'$	, duality applied to (P2)
$try \wedge try' \wedge turn \mapsto \neg try'$	, lhs strengthening
$try \wedge \neg(try' \wedge turn) \mapsto \neg try$	, rewriting (P) using $\neg turn \equiv turn'$
$try \mapsto \neg try \vee \neg try'$	, disjunction of the above two
$try \mapsto \neg try \vee (\neg try' \wedge try)$	, rewriting the rhs
$try \mapsto \neg try$	, cancellation using (P1)
$try \wedge \neg cs$ **co** $try \vee cs$	, (S6)
$try \wedge \neg cs \mapsto cs$	, PSP of above two
$try \mapsto cs$	, disjunction with $try \wedge cs \mapsto cs$

Appendix C. Example: Associative, Commutative fold

We consider a recursively defined program f where the code of f_1 is given and f_{k+1} is defined to be $f_1 \; [] \; f_k$. This structure dictates that the specification s_k of f_k must permit proof of (1) s_1 from the code of f_1, and (2) s_{k+1} from s_1 and s_k, using induction on k. This example illustrates the usefulness of the various composition rules for the perpetual properties. The program is not easy to understand intuitively; it does need a formal proof.

Appendix C.1. Informal Problem Description

Given is a bag u on which $\oplus$ is a commutative and associative binary operation. Define Σu, *fold* of u, to be the result of applying $\oplus$ repeatedly to all pairs of elements of u until there is a single element. It is required to replace all the elements of u by Σu. Program f_k, for $k \geq 1$, decreases the size of u by k while preserving its fold. That is, f_k transforms the original bag u' to u such that: (1) $\Sigma u = \Sigma u'$, and (2) $|u| + k = |u'|$, provided $|u'| > k$, where $|u|$ is the size of u. Therefore, execution of f_{n-1}, where n is the size of u' and $n > 1$, computes a single value in u that is the fold of the initial bag.

Below, $get(x)$ removes an item from u and assigns its value to variable x. This operation can be completed only if u has an item. And $put(x \oplus y)$, a non-blocking action, stores $x \oplus y$ in u. The formal semantics of *get* and *put* are given by the following assertions where u' is constant:

$$\{u = u'\} \; get(z) \; \{u' = u \cup \{z\}\}$$
$$\{u = u'\} \; put(z) \; \{u = u' \cup \{z\}\}$$

The fold program f_k for all k, $k \geq 1$, is given by:

$$f_1 \;\; = \;\; |u| > 0 \;\rightarrow\; get(x); \;\; |u| > 0 \;\rightarrow\; get(y); \;\; put(x \oplus y)$$
$$f_{k+1} = \;\; f_1 \; [] \; f_k, \quad k \geq 1$$

Specification and proof of safety properties appear in Section Appendix C.2, next, and progress properties in Section Appendix C.3. Observe that

Appendix C.2. Terminal property

The relevant safety property of f_k, for all k, $k \geq 1$, is a terminal property:

$$\{u = u'\}\ f_k\ \{\Sigma u = \Sigma u',\ |u| + k = |u'|\}$$

This property can not be proved from a local annotation alone because it names the shared variable u in its pre- and postconditions. We suggest an enhanced safety property using certain auxiliary variables.

Auxiliary variables. The following auxiliary variables of f_k are local to it:

1. w_k: the bag of items removed from u that are, as yet, unfolded. That is, every *get* from u puts a copy of the item in w_k, and $put(x \oplus y)$ removes x and y from w_k. Initially $w_k = \{\}$.
2. nh_k: the number of halted threads where a thread halts after completing a *put*. A *get* does not affect nh_k and a *put* increments it. Initially $nh_k = 0$.

Introduction of any auxiliary variable aux_k for f_k follows a pattern: (1) specify the initial value of aux_k, (2) define aux_1 by modifying the code of f_1, corresponding to the basis of a recursive definition, and (3) define aux_{k+1} in terms of aux_1 and aux_k.

We adopt this pattern for defining w_k and nh_k for all k, $k \geq 1$. First, the initial values of w_k and nh_k are $\{\}$ and 0, respectively. Second, w_1 and nh_1 are defined below in f_1. Third, $w_{k+1} = w_1 \cup w_k$ and $nh_{k+1} = nh_1 + nh_k$, for all k. The modified program for f_1 is as follows where $\langle \cdots \rangle$ is an action.

```
$f_1 = |u| > 0 \ \rightarrow\ \langle get(x);\ w_1 := w_1 \cup \{x\}\rangle;$
      $|u| > 0 \ \rightarrow\ \langle get(y);\ w_1 := w_1 \cup \{y\}\rangle;$
      $\langle put(x \oplus y);\ w_1 := w_1 - \{x, y\};\ nh_1 := 1\rangle$
```

Note: The definition of auxiliary variables is problematic for nh_2, for instance; by definition, $nh_2 = nh_1 + nh_1$. However, these are two different instances of nh_1 referring to the local variables of the two instances of f_1 in f_2. This is not a problem in the forthcoming proofs because the proofs always refer to indices 1, k and $k+1$, and, nh_k is treated as being different from nh_1.

