Bilateral Proofs of Safety and Progress Properties of Concurrent Programs
(Working Draft)

Jayadev Misra
December 8, 2015

Contents

1 Introduction 3

2 Program and Execution Model 4
  2.1 Program Structure 4
  2.2 Program execution 5
  2.3 Example: Distributed counter 6

3 Introduction to the proof theory 6
  3.1 Local Annotation 6
  3.2 Example: Distributed Counter, contd. 7
  3.3 Notational conventions for specifications 7

4 Safety Properties 8
  4.1 Safety Property co 8
  4.2 Special cases of co 8
  4.3 Meta-rules 9
  4.4 Example: Distributed Counter, contd. 9

5 Progress Properties 10
  5.1 Progress Property: transient 10
  5.2 Progress Property: ensures 12
  5.3 Progress Property: Leads-to 14

6 Concluding Remarks, To be done 15

A Appendix: Derived Rules 16
  A.1 Derived Rules for co and its special cases 16
  A.2 Derived Rules for transient 17
  A.3 Derived Rules for en 17
A.3.1 Counterparts of rules for transient .......................... 17
A.3.2 Additional derived rules ........................................ 18
A.4 Derived Rules for leads-to ........................................ 19
  A.4.1 Lightweight rules ................................................ 19
  A.4.2 Heavyweight rules .............................................. 19
  A.4.3 Lifting Rule ..................................................... 20
B Example: Mutual exclusion ........................................... 20
  B.1 Program .................................................................... 21
  B.2 Safety and progress properties ................................... 22
    B.2.1 Safety proof: invariant \(\neg(cs \land cs')\) ................. 22
    B.2.2 Progress proof: try \(\mapsto\) cs in \(M \parallel M'\) ......... 23
C Example: Associative, Commutative fold .......................... 24
  C.1 Proof of terminal property (T) .................................. 25
  C.2 Proof of Progress property (P) .................................. 27
    C.2.1 Progress proof \(p_1 \mapsto q_1\) in \(f_1\) ...................... 27
    C.2.2 Progress proof, \(p_{k+1} \mapsto q_{k+1}\) in \(f_{k+1}\) .......... 27
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 a cache coherence protocol, are typically proved using model checking, which has 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 [5], in which a program is specified by a pair 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 [6] and Owicki and Lamport [8].

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 [7]. 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 B includes a proof of a mutual exclusion program. Unlike traditional proofs, it is based on composing the specifications of the components. Appendix C includes proof of a recursively-defined concurrent program, where code of one component is only implicit, again using composable specifications.

2 Program and Execution Model

2.1 Program Structure

The syntax used for programs and its components is given below.

\[
\begin{align*}
\alpha &::= \text{command} := \text{terminating code} \\
&::= \text{action} := \text{predicate} \rightarrow \alpha \\
f, g &::= \text{component} ::= \text{action} \mid f \parallel g \mid \text{seq}(f_0, f_1, \cdots f_n) \\
\text{program} &::= f
\end{align*}
\]

A command is a piece of code that is guaranteed to terminate when executed in isolation. We assume that this guarantee is independently available, and the command is always scheduled to be executed in isolation, i.e., without preemption. An action is a conditional command; the execution of the command can be started only if the associated predicate, called a guard, holds. 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 \( f \parallel g \) 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. The direct subcomponents of \( \text{seq}(f_0, f_1, \cdots f_n) \) are \( f_0, f_1, \cdots f_n \). A subcomponent of a program is either a direct subcomponent or a subcomponent of some direct subcomponent. Note that a program is never empty.

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

A program is a component that is 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 execution as a component. Any accessible variable of a component in a
A sequential program is local to it. However, a local variable $x$ of $f$ 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 program step is an instance of the execution of one of its actions. The execution of action $b \rightarrow \alpha$ is effective in a given state if $b$ holds in that state; then $\alpha$ is executed to completion without preemption by another component. Execution of the action is ineffective if $b$ is false in that state. Effective execution moves the program control beyond the action; an ineffective step leaves the program control at its current point and does not alter the state. Ineffective execution models busy-waiting whereby the action execution is retried at some future moment.

The value read from a local variable in a step is the last value written to it in the component. The value read from a non-local variable is arbitrary, to model that a concurrent component may have written an arbitrary value to it.

**Terminology** A program point is any point in the program text preceding an action, or the point of program termination. A control point is a program point at which the program control resides at any moment during its execution. There may be multiple control points simultaneously because of concurrency.

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 [4] in his classic paper. In a deadlocked execution an action that is permanently blocked is attempted infinitely often ineffectively, and the control resides permanently at the point preceding the action.
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 [7].

The protocol 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 [3]; in each execution of the statement an alternative whose guard holds is executed without preemption.

\begin{verbatim}
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
\end{verbatim}

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 Local Annotation

Local annotation of a component associates predicates with control points so that whenever the program control is at a particular point the associated predicate holds. This guarantee is met even under concurrent execution of the component with other components. The annotation strategy is to use only local predicates of a component in the annotation. Since a component has exclusive write-access to all variables in a local predicate during its execution, the values of these variables remain unchanged as long as the program control for the component does not change, hence the predicate continues to hold.

A local annotation is constructed using the traditional proof rules for the sequential constructs. The proof rule for guarded action and for the join construct are given below. Note that the join rule is not sound in general, but it is sound when applied only with local predicates. Call all these rules basic proof rules to distinguish them from other rules defined later in the paper.

Henceforth, all annotations in proof rules are local annotations.
A shortcoming of local annotation is that a variable that is local to \( f \parallel g \) but non-local to both \( f \) and \( g \) can not appear in a local annotation by the application of these rules alone. The invariance meta-rule, given below in Section 4.3, overcomes this problem.

### 3.2 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.

\[
f_j ::
\begin{align*}
\text{initially} & \quad \text{old}_j, \text{new}_j = 0, 0 \\
\text{loop} & \quad \{true\} \\
\alpha_j & \quad \text{new}_j := \text{old}_j + 1; \\
\text{if} & \quad \beta_j :: \{\text{new}_j = \text{old}_j + 1\} \text{ ctr} = \text{old}_j \rightarrow \text{ctr} := \text{new}_j \quad \{true\} \\
| & \quad \gamma_j :: \{\text{new}_j = \text{old}_j + 1\} \text{ ctr} \neq \text{old}_j \rightarrow \text{old}_j := \text{ctr} \quad \{true\} \\
\} & \quad \{true\} \\
\text{forever} &
\end{align*}
\]

### 3.3 Notational conventions for specifications

Write the specification of a component in the form \( \{r\} f \{Q \mid s\} \) where:

1. \( r \) and \( s \) are pre- and postcondition of \( f \), and
2. \( Q \) is a set of perpetual properties of \( f \) under precondition \( r \).

