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

Jayadev Misra

March 8, 2016

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 ................................. 5

3 Introduction to the proof theory 6
  3.1 Specification .................................................. 6
  3.2 Local Annotation .............................................. 7
  3.3 Example: Distributed Counter, contd. ....................... 8
  3.4 Meta-rules .................................................... 8

4 Safety Properties 8
  4.1 Safety Property $\mathbf{co}$ ..................................... 9
  4.2 Special cases of $\mathbf{co}$ .................................... 9
  4.3 Meta-rules .................................................... 9
  4.4 Example: Distributed Counter, contd. ....................... 10

5 Progress Properties 10
  5.1 Progress Property: transient ................................. 10
  5.2 Progress Property: ensures .................................. 13
  5.3 Progress Property: Leads-to ................................. 14

6 Related Work 15
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 unscaleable. 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 [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 [8] and Owicki and Lamport [12].

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 [9]. 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 throughout the paper. 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 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.

\[
\begin{align*}
\text{action} & ::= \text{guard} \rightarrow \text{body} \\
f, g ::= \text{component} & ::= \text{action} \mid f \parallel g \mid \text{seq}(f_0, f_1, \ldots, f_n) \\
\text{program} & ::= f
\end{align*}
\]

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 \texttt{true} is often dropped.

An action without a guard is \texttt{non-blocking} and a guarded action \texttt{blocking}.

A structured component is either: (1) a \texttt{join} of the form \( f \parallel g \) where \( f \) and \( g \) are its \texttt{direct subcomponents}, or (2) \texttt{seq}(\( f_0 \), \( f_1 \), \ldots, \( f_n \)) 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 \texttt{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 \texttt{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, \( (((f \parallel g) ; (f' \parallel g')) \) is a program with \( f \), \( f' \), \( g \) and \( g' \) as components.

A \texttt{program} is a component that is meant to be executed alone.

Access rights to variables A variable named in a component is \texttt{accessible} to it. Variable \( x \) is \texttt{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 \( f \parallel g \) 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 $\alpha$ 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 $\alpha$ 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 [4] 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 [9].

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

\[
\text{initially } \quad ctr = 0 \\
\text{\( f_j :: \)} \\
\text{\quad initially } \quad old_j, new_j = 0, 0 \\
\text{\quad loop} \\
\text{\quad \quad new_j := old_j + 1;} \\
\text{\quad \quad if } \begin{cases} 
\text{\quad \quad \quad ctr = old_j \rightarrow ctr := new_j} \\
\text{\quad \quad \quad ctr \neq old_j \rightarrow old_j := ctr}
\end{cases} \\
\text{\quad \quad 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, its end state is \( s \), 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:

$$\{p \land b\} \alpha \{q\}$$

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

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 $\text{seq}$ 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.

$$\{r\} g \{s\}$$

$$\{r'\} h \{s'\}$$

$$\{r \land r'\} g || h \{s \land s'\}$$

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.

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

3.4 Meta-rules

The following general rules apply for specifications.

- (lhs strengthening; rhs weakening)

\[
\frac{r' \Rightarrow r, \ s \Rightarrow s', \ Q' \subseteq Q, \ r' \text{ and } s' \text{ are local to } f}{\{r'\} f \{Q' | s'\}}
\]

- (Conjunction; Disjunction)

\[
\begin{align*}
\{r\} f \{Q | s\} & \\
\{r'\} f \{Q' | s'\} & \\
\{r \land r'\} f \{Q \cup Q' | s \land s'\} & \\
\{r \lor r'\} f \{Q \cap Q' | s \lor s'\}
\end{align*}
\]

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 $r \land r'$-executions is the intersection of $r$-executions and $r'$-executions. Therefore, the postcondition of any execution in $r \land r'$-executions satisfies $s \land s'$ 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, $\text{co}$, and its special cases, taken from Misra [9]. Safety properties are derived from local annota-
tions 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.

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

As an example, the statement “every change in integer variable \( ctr \) can only increment its value” may be formalized as \( ctr = m \co ctr = m \lor ctr = m + 1 \), 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 \( f \)

\[
\begin{align*}
\text{stable } p & \equiv p \co p \\
\text{constant } e & \equiv (\forall c :: \text{stable } e = c) \\
\text{invariant } p & \equiv \text{initially } p \text{ and stable } p
\end{align*}
\]

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 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 \( \sigma \),

9
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.

**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 \( \sigma \).

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 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., \( ctr = m \textbf{ co } ctr = m \lor ctr = m + 1 \) in \( f \). Using the inheritance rule, it is sufficient to prove this property in every component \( f_j \). In \( f_j \), only \( \beta_j \) may change the value of \( ctr \); so we need only show the following whose proof is immediate:

\[
\{ ctr = m \land new_j = old_j + 1 \land ctr = old_j \} \Rightarrow new_j \{ ctr = m \lor ctr = m + 1 \}
\]

### 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 \textit{leads-to}, is defined in Section 5.3 (page 14). First, we introduce two simpler progress properties, \textit{transient} and \textit{ensures}. 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 an execution of \( f \), \( \neg p \) holds then or eventually in that execution. In temporal logic notation \( p \) is transient is written as \( \square \diamond \neg p \). Note that \( \neg p \) may not continue to
hold after \( p \) has been falsified. Predicate \( \textit{false} \) is transient because \( \textit{false} \Rightarrow \textit{true} \), and, hence \( \neg\textit{false} \) holds whenever \( \textit{false} \) 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, \( \textit{post}_f \) is a local predicate of \( f \) that holds on the termination of \( f \) but nowhere else within \( f \). Such a predicate always exists, say, by encoding the termination control point into it.

For a non-terminating program, \( \textit{post}_f \) is \( \textit{false} \).

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

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

- **(Concurrency)**
  \[
  \{r\} f \{\textit{transient} \ p\} \\
  \{r\} f || g \{\textit{transient} \ p\}
  \]

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

Figure 1: Definition of \( \textit{transient} \)

\textbf{Justifications}  The formal justification is based on induction on the program structure: (1) show that the basis rule is justified and then inductively prove the remaining rules. We give informal justification 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 \( \textit{post}_f \) holds. Hence, \( \neg p \lor \textit{post}_f \), i.e.\( (\neg p \land \neg \textit{post}_f) \), hold eventually in all cases. Note that if \( \textit{pre} \Rightarrow \neg p \) then \( \textit{pre} \land p \) is \( \textit{false} \) and the hypotheses are vacuously satisfied. If \( f \) never terminates, \( \neg \textit{post}_f \) 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, \( \alpha; \beta \). To prove \( \textit{transient} \ p \) it is sufficient that \( \alpha \) establish \( \neg p \) or that it execute effectively, thus establishing \( \textit{post}_\alpha \), and that \( \beta \) 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 \textbf{transient} \( p \) holds in \( f \).

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 “\textbf{if} \( b \) \textbf{then} \( f_0 \) \textbf{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 \( f \mid g \): Consider an execution in which, say, \( f \) has not terminated when \( p \) holds. From the arguments for the concurrency rule, using that \textbf{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 \textbf{transient} for