Specification of safety property. The safety specification of f_k, for all k, $k \geq 1$, is given by:

$$\begin{array}{l} \qquad\quad \{u = u',\ w_k = \{\},\ nh_k = 0\} \\ \qquad\qquad\qquad\qquad f_k \\ \{\textbf{constant}\ \Sigma(u \cup w_k),\ \textbf{constant}\ |u| + |w_k| + nh_k \\ \ \ \mid\ w_k = \{\},\ nh_k = k\} \end{array} \tag{S}$$

• Proof of (S) for f_1: Construct the following local annotation of f_1.

$$
\begin{array}{l}
\qquad \{w_1 = \{\},\ nh_1 = 0\} \\
|u| > 0 \ \rightarrow\ \langle get(x);\ w_1 := w_1 \cup \{x\}\rangle; \\
\qquad \{w_1 = \{x\},\ nh_1 = 0\} \\
\quad |u| > 0 \ \rightarrow\ \langle get(y);\ w_1 := w_1 \cup \{y\}\rangle; \\
\qquad \{w_1 = \{x, y\},\ nh_1 = 0\} \\
\qquad \langle put(x \oplus y);\ w_1 := w_1 - \{x, y\};\ nh_1 := 1\rangle \\
\qquad \{w_1 = \{\},\ nh_1 = 1\}
\end{array}
$$

The perpetual and terminal properties of f_1 in (S) are easily shown using this annotation and employing the semantics of *get* and *put*.

• Proof of (S) for f_{k+1}: by induction on k. Use the following abbreviations in the proof.

$$
\begin{array}{lcl}
a_k & \equiv & w_k = \{\} \ \wedge\ nh_k = 0 \\
b_k & \equiv & \textbf{constant}\ \Sigma(u \cup w_k),\ \textbf{constant}\ |u| + |w_k| + nh_k \\
c_k & \equiv & w_k = \{\} \ \wedge\ nh_k = k
\end{array}
$$

$$
\begin{array}{lll}
\{a_1\}\ f_1\ \{b_1 \mid c_1\} & \text{, from the annotation of } f_1 & \\
\{a_1\}\ f_1\ \{b_{k+1} \mid c_1\} & \text{, } w_k,\ nh_k \text{ constant in } f_1 & (1) \\
\{a_k\}\ f_k\ \{b_k \mid c_k\} & \text{, inductive hypothesis} & \\
\{a_k\}\ f_k\ \{b_{k+1} \mid c_k\} & \text{, } w_1,\ nh_1 \text{ constant in } f_k & (2) \\
\{a_1 \wedge a_k\}\ f_1\ []\ f_k\ \{c_1 \wedge c_k\} & \text{, join proof rule on (1,2)} & \\
\{a_1 \wedge a_k\}\ f_1\ []\ f_k\ \{b_{k+1} \mid c_1 \wedge c_k\} & \text{, inheritance on (1,2)} & \\
\{a_{k+1}\}\ f_{k+1}\ \{b_{k+1} \mid c_{k+1}\} & \text{, } a_{k+1} \Rightarrow a_1 \wedge a_k \text{ and } c_1 \wedge c_k \Rightarrow c_{k+1} &
\end{array}
$$

The terminal property $\{u = u'\}\ f_k\ \{\Sigma u = \Sigma u',\ |u| + k = |u'|\}$ follows from (S) as follows. The initial values of $\Sigma(u \cup w_k)$ and $|u| + |w_k| + nh_k$ are, respectively, $\Sigma u'$ and $|u'|$, so deduce that **invariant** $\Sigma(u \cup w_k) = \Sigma u'$ and also **invariant** $|u| + |w_k| + nh_k = |u'|$. Given that f_k is a program whose terminal properties are being proved, we may apply the invariance rule of Section 4.3 with these invariants. So, deduce $\Sigma u = \Sigma u'$, $|u| + k = |u'|$ as a postcondition of f_k with the given precondition.