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.
4 Safety Properties

A safety property is perpetual. We develop a safety property, \( \text{co} \), and its special cases, taken from Misra [7]. 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 \( \text{co} \)

Write \( p \text{ co} q \) 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 \( p \). This meaning has two consequences: (1) in any execution of \( f \) as a program once \( p \) holds it continues to hold until \( q \) is established, though \( q \) may never be established, and (2) as a composition rule, \( p \text{ co} q \) holds in program if it holds in every component of it. For example, the informal statement “every change in integer variable \( x \) when \( x = 0 \) can only increase its value” may be formalized as \( x = 0 \text{ co} x \geq 0 \). This permits \( x \) to be decreased only when \( x \) is non-zero.

Formal definition of \( \text{co} \) is given by

\[
\begin{array}{c}
\{r\} f \{s\} \\
\{pre \land b \land p\} \alpha \{q\} \\
\{r\} f \{p \text{ co} q\} [s] \\
\end{array}
\]

Note: The guard \( b \) of an action of \( f \) need not be local to \( f \). Recall that in the effective execution of \( b \rightarrow \alpha \) the evaluation of \( b \) and execution of \( \alpha \) are indivisible. So, once the action execution begins \( b \) can not be falsified, i.e., \( b \) can be taken as a precondition for effective execution of the action.

4.2 Special cases of \( \text{co} \)

Define special cases of \( \text{co} \) 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 \( f \)

- \[ \text{stable} \ p \equiv p \text{ co} p \]
- \[ \text{constant} \ e \equiv (\forall c :: \text{stable} \ e = c) \]
- \[ \text{invariant} \ p \equiv \text{initially} \ p \text{ and stable} \ p \]

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 \( \text{co} \) and some of its special cases, which are heavily used in actual proofs, are given in Appendix A.1.
4.3 Meta-rules

The following general rules apply for specifications.

- (lhs strengthening and rhs weakening)
  \[
  \begin{align*}
  r' & \Rightarrow r, \ s \Rightarrow s', \ Q' \subseteq Q, \ r' \text{ and } s' \text{ are local to } f \\
  \{r'\} f \{Q \mid s\} & \Rightarrow \{r\} f \{Q' \mid s'\}
  \end{align*}
  \]

- (Conjunction)
  \[
  \begin{align*}
  \{r\} f \{Q \mid s\} & \Rightarrow \{r'\} f \{Q' \mid s'\} \\
  \{r \land r'\} f \{Q \cup Q' \mid s \land s'\}
  \end{align*}
  \]

- (Inheritance)
  \[
  \text{Given: } \forall i :: \{r_i\} f_i \{s_i\} \Rightarrow \{r\} f \{s\}
  \]
  \[
  \text{Assert: } \forall i :: \{r_i\} f_i \{\sigma_i \mid s_i\}
  \]

- (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.

The lhs strengthening and rhs weakening rule is inspired by a similar rule for Hoare-triples for the pre- and postconditions. Additionally, since the properties in $Q$ are independent, any number of them may be removed. For a program all variables are local; so, the rule can be applied to assert over all variables in $r$ and $s$.

In the conjunction rule, $\{r\} f \{Q \mid s\}$ asserts that any execution of $f$ starting in a $r$-state may terminate only in a $s$-state, and each property in $Q$ holds throughout the execution. Dual remarks apply to $\{r'\} f \{Q' \mid s'\}$. Therefore, starting in a $r \land r'$-state, at termination both $s \land s'$ hold and so do every property in $Q \cup Q'$.

Inheritance rule is based on the following observation about safety properties: if a safety property holds for all components of $f$ it holds for $f$ as well. Given the basic proof rule at left the inheritance proof rule at right can be asserted for any safety property $\sigma$.

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 $f \parallel g$, that may not be a local predicate of either $f$ or $g$, could be conjoined to predicates in an annotation of $f \parallel g$. Additionally, all variables in a program are local; so, any invariant can be substituted for true for 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., $ctr = m \implies ctr = m \lor ctr = m + 1$ in $f$.

We reproduce the annotation from Section 3.2.
\[ f_j :: \]
\[ \text{initially} \quad \text{old}_j, \text{new}_j = 0, 0 \]
\[ \{ \text{true} \} \]
\[ \text{loop} \]
\[ \{ \text{true} \} \]
\[ \alpha_j :: \text{new}_j := \text{old}_j + 1; \]
\[ \{ \text{new}_j = \text{old}_j + 1 \} \]
\[ \text{if} [ \begin{array}{ll}
\beta_j :: \{ \text{new}_j = \text{old}_j + 1, \text{ctr} = \text{old}_j \rightarrow \text{ctr} := \text{new}_j \} \{ \text{true} \} \\
\gamma_j :: \{ \text{new}_j = \text{old}_j + 1, \text{ctr} \neq \text{old}_j \rightarrow \text{old}_j := \text{ctr} \} \{ \text{true} \} 
\end{array} ] \]
\[ \{ \text{true} \} \]
\[ \text{forever} \]

We show \( \text{ctr} = m \text{ co } \text{ctr} = m \lor \text{ctr} = m + 1 \) in \( f_j \). Since only \( \beta_j \) may change the value of \( \text{ctr} \) we need to show:

\[ \{ \text{ctr} = m \land \text{new}_j = \text{old}_j + 1 \land \text{ctr} = \text{old}_j \} \text{ctr} := \text{new}_j \{ \text{ctr} = m \lor \text{ctr} = m + 1 \} \]

In this and subsequent proofs, we omit most of the mechanical aspects, such as generation of verification conditions and their formal proofs. The proof above is immediate. Since \( \text{ctr} = m \text{ co } \text{ctr} = m \lor \text{ctr} = m + 1 \) holds in \( f_j \), for all \( j \), it holds in \( f \), using the inheritance rule.

## 5 Progress Properties

We are 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 \textit{leads-to}, is defined in Section 5.3 (page 14). We introduce two simpler progress properties, \textit{transient} and \textit{ensures} first. Transient is a fundamental progress property, the counterpart of the safety property \textit{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 \textit{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 \( f \)’s execution, \( \neg p \) holds then or eventually in that execution. Consequently, a transient predicate never holds at termination. In temporal logic notation \( p \) is transient is written as \( \Box \Diamond \neg p \). Note that \textit{false} is transient because \( \textit{false} \Rightarrow \textit{true} \), and, hence \( \neg \textit{false} \) holds whenever \textit{false} holds.

The proof rules are given in Figure 5.1. Below, \( \text{post}_f \) is a local predicate of \( f \) that holds at the termination of \( f \) but nowhere else within \( f \). Such a predicate
always exists, say, by encoding the program termination point into it. For a non-terminating program, \( \text{post}_f \) is false.