Appendix C.3. Specification and proof of progress property

The relevant progress property is that if u has more than k elements initially, f_k terminates eventually. That is, $|u'| > k \mapsto nh_k = k$ in f_k. Initially $|u'| = |u| + |w_k| + nh_k$, so it is enough to prove $|u| + |w_k| + nh_k > k \mapsto nh_k = k$. Abbreviate $|u| + |w_k| + nh_k > k$ by p_k and $nh_k = k$ by q_k so that for all k, $k \geq 1$, the required progress property is:

$$p_k \mapsto q_k \text{ in } f_k \qquad \text{(P)}$$

The proof of (P) is by induction on k, as shown in Sections Appendix C.3.1 and Appendix C.3.2.

Appendix C.3.1. Progress proof, $p_1 \mapsto q_1$ in f_1

We show that p_1 **en** q_1, from which $p_1 \mapsto q_1$ follows by applying the basis rule of *leads-to*. We reproduce the annotation of f_1 for easy reference.

```
        {w_1 = {}, nh_1 = 0}
|u| > 0  →  ⟨get(x); w_1 := w_1 ∪ {x}⟩;
        {w_1 = {x}, nh_1 = 0}
  |u| > 0  →  ⟨get(y); w_1 := w_1 ∪ {y}⟩;
        {w_1 = {x, y}, nh_1 = 0}
      ⟨put(x ⊕ y); w_1 := w_1 − {x, y}; nh_1 := 1⟩
        {w_1 = {}, nh_1 = 1}
```

Next, prove p_1 **en** q_1 using the sequencing rule of **en**, from Section Appendix A.3.1. It amounts to showing that if p_1 holds before any action then the action is effectively executed and q_1 holds on completion of f_1. As shown in Section Appendix C.2 p_1 is stable, and from the annotation q_1 holds on completion of f_1. Therefore, it suffices to show that if p_1 holds initially then every action is effectively executed. The *put* action is always effectively executed. Using the given annotation, the verification conditions for the two *get* actions are shown below in full:

$$w_1 = \{\} \wedge nh_1 = 0 \wedge |u| + |w_1| + nh_1 > 1 \;\Rightarrow\; |u| > 0, \text{ and}$$
$$w_1 = \{x\} \wedge nh_1 = 0 \wedge |u| + |w_1| + nh_1 > 1 \;\Rightarrow\; |u| > 0$$

These are easily proved.

Appendix C.3.2. Progress proof, $p_{k+1} \mapsto q_{k+1}$ in f_{k+1}

The main part of the proof uses the observation that every action, *put* and *get*, reduces a well-founded *metric*. The metric is the pair $(|u| + |w_k|, |u|)$ and the order relation is lexicographic. Clearly, the metric is bounded from below because each set size is at least 0. The crux of the proof is to show that if program f_k is not terminated there is some action that can be executed, i.e., there is no deadlock. Henceforth, abbreviate $(|u| + |w_k|, |u|)$ by z_k and any pair of non-negative integers by n.

First, observe that z_k can only decrease or remain the same in f_k, that is, **stable** $z_k \preceq n$ in f_k, where $\preceq$ is the lexicographic ordering. The proof is by induction on k and it follows the same pattern as all other safety proofs. In f_1, informally, every effective *get* preserves $|u| + |w_1|$ and decreases $|u|$, and a *put* decreases $|u|+|w_1|$. For the proof in f_{k+1}: from above, **stable** $z_1 \preceq n$ in f_1, and inductively **stable** $z_k \preceq n$ in f_k. Since **constant** w_k in f_1 and **constant** w_1 in f_k, **stable** $z_{k+1} \preceq n$ in both f_1 and f_k. Apply the inheritance rule to conclude that **stable** $z_{k+1} \preceq n$ in f_{k+1}.

The progress proof of $p_{k+1} \mapsto q_{k+1}$ in f_{k+1} is based on two simpler progress results, (P1) and (P2). (P1) says that any execution starting from p_{k+1} results in the termination of either f_1 or f_k. And, (P2) says that once either f_1 or f_k terminates the other component also terminates. The desired result, $p_{k+1} \mapsto q_{k+1}$, follows by using transitivity on (P1) and (P2).

$p_{k+1} \mapsto p_{k+1} \wedge (q_1 \vee q_k)$ in f_{k+1} (P1)
$p_{k+1} \wedge (q_1 \vee q_k) \mapsto q_{k+1}$ in f_{k+1} (P2)

The proofs mostly use the derived rules of *leads-to* from Section Appendix A.4. Note that z_{k+1} includes all the shared variables between f_1 and f_k, namely u, so that the lifting rule can be used with z_{k+1}. Also note that $p_{k+1} \Rightarrow (p_1 \vee p_k)$, $p_{k+1} \wedge q_1 \Rightarrow p_k$, $p_{k+1} \wedge q_k \Rightarrow p_1$ and $q_1 \wedge q_k \Rightarrow q_{k+1}$. As shown in Section Appendix C.2 p_k is constant, hence stable, and q_k is also stable in f_k.