- **(Basis)**
  \[
  \{ r \} f \{ s \}
  \]
  For every action \( b \to \alpha \) of \( f \) with precondition \( \text{pre} : \)
  \[
  \text{pre} \land p \Rightarrow b
  \]
  \[
  \{ \text{pre} \land p \} \alpha \{ \neg p \}
  \]

- **(Sequencing)**
  \[
  \{ r \} f \{ \text{transient} \ p \land \neg \text{post}_f \} \text{ post}_f \}
  \]
  \[
  \{ r \} f \{ \text{transient} \ p \} \}
  \]

- **(Concurrency)**
  \[
  \{ r \} f \{ \text{transient} \ p \} \}
  \]

- **(Inheritance)**
  Given: \( (\forall i :: \{ r_i \} f_i \{ s_i \}) \) 
  \[
  \{ r \} f \{ \text{transient} \ p \} \}
  \]
  Assert: \( (\forall i :: \{ r_i \} f_i \{ \text{transient} \ p \} \{ s_i \}) \)

Figure 1: Definition of \text{transient}

**Justifications** 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, even ineffectively, the program has terminated and \( \text{post}_f \) holds. Hence, \( \neg p \lor \text{post}_f \), i.e. \( \neg(p \land \neg \text{post}_f) \), hold eventually in all cases. Note that \( p \) may not continue to hold after the effective execution of an action. Also, if \( \text{pre} \Rightarrow \neg p \) then \( \text{pre} \land p \) is false and the hypotheses are vacuously satisfied.

If \( f \) never terminates, \( \text{post}_f \) is false and then \( \neg p \) 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, \( \alpha; \beta \). To prove \text{transient} \( p \) it is sufficient that \( \alpha \) establish \( \neg p \) or that it execute effectively, thus establishing \( \text{post}_\alpha \), and that \( \beta \) establish \( \neg p \). The sequencing rule generalizes this observation to components. Being a local predicate, \( \text{post}_f \) 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 or preserves \( p \). Since \( f \) is guaranteed to establish \( \neg p \) the result holds.

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 “\( \text{if} \ b \ \text{then} \ f_0 \ \text{else} \ f_1 \)” 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.

\[ \square \]

Notes

1. 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.

2. 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 \( p \ \text{en} \ q \) 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 \( \{ r \} \ f \ \{ p \ \text{en} \ q \} \) and defined as follows:

\[
\{ r \} \ f \ \{ p \wedge \neg q \ \text{co} \ p \vee q, \text{transient} \ p \wedge \neg q \} \quad \{ r \} \ f \ \{ p \ \text{en} \ q \}
\]

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 \text{en} \ are given in Appendix A.3.
**Example: Distributed counter, contd.** We prove a progress property from the annotated program from Section 3.2, reproduced below.

\[
f_j ::
\begin{array}{l}
\text{initially } old_j, new_j = 0, 0 \\
\{ \text{true} \}
\end{array}
\]

\[
\begin{array}{l}
\text{loop} \\
\{ \text{true} \}
\end{array}
\]

\[
\begin{array}{l}
\alpha_j :: new_j := old_j + 1; \\
\{ new_j = old_j + 1 \}
\end{array}
\]

\[
\begin{array}{l}
\text{if } \begin{array}{l}
\beta_j :: \{ new_j = old_j + 1 \} \Rightarrow \text{ctr} := new_j \{ \text{true} \} \\
\gamma_j :: \{ new_j = old_j + 1 \} \Rightarrow \text{ctr} \neq old_j \Rightarrow \text{old}_j := \text{ctr} \{ \text{true} \}
\end{array}
\end{array}
\]

\[
\{ \text{true} \}
\]

\[
\text{forever}
\]

Our ultimate goal is to prove that for any integer \( m \) if \( \text{ctr} = m \) at some point during an execution of \( f_j \), eventually \( \text{ctr} > m \). We will prove this property using a stronger ensures property, as shown.

At any point in the execution of \( f \) let \( nb \) be the number of threads \( f_j \) for which \( \text{ctr} \neq old_j \). We show that every step of \( f_j \) either increases \( \text{ctr} \) or decreases \( nb \) while preserving \( \text{ctr} \)'s value, i.e., for any \( m \) and \( N \), in \( f_j \)

\[
\text{ctr} = m \land nb = N \text{ en } \text{ctr} = m \land nb < N \lor \text{ctr} > m \quad (E)
\]

We use the rules for \( \text{en} \) given in Appendix A.3 (page 17). 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 \( \alpha_j \) always terminates, \( (E) \) needs to be shown only for the \( \text{if} \) statement. From the semantics of \( \text{if} \), to prove \( (E) \) it is sufficient to show that effective execution of each of \( \beta_j \) and \( \gamma_j \) given precondition \( \text{ctr} = m \land nb = N \) establishes \( \text{ctr} = m \land nb < N \lor \text{ctr} > m \) as postcondition, as follows:

\[
\beta_j :: \{ \text{ctr} = m \land nb = N \land new_j = old_j + 1 \land \text{ctr} = old_j \}
\]

\[
\text{ctr} := \text{new}_j \\
\{ \text{ctr} = m \land nb < N \lor \text{ctr} > m \}
\]

\[
\gamma_j :: \{ \text{ctr} = m \land nb = N \land new_j = old_j + 1 \land \text{ctr} \neq old_j \}
\]

\[
\text{old}_j := \text{ctr} \\
\{ \text{ctr} = m \land nb < N \lor \text{ctr} > m \}
\]

The postconditions of \( \beta_j \) and \( \gamma_j \) are, respectively, \( \text{ctr} = m + 1 \) and \( \text{ctr} = m \land nb < N \), thus completing the proof.

Since the ensures property holds in every \( f_j \) it holds in \( f \) as well, using the inheritance rule from Appendix A.3 (page 17).
5.3 Progress Property: Leads-to

Lamport [6] first identified leads-to for concurrent programs as the appropriate generalization of termination for sequential programs. The informal meaning of $p \rightarrow q$ (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) $\frac{p \text{ en } q}{p \rightarrow q}$
- (transitivity) $\frac{p \rightarrow q, q \rightarrow r}{p \rightarrow r}$
- (disjunction) For any (finite or infinite) set of predicates $S$
  
  \[
  \begin{align*}
  \forall p : p \in S : p & \rightarrow q \\
  \lor p : p \in S : p & \rightarrow q
  \end{align*}
  \]

Derived rules for $\rightarrow$ are given in Appendix A.4 (page 19). leads-to is not conjunctive, nor does it obey the inheritance rule, so that even if $p \rightarrow q$ holds in both $f$ and $g$, it may not hold in $f \parallel g$.

**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 A.4.2.

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

\[
\begin{align*}
ctr &= m \land nb = N \quad \text{en} \quad ctr = m \land nb < N \lor ctr > m \\
& \text{proven in Section 5.2} \\
ctr &= m \land nb = N \implies ctr = m \land nb < N \lor ctr > m \\
& \text{Applying the basis rule of leads-to} \\
ctr &= m \implies ctr > m \\
& \text{Induction rule, use the well-founded order} < \text{over natural numbers} \\
true & \implies ctr > C, \text{for any integer } C \\
& \text{Induction rule, use the well-founded order} < \text{over natural numbers.}
\end{align*}
\]

**Compositional aspects of leads-to** The primary progress property, leads-to, does not obey the inheritance rule that holds for all other properties; given $p \rightarrow q$ in both $f$ and $g$, it can not be asserted that $p \rightarrow q$ in $f \parallel g$. Further, even though the inheritance rule for leads-to holds for sequential language constructs, it is often too weak to apply in practice. For example, for a loop while $b$ do $f$ the inheritance rule says that $p \rightarrow q$ in $f$ implies $p \rightarrow q$ for the loop. The requirement $p \rightarrow q$ in $f$ is often too strong to establish. For specific language constructs better rules (with weaker antecedents) can often be constructed. We sketch a rule below for the termination of a loop; note that the loop body may be concurrent components.
Introduce a function of the state that decreases along a well-founded order in each iteration of the loop unless \( q \) is established. For example, the following rule says that execution of the loop with invariant \( I \) terminates if each iteration decreases a metric \( M \) along a well-founded order and terminates, or establishes \( I \land \neg b \), the loop exit condition.

\[
\{ I \land b \land M = m \} \ f \ \{ true \mapsto post_f \land (M \prec m \lor \neg b) \} \ I, \text{ for all } m \\
\{ I \} \ \text{while } b \ \text{do } f \ \{ true \mapsto I \land \neg b \} \ I \land \neg b
\]

### 6 Concluding Remarks, To be done

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.

The non-interference theory of Owicki and Gries, rely-guarantee of Jones, Separation logic of Reynolds and O'Hearn are directly related to the work here. A longer summary of their contributions will be written later.
A Appendix: Derived Rules

A.1 Derived Rules for co and its special cases

 Derived rules for co are given in Figure 2 and for the special cases in Figure A.1. The rules are easy to derive; see Chapter 5 of Misra [7].

\[
\begin{align*}
\text{false} & \ 	ext{co} \ q \\
p \ 	ext{co} \ q, p' & \ 	ext{co} \ q' \quad (\text{CONJUNCTION}) \\
p \land p' & \ 	ext{co} \ q \land q' \\
p \ 	ext{co} \ q \\
p \land p' & \ 	ext{co} \ q \\
p' \ 	ext{co} \ q' \quad (\text{DISJUNCTION}) \\
p \lor p' & \ 	ext{co} \ q \lor q' \\
p \ 	ext{co} \ q \\
p & \ 	ext{co} \ q \\
(p \land p') & \ 	ext{co} \ q \land q' \\
(p \lor p') & \ 	ext{co} \ q \lor q' \\
\end{align*}
\]

Figure 2: Derived rules for co

The top two rules in Figure 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 [3] 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.

- (stable conjunction, stable disjunction)
  \[
  p \ 	ext{co} \ q, \ 	ext{stable} \ r \\
p \land r & \ 	ext{co} \ q \land r \\
p \lor r & \ 	ext{co} \ q \lor r
  \]

- (Special case of the above) stable p, stable q
  \[
  \text{stable} \ p \land q \\
  \text{stable} \ p \lor q
  \]

- invariant p, invariant q
  \[
  \text{invariant} \ p \land q \\
  \text{invariant} \ p \lor q
  \]

- \{r\} f \ \{stable \ p\} \\
  \{r \land p\} f \ \{p\} \\
- \{r\} f \ \{constant \ c\} \\
  \{r \land c = c\} f \ \{c = c\}

- (constant formation) Any expression built out of constants is constant.

Figure 3: Derived rules for the special cases of co
A.2 Derived Rules for transient

- **transient false.**

- (Strengthening) Given transient \( p \), transient \( p \land q \) for any \( q \).

To prove transient \( false \) use the basis rule. The proof of the next 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 rules 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 \( p' \land \neg post_f \) for some \( p' \). Then in some annotation of \( f \) where action \( b \rightarrow \alpha \) has the precondition \( pre \):

   (1) \( pre \land p' \Rightarrow b \), and (2) \( \{ pre \land p' \} \alpha \{ \neg p' \} \).

   (1') From predicate calculus \( pre \land p' \land q \Rightarrow b \), and

   (2') from Hoare logic \( \{ pre \land p' \land q \} \alpha \{ \neg p' \} \).

   Applying the basis rule, transient \( p' \land q \land \neg post_f \), i.e., transient \( p \land q \).

2. (Sequencing) In \( f ; g \), transient \( p \land \neg post_f \) in \( f \) and transient \( p \) in \( g \). Inductively, transient \( p \land q \land \neg post_f \) in \( f \) and transient \( p \land q \) in \( g \). Applying the sequencing rule, transient \( p \land q \).

3. (Concurrency, Inheritance) Similar proofs.

A.3 Derived Rules for en

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)

\[
\begin{align*}
\{ r \} f \{ s \} \\
\text{For every action } b \rightarrow \alpha \text{ with precondition } pre \text{ in the annotation :} \\
pre \land p \land \neg q \Rightarrow b \\
\{ pre \land p \land \neg q \} \alpha \{ q \} \\
\{ r \} f \{ p \text{ en } p \land q \lor q \mid s \}
\end{align*}
\]

- (Sequencing)

\[
\begin{align*}
\{ r \} f \{ p \text{ en } p \land \text{ post}_f \lor q \mid \text{ post}_f \} \\
\{ \text{ post}_f \} g \{ p \text{ en } q \mid \} \\
\{ r \} f ; g \{ p \text{ en } q \mid \}
\end{align*}
\]

- (Concurrency)

\[
\begin{align*}
p \text{ en } q \text{ in } f \\
p \land \neg q \text{ co } p \lor q \text{ in } g \\
p \text{ en } q \text{ in } f \parallel g
\end{align*}
\]
• (Inheritance) Assuming the basic proof rule at left the inheritance proof rule at right can be asserted.

Given: \( \forall i :: \{r_i\} f_i \{s_i\} \)

Assert: \( \forall i :: \{r_i\} f_i \{p \text{ en } q | s_i\} \)

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 \textit{transient} and \textit{co}. We show one such proof, for the PSP rule. Observe that ensures is only partially conjunctive and not disjunctive, unlike \textit{co}.

1. (implication) \( \frac{p \Rightarrow q}{p \text{ en } q} \)

   Consequently, \( \text{false en } q \) and \( p \text{ en } \text{true} \) for any \( p \) and \( q \).

2. (rhs weakening) \( \frac{p \text{ en } q}{p \text{ en } q \lor q'} \)

3. (partial conjunction) \( \frac{p \text{ en } q}{p' \text{ en } q} \) \( \frac{p \land p' \text{ en } q}{p \text{ en } q} \)

4. (lhs manipulation) \( \frac{p \land \neg q \Rightarrow p' \Rightarrow p \lor q}{p \text{ en } q \equiv p' \text{ en } q} \)

   Observe that \( p \land \neg q \equiv p' \land \neg q \) and \( p \lor q \equiv p' \lor q \). So, \( p \) and \( q \) are interchangeable in all the proof rules. As special cases, \( p \land \neg q \text{ en } q \equiv p \text{ en } q \equiv p \lor q \text{ en } q \).