Proof of (P1). $p_{k+1} \mapsto p_{k+1} \wedge (q_1 \vee q_k)$ in f_{k+1}: Below all properties are in f_{k+1}. Lifting rule refers to rule (L') of Section Appendix A.4.3. We have already shown $p_1 \mapsto q_1$ and, inductively, $p_k \mapsto q_k$.

$p_1 \wedge z_{k+1} = n \mapsto q_1 \vee z_{k+1} \neq n$, Lifting rule on $p_1 \mapsto q_1$ in f_1
$p_k \wedge z_{k+1} = n \mapsto q_k \vee z_{k+1} \neq n$, Lifting rule on $p_k \mapsto q_k$ in f_k
$(p_1 \vee p_k) \wedge z_{k+1} = n \mapsto (q_1 \vee q_k) \vee z_{k+1} \neq n$
, disjunction
$(p_1 \vee p_k) \wedge z_{k+1} = n \mapsto (q_1 \vee q_k) \vee z_{k+1} \prec n$
, (PSP) with **stable** $z_{k+1} \preceq n$
$p_{k+1} \wedge (p_1 \vee p_k) \wedge z_{k+1} = n \mapsto p_{k+1} \wedge (q_1 \vee q_k) \vee p_{k+1} \wedge z_{k+1} \prec n$
, conjunction with **stable** p_{k+1}
$p_{k+1} \wedge z_{k+1} = n \mapsto p_{k+1} \wedge z_{k+1} \prec n \vee p_{k+1} \wedge (q_1 \vee q_k)$
, $p_{k+1} \Rightarrow (p_1 \vee p_k)$
$p_{k+1} \mapsto p_{k+1} \wedge (q_1 \vee q_k)$, induction rule of *leads-to*

Proof of (P2). $p_{k+1} \wedge (q_1 \vee q_k) \mapsto q_{k+1}$ in f_{k+1}: Below all properties are in f_{k+1}.

$p_1 \wedge z_{k+1} = n \mapsto q_1 \vee z_{k+1} \neq n$, Lifting rule on $p_1 \mapsto q_1$ in f_1
$p_1 \wedge z_{k+1} = n \mapsto q_1 \vee z_{k+1} \prec n$, conjunction with **stable** $z_{k+1} \preceq n$
$q_k \wedge p_1 \wedge z_{k+1} = n \mapsto q_k \wedge q_1 \vee q_k \wedge z_{k+1} \prec n$
, conjunction with **stable** q_k
$p_{k+1} \wedge q_k \wedge p_1 \wedge z_{k+1} = n \mapsto p_{k+1} \wedge q_k \wedge q_1 \vee p_{k+1} \wedge q_k \wedge z_{k+1} \prec n$
, conjunction with **stable** p_{k+1}
$p_{k+1} \wedge q_k \wedge z_{k+1} = n \mapsto p_{k+1} \wedge q_k \wedge z_{k+1} \prec n \vee p_{k+1} \wedge q_k \wedge q_1$
, $p_{k+1} \wedge q_k \Rightarrow p_1$
$p_{k+1} \wedge q_k \mapsto p_{k+1} \wedge q_1 \wedge q_k$, induction rule of *leads-to*
$p_{k+1} \wedge q_k \mapsto q_1 \wedge q_k$, rhs weakening
$p_{k+1} \wedge q_1 \mapsto q_1 \wedge q_k$, similarly
$p_{k+1} \wedge (q_1 \vee q_k) \mapsto q_{k+1}$, disjunction and $q_1 \wedge q_k \Rightarrow q_{k+1}$

Appendix D. References

[1] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently.* PhD thesis, Massachusetts Institute Of Technology, September 1995.

[2] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[3] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 480–494, 2010.

[4] Edsger W. Dijkstra. Guarded commands, nondeterminacy, and the formal derivation of programs. *Communications of the ACM*, 8:453–457, 1975.

[5] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[6] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580,583, 1969.

[8] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP83*, pages 321–332. North-Holland, 1983.

[9] C. B. Jones. Tentative steps toward a development method for interfering programs. *TOPLAS*, 5(4):596–619, 1983.

[10] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, SE-3(2):125–143, Mar 1977.

[11] Jayadev Misra. *A Discipline of Multiprogramming.* Monographs in Computer Science. Springer-Verlag New York Inc., New York, 2001.

[12] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE*, SE,7(4):417–426, July 1981.

[13] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19:279–285, May 1976.

[14] S. Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[15] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[16] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society, Washington, DC, USA, 2002.