5. (PSP) The general rule is at left, and a special case at right using \textit{stable} \( r \) as \( r \text{ co } r \).

   \[
   \begin{align*}
   \text{(PSP)} & \quad p \text{ en } q \\
   r \text{ co } s & \quad p \text{ en } q \\
   p \land r \text{ en } (q \land r) \lor (\neg r \land s) & \quad p \land r \text{ en } q \land r
   \end{align*}
   \]

6. (Special case of Concurrency) \( \frac{p \text{ en } q \text{ in } f \text{ stable } p \text{ in } g}{p \text{ en } q \text{ in } f \parallel g} \)

   Proof of (PSP): From the hypotheses:

   \begin{align*}
   \text{transient} & \quad p \land \neg q \quad (1) \\
   p \land \neg q & \quad \text{co } p \land q \quad (2) \\
   r & \quad \text{co } s \quad (3)
   \end{align*}

   We have to show:
transient \( p \wedge r \wedge \neg(q \wedge r) \wedge \neg(r \wedge s) \) \hspace{1cm} (4)
\( p \wedge r \wedge \neg(q \wedge r) \wedge \neg(r \wedge s) \mathrm{co} \ p \wedge r \vee q \wedge r \vee \neg r \wedge s \) \hspace{1cm} (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:

\[
\begin{align*}
    p \wedge r \wedge \neg q & \quad \mathrm{co} \quad p \wedge s \vee q \wedge s \\
    \equiv & \quad \{ \text{expand both terms in rhs} \} \\
    p \wedge r \wedge \neg q & \quad \mathrm{co} \quad p \wedge r \wedge s \vee p \wedge \neg r \wedge s \vee q \wedge r \wedge s \vee q \wedge \neg r \wedge s \\
    \Rightarrow & \quad \{ \text{lhs strengthening and rhs weakening} \} \\
    p \wedge r \wedge \neg q \wedge s & \quad \mathrm{co} \quad p \wedge r \vee q \wedge r \vee \neg r \wedge s
\end{align*}
\]

A.4 Derived Rules for leads-to

The rules are taken from Misra [7] 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.

A.4.1 Lightweight rules
1. (implication) \( p \Rightarrow q \)
2. (lhs strengthening, rhs weakening)
   \[
   \frac{p \Rightarrow q}{\frac{p' \wedge p \Rightarrow q}{p \Rightarrow q \vee q'}}
   \]
3. (disjunction)
   \[
   \left( \forall i :: p_i \Rightarrow q_i \right) \Rightarrow \left( \forall i :: p_i \Rightarrow q_i \right)
   \]
   where \( i \) is quantified over an arbitrary finite or infinite index set, and \( p_i, q_i \) are predicates.
4. (cancellation) \( p \Rightarrow q \vee r \quad r \Rightarrow s \)
   \[
   \frac{p \Rightarrow q \vee r}{p \Rightarrow q \vee s}
   \]

A.4.2 Heavyweight rules
1. (impossibility) \( p \Rightarrow \mathrm{false} \)
   \[
   \frac{p \Rightarrow q}{\neg p}
   \]
2. (PSP) \( p \wedge p' \Rightarrow q \wedge p' \)
   \[
   p \Rightarrow q
   \]
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 $W$.

$$\forall m :: p \land M = m \implies (p \land M \prec m) \lor q$$

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

$$\forall i :: p_i \implies q_i \lor b$$

$$\forall i :: q_i \lor b \lor q_i$$

A.4.3 Lifting Rule

This rule permits lifting a leads-to property of $f$ to $f \parallel g$ 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 the proof of $p \implies q$ in $f$.

$$p \implies q \text{ in } f$$

$$p \land x = X \implies q \lor x \neq X \text{ in } f \parallel g$$

In particular, if $f$ has no shared variable then

$$p \implies q \text{ in } f$$

$$p \implies q \text{ in } f \parallel g$$

An informal justification of this rule is as follows. Consider any execution of $f \parallel g$ starting in a state where $p \land x = X$ holds. If $x$ is modified during the execution then $x \neq X$ is established. If $x$ is never modified then $g$ does not modify any accessible variable of $f$; so, $f$ is oblivious to the presence of $g$, and it establishes $q$.

The formal proof is by induction on the structure of the proof of $p \implies q$ in $f$. For the base case, given that $p \text{ en } q$ in $f$ show that $p \land x = X \text{ en } q \lor x \neq X$ in $f \parallel g$. See Section 8.2.5 in Misra [7] for a formal justification of a related result. The other cases are easy to prove by induction.

B Example: Mutual exclusion

We prove a coarse-grained version of a 2-process mutual exclusion program due to Peterson [9]. This program is an instance of a tightly-coupled system where, typically, the codes of all the components have to be considered together to construct a proof. In contrast, we construct a composable specification of each
component (in fact, the codes of the components are symmetric, and so are their specifications), and combine the specifications to derive a proof.

The program has two processes $M$ and $M'$. Process $M$ has two local boolean variables, $\text{try}$ and $\text{cs}$ where $\text{try}$ remains $true$ as long as $M$ is attempting to enter the critical section or in it and $\text{cs}$ is $true$ as long as it is in the critical section; $M'$ has $\text{try}'$ and $\text{cs}'$. They both have access to a non-local boolean variable $\text{turn}$.

The codes for $M$ and $M'$ are almost duals of each other in the sense that every local variable in $M$ is correspondingly replaced by its primed counterpart in $M'$. Variable $\text{turn}$ is set $true$ in $M$ and tested for value $false$, and in $M'$ in exactly the opposite manner. To obtain exact dual code for $M$ and $M'$, rewrite $\text{turn} := \text{false}$ in $M'$ as $\text{turn}' := \text{true}$; similarly the test for $\text{turn}$ is replaced by the dual test for $\text{turn}'$. Regard $\text{turn}'$ as the complement of $\text{turn}$. Then the codes of $M$ and $M'$ are exact duals in the sense that the code structure is identical and every unprimed variable in one is primed in the other, and vice versa. Therefore, for any property of $M$ its dual is a property of $M'$. And, any property of $M \[] M'$ yields its dual also as a property, thus reducing the proof length by around half.

B.1 Program

The global initialization and the code for $M$, along with a local annotation, is given below. The code for $M'$ is not shown because it is just the dual of $M$. 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 each iteration.

initially $\text{cs, cs}' = false, false$ — global initialization

$M$: initially $\text{try} = false$

\{
\lnot \text{try}, \lnot \text{cs}\}

loop

— unrelated computation that may not terminate;

\{
\lnot \text{try}, \lnot \text{cs}\} \quad \alpha: \quad \text{try, turn} := \text{true}, \text{true};

\{\text{try, } \lnot \text{cs}\} \quad \beta: \quad \lnot \text{try}' \lor \text{turn}' \rightarrow \text{cs} := \text{true}; — \text{Enter critical section}

\{\text{try, cs}\} \quad \gamma: \quad \text{try, cs} := \text{false}, \text{false} — \text{Exit critical section}

forever

Remarks on the program The given program is based on a simplification of an algorithm in Peterson [9]. In the original version the assignment in $\alpha$ may be decoupled to the sequence $\text{try} := \text{true}; \text{turn} := \text{true}$. The tests in $\beta$ may be made separately for each disjunct in arbitrary order. Action $\gamma$ may be written in sequential order $\text{try} := \text{false}; \text{cs} := \text{false}$. These changes can be
easily accommodated within our proof theory by introducing auxiliary variables to record the program control.

B.2 Safety and progress properties

It is required to show: (1) the safety property that both $M$ and $M'$ are not within their critical sections simultaneously, i.e., invariant $\neg(cs \land cs')$, and (2) a progress property that any process attempting to enter its critical section will succeed eventually, i.e., $try \leftrightarrow cs$ and $try' \leftrightarrow cs'$. We prove just $try \leftrightarrow cs$ since its dual also holds.

B.2.1 Safety proof: invariant $\neg(cs \land cs')$

We prove:

\begin{align*}
invariant & \; cs \Rightarrow try \text{ in } M \parallel M' \tag{S1} \\
invariant & \; cs' \land try \Rightarrow turn \text{ in } M \parallel M' \tag{S2}
\end{align*}

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

\[
\begin{align*}
& cs \land cs' \\
\Rightarrow & \; \{\text{from (S1) and its dual}\}
& cs \land try' \land cs' \land try \\
\Rightarrow & \; \{\text{from (S2) and its dual}\}
& turn \land turn' \\
\Rightarrow & \; \{turn \text{ and } turn' \text{ are complements}\}
& false
\end{align*}
\]

Proofs of (S1) and (S2) The predicates in (S1) and (S2) hold initially because initially $cs, cs' = false, false$. Next, we show these predicates to be stable. The proof is compositional, by proving each to be stable in $M$ and $M'$. The proof is simplified by duality of their codes. First, we show the following stable properties of $M$ from which we deduce (S1) and (S2):

\begin{align*}
stable & \; cs \Rightarrow try \text{ in } M \tag{S3} \\
stable & \; try \Rightarrow turn \text{ in } M \tag{S4} \\
stable & \; cs \land try' \Rightarrow turn' \text{ in } M \tag{S5}
\end{align*}

Proofs of (S3), (S4) and (S5) The proofs are entirely straight-forward, but each property has to be shown stable for each action. We show the proofs in Table 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 predicate is stable, or (2) “unaff.” meaning that the variables in the predicate are unaffected by this action execution. The only entry that is left out is for the case that $\beta$ preserves $cs \land try' \Rightarrow turn'$; this is shown as follows. The guard of $\beta$ can be written as $try' \Rightarrow turn'$ and execution of $\beta$ affects neither $try'$ nor $turn'$, so $try' \Rightarrow turn'$ is a postcondition, and so is $cs \land try' \Rightarrow turn'$.
Proofs of (S1) and (S2) from (S3), (S4) and (S5): As we have remarked earlier, the predicates in (S1) and (S2) hold initially, so we prove only their stability.

- **(S1)** \textbf{stable} cs ⇒ try in \(M \parallel M'\):
  \begin{itemize}
  \item stable cs ⇒ try in \(M\), shown as (S3)
  \item stable cs ⇒ try in \(M'\), \(cs\) and try are constant in \(M'\)
  \item stable cs ⇒ try in \(M \parallel M'\), Inheritance rule
  \end{itemize}

- **(S2)** stable \(cs' \land \text{try} \Rightarrow \text{turn}\) in \(M \parallel M'\):
  \begin{itemize}
  \item stable try ⇒ turn in \(M\), shown as (S4)
  \item stable \(cs' \land \text{try} \Rightarrow \text{turn}\) in \(M\), \(cs'\) constant in \(M\)
  \item stable \(cs' \land \text{try} \Rightarrow \text{turn}\) in \(M'\), dual of (S5)
  \item stable \(cs' \land \text{try} \Rightarrow \text{turn}\) in \(M \parallel M'\), Inheritance rule
  \end{itemize}

B.2.2 Progress proof: try ⇒⇒ cs in \(M \parallel M'\)

The proof is based on one safety and one ensures properties.

\begin{align*}
\text{try} \land \neg cs \co cs & \text{ in } M \parallel M' \quad \text{(S6)} \\
\text{try} \land (\neg \text{try}' \lor \text{turn}') & \en \neg \text{try} \text{ in } M \parallel M' \quad \text{(P1)}
\end{align*}

Proofs of (S6) and (P1): To prove (S6): prove \(\text{try} \land \neg cs \co cs\) in \(M\), which is entirely straight-forward, and that the property, being local to \(M\), is unaffected by \(M'\). Using inheritance, the result follows.

To prove (P1): we first show

\begin{align*}
\text{try} \land (\neg \text{try}' \lor \text{turn}') & \en \neg \text{try} \text{ in } M \quad \text{(P1.1)} \\
\text{stable} \neg \text{try} \lor \text{turn} & \text{ in } M \quad \text{(P1.2)}
\end{align*}

The proof of (P1.1) uses the sequencing rule and (P1.2) is straight-forward. Then, using duality on (P1.2), \textbf{stable} \(\neg \text{try}' \lor \text{turn}'\) in \(M'\). Next, using the concurrency rule for ensures with (P1.1) the desired result, (P1), holds.
Proof of $\text{try} \mapsto cs$ in $M \parallel M'$: All the properties below are in $M \parallel M'$.

Get (P2) from (P1), and (P2.1) and (P2.2) from (P2) using lhs strengthening.

- $\text{try} \land (\neg \text{try}' \lor \text{turn}') \mapsto \neg \text{try}$ (P2)
- $\text{try} \land \neg \text{try}' \mapsto \neg \text{try}$ (P2.1)
- $\text{try} \land \text{turn}' \mapsto \neg \text{try}$ (P2.2)

The main proof:

- $\text{try} \land \text{turn}' \mapsto \neg \text{try}$, from (P2.2)
- $\text{try}' \land \text{turn} \mapsto \neg \text{try}'$, duality
- $\text{try} \land \text{try}' \land \text{turn} \mapsto \neg \text{try} \lor \text{try} \land \neg \text{try}'$, lhs strengthening and rhs weakening
- $\text{try} \land \text{try}' \land \text{turn} \mapsto \neg \text{try}$, use (P2.1) in rhs
- $\text{try} \land \neg \text{cs} \land \text{cs}$, (S6)
- $\text{try} \land \neg \text{cs} \mapsto \text{cs}$, PSP of above two
- $\text{try} \mapsto \text{cs}$, disjunction with $\text{try} \land \text{cs} \mapsto \text{cs}$

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 $f_1 \parallel f_k$. 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.

Problem Description Given is a bag $u$ on which $\oplus$ is a commutative and associative binary operation. Define $\Sigma u$, fold of $u$, to be the result of applying $\oplus$ to all the elements of $u$. Henceforth, $|u|$ denotes the size of $u$.

It is required to replace all the elements of $u$ by $\Sigma u$. 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 $u_0$ to $u$ such that: (1) $\Sigma u_0 = \Sigma u$, and (2) $|u| = |u_0| - k$, provided $|u_0| > k$. Therefore, execution of $f_{n-1}$, where $n$ is the size of $u_0$ and $n > 1$, computes a single value in $u$ that is the fold of the initial bag.

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

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

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

- $f_1 = \ \text{if } |u| > 0 \ \text{then } \text{get}(x) \ \text{else } \text{get}(y) \ \text{end}$
- $f_{k+1} = \ f_k \parallel f_k, \ \text{if } k \geq 1$
Auxiliary variables  To construct a specification introduce the following local auxiliary variables of \( f_k \):

1. \( w_k \): the bag of items removed from \( u \) that are, as yet, unfolded. That is, every \( \text{get} \) from \( u \) puts a copy of the item in \( w_k \), and \( \text{put}(x \oplus y) \) removes \( x \) and \( y \) from \( w_k \). Initially \( w_k = \{ \} \).

2. \( np_k \): the number of completed \( \text{put} \) operations. A \( \text{get} \) does not affect \( np_k \) and a \( \text{put} \) increments it. Initially \( np_k = 0 \).

The modified program is as follows where \( \langle \cdots \rangle \) is a command.

\[
\begin{align*}
\text{f}_1 &= |u| > 0 \rightarrow \langle \text{get}(x); w_1 := w_1 \cup \{x\} \rangle; \\
|u| > 0 & \rightarrow \langle \text{get}(y); w_1 := w_1 \cup \{y\} \rangle; \\
\langle \text{put}(x \oplus y); w_1 := w_1 - \{x, y\}; np_1 := 1 \rangle \\
\text{f}_{k+1} &= \text{f}_1 \sqcap \text{f}_k, \quad k \geq 1
\end{align*}
\]

Note that \( w_{k+1} = w_1 \cup w_k \) and \( np_{k+1} = np_1 + np_k \), for all \( i \) and \( j \).

Specification of \( f_k \)  These auxiliary variables are not adequate to state that the program terminates. We can introduce yet another auxiliary boolean variable \( h_k \) for this purpose as follows: (1) \( h_1 \) is initially false, and becomes true at the completion of \( f_1 \), and (2) \( h_{k+1} = h_1 \land h_k \). Then show that the initial conditions leads-to \( h_k \), for all \( k \). It can be shown that \( h_k \equiv np_k = k \). Therefore, we do not introduce \( h_k \) as a variable, instead, the progress property (P) is written using \( np_k \).

We assert, for all \( k, k \geq 1 \):

\[
\begin{align*}
\{u = u_0\} & \text{f}_k \{ \Sigma u = \Sigma u_0, |u| + k = |u_0| \} \quad (T) \\
|u| + |w_k| + np_k > k & \rightarrow np_k = k \text{ in } f_k \quad (P)
\end{align*}
\]

The terminal property (T) says that \( f_k \) transforms \( u \) in the expected manner. The progress property (P) says that execution of \( f_k \) terminates (\( np_k = k \)) if \( |u| + |w_k| + np_k > k \) holds at any moment during its execution; since initially \( |w_k| + np_k = 0 \), this amounts to requiring that initially \( |u| > k \). Without this condition the program becomes deadlocked. We prove (T) in Section C.1, and (P) in Section C.2.

C.1 Proof of terminal property (T)

Property (T) can not be proved from a local annotation alone because it names the shared variable \( u \) in its pre- and postconditions. So, we prove a number of perpetual properties and then use some of the meta-rules, from Section 4.3, to establish (T). First, we show for all \( k, k \geq 1 \):

\[
\begin{align*}
\{w_k = \{\}, np_k = 0\} & \text{f}_k \\
\{\text{constant } \Sigma(u \cup w_k), \text{ constant } |u| + |w_k| + np_k \mid w_k = \{\}, np_k = k \} & (T')
\end{align*}
\]
We use the following abbreviations in the proof.

\[ a_k \equiv w_k = \{\} \land np_k = 0 \]
\[ b_k \equiv \text{constant } \Sigma(u \cup w_k), \text{ constant } |u| + |w_k| + np_k \]
\[ c_k \equiv w_k = \{\} \land np_k = k \]

So, we need to show \( \{a_k\} f_k \{b_k | c_k\} \). Proof is by induction on \( k \).

- Proof for \( f_1 \): Consider the following local annotation of \( f_1 \).

\[
\begin{align*}
\{w_1 = \{\}, np_1 = 0\} \\
|u| > 0 & \rightarrow \langle \text{get}(x); \ w_1 := w_1 \cup \{x\}\rangle; \\
\{w_1 = \{x\}, np_1 = 0\} \\
|u| > 0 & \rightarrow \langle \text{get}(y); \ w_1 := w_1 \cup \{y\}\rangle; \\
\{w_1 = \{x, y\}, np_1 = 0\} \\
\langle \text{put}(x \oplus y); \ w_1 := w_1 - \{x, y\}; \ np_1 := 1\rangle \\
\{w_1 = \{\}, np_1 = 1\}
\end{align*}
\]

The perpetual and terminal properties of \( f_1 \) in \( (T') \) are easily shown using this annotation.

- Proof for \( f_{k+1} \):

\[
\begin{align*}
\{a_1\} f_1 \{b_1 | c_1\} & \quad , \text{from the annotation of } f_1 \\
\{a_1\} f_1 \{b_{k+1} | c_1\} & \quad , \text{ } w_k, np_k \text{ constant in } f_1 \quad (1) \\
\{a_k\} f_k \{b_k | c_k\} & \quad , \text{ inductive hypothesis} \\
\{a_k\} f_k \{b_{k+1} | c_k\} & \quad , \text{ } w_1, np_1 \text{ constant in } f_k \quad (2) \\
\{a_1 \land a_k\} f_1 \{c_1 \land c_k\} & \quad , \text{ join proof rule on } (1,2) \\
\{a_1 \land a_k\} f_1 \{c_1 \land c_k\} & \quad , \text{ inheritance on } (1,2) \\
\{a_{k+1}\} f_{k+1} \{b_{k+1} | c_{k+1}\} & \quad , \text{ } a_{k+1} = a_1 \land a_k \text{ and } c_{k+1} = c_1 \land c_k
\end{align*}
\]

**Proof of terminal property \( (T) \)** Using \( (T') \) and the precondition \( u = u_0 \) we show the terminal properties of \( (T) \), that \( \Sigma u = \Sigma u_0 \) and \( |u| + k = |u_0| \) are postconditions of program \( f_k \), for all \( k, k \geq 1 \).

\[
\begin{align*}
\{a_k\} f_k \{b_k | c_k\} & \quad , \text{proved above as } (T') \\
\{u = u_0, a_k\} f_k \{b_k | c_k\} & \quad , \text{ lhs strengthening from Section 4.3} \\
\{u = u_0, a_k\} f_k \{\text{invariant } \Sigma(u \cup w_k) = \Sigma u_0, \text{ invariant } |u| + |w_k| + np_k = |u_0| \land |c_k\} \\
\{u = u_0, a_k\} f_k \{\Sigma(u \cup w_k) = \Sigma u_0, \ |u| + |w_k| + np_k = |u_0| \land c_k\} \\
\{u = u_0, a_k\} f_k \{u = \Sigma u_0, \ |u| + k = |u_0| \land c_k\} \\
\{u = u_0, w_k = \{\} \land np_k = 0\} f_k \{u = \Sigma u_0, \ |u| + k = |u_0| \} \\
\end{align*}
\]
C.2 Proof of Progress property (P)

Property (P) is: \(|u| + |w_k| + np_k > k \implies np_k = k\) in \(f_k\). Abbreviate \(|u| + |w_k| + np_k > k\) by \(p_k\) and \(np_k = k\) by \(q_k\) so that for all \(k, k \geq 1:\)

\[ p_k \mapsto q_k \text{ in } f_k \]

We have shown in Section C.1 that \(p_k\) is constant, hence stable, in \(f_k\). The proof of \(p_k \mapsto q_k\) is by induction on \(k\), as shown in Sections C.2.1 and C.2.2.

C.2.1 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.

\[
\begin{align*}
\{w_1 = \{\}, np_1 = 0\} \\
|u| > 0 & \implies \langle get(x); w_1 := w_1 \cup \{x\}\rangle; \\
\{w_1 = \{x\}, np_1 = 0\} \\
|u| > 0 & \implies \langle get(y); w_1 := w_1 \cup \{y\}\rangle; \\
\{w_1 = \{x, y\}, np_1 = 0\} \\
\langle put(x \oplus y); w_1 := w_1 - \{x, y\}; np_1 := 1\rangle \\
\{w_1 = \{\}, np_1 = 1\}
\end{align*}
\]

Next, prove \(p_1\) en \(q_1\) using the sequencing rule of en, from Section 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 afterwards. From the annotation \(q_1\) holds as a postcondition. Further, as shown in Section C.1, \(p_1\) is stable. Therefore, it suffices to show that if \(p_1\) holds initially then every action is effectively executed.

Using the given annotation, the verification conditions for the two \(get\) actions, that are easily proved, are (expanding \(p_1\)):

\[
\begin{align*}
w_1 = \{\} \land np_1 = 0 \land |u| \land |w_1| + np_1 > 1 & \implies |u| > 0, \text{ and} \\
w_1 = \{x\} \land np_1 = 0 \land |u| \land |w_1| + np_1 > 1 & \implies |u| > 0
\end{align*}
\]

C.2.2 Progress proof, \(p_{k+1} \mapsto q_{k+1}\) in \(f_{k+1}\)

First, we prove a safety result in \(f_k\), for all \(k\). Let \(z_k\) denote the list \((u, w_k)\). Define a well-founded order relation \(<\) over pairs of sets such as \(z_k\): \((a, b) < (a', b')\) if \(|a| + |b|, |a|\) is lexicographically smaller than \(|a'| + |b'|, |a'|\).

We assert that for all \(k\), stable \(z_k \preceq n\) in \(f_k\). 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 \(w_k\) 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, \( p_{k+1} \rightarrow q_{k+1} \) 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, \( p_{k+1} \rightarrow q_{k+1} \), follows by using transitivity on (P1) and (P2).

P1. \( p_{k+1} \rightarrow p_{k+1} \land (q_1 \lor q_k) \) in \( f_{k+1} \)

P2. \( p_{k+1} \land (q_1 \lor q_k) \rightarrow q_{k+1} \) in \( f_{k+1} \)

We have \( p_1 \rightarrow q_1 \) and, inductively, \( p_k \rightarrow q_k \). The proofs mostly use the derived rules of leads-to from Section 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 \( p_{k+1} \Rightarrow (p_1 \lor p_k), p_{k+1} \land q_1 \Rightarrow p_k, p_{k+1} \land q_k \Rightarrow p_1 \) and \( q_1 \land q_k \Rightarrow q_{k+1} \).

**Proof of (P1)** \( p_{k+1} \rightarrow p_{k+1} \land (q_1 \lor q_k) \) in \( f_{k+1} \): Below all properties are in \( f_{k+1} \).

\[
\begin{align*}
p_1 \land z_{k+1} = n &\rightarrow q_1 \lor z_{k+1} \neq n, \text{ lifting on } p_1 \rightarrow q_1 \text{ in } f_1 \\
p_k \land z_{k+1} = n &\rightarrow q_k \lor z_{k+1} \neq n, \text{ lifting on } p_k \rightarrow q_k \text{ in } f_k \\
(p_1 \lor p_k) \land z_{k+1} = n &\rightarrow (q_1 \lor q_k) \lor z_{k+1} \neq n, \text{ disjunction}
\end{align*}
\]

\( (p_1 \lor p_k) \land z_{k+1} = n \rightarrow (q_1 \lor q_k) \lor z_{k+1} \prec n \), (PSP) with stable \( z_{k+1} \leq n \)

\( p_{k+1} \land (p_1 \lor p_k) \land z_{k+1} = n \rightarrow p_{k+1} \land (q_1 \lor q_k) \lor p_{k+1} \land z_{k+1} \prec n \), conjunction with stable \( p_{k+1} \)

\( p_{k+1} \land z_{k+1} = n \rightarrow p_{k+1} \land z_{k+1} \prec n \lor p_{k+1} \land (q_1 \lor q_k) \), \( p_{k+1} \Rightarrow (p_1 \lor p_k) \)

\( p_{k+1} \rightarrow p_{k+1} \land (q_1 \lor q_k) \), induction rule of leads-to

**Proof of (P2)** \( p_{k+1} \land (q_1 \lor q_k) \rightarrow q_{k+1} \) in \( f_{k+1} \): Below all properties are in \( f_{k+1} \).

\[
\begin{align*}
p_1 \land z_{k+1} = n &\rightarrow q_1 \lor z_{k+1} \neq n, \text{ lifting on } p_1 \rightarrow q_1 \text{ in } f_1 \\
p_1 \land z_{k+1} = n &\rightarrow q_1 \land z_{k+1} \prec n, \text{ conjunction with stable } z_{k+1} \leq n \\
q_k \land p_1 \land z_{k+1} = n &\rightarrow q_k \land q_1 \lor q_k \land z_{k+1} \prec n, \text{ conjunction with stable } q_k
\end{align*}
\]

\( p_{k+1} \land q_k \land p_1 \land z_{k+1} = n \rightarrow p_{k+1} \land q_k \land q_1 \lor p_{k+1} \land q_k \land z_{k+1} \prec n \), conjunction with stable \( p_{k+1} \)

\( p_{k+1} \land q_k \land z_{k+1} = n \rightarrow p_{k+1} \land q_k \land z_{k+1} \prec n \lor p_{k+1} \land q_k \land q_1 \), \( p_{k+1} \land q_k \Rightarrow p_1 \)

\( p_{k+1} \land q_k \rightarrow p_{k+1} \land q_1 \land q_k \), induction rule of leads-to

\( p_{k+1} \land q_k \rightarrow q_1 \land q_k \), rhs weakening

\( p_{k+1} \land q_1 \rightarrow q_1 \land q_k \), similarly

\( p_{k+1} \land (q_1 \lor q_k) \rightarrow q_{k+1} \), disjunction and \( q_1 \land q_k \Rightarrow q_{k+1} \)
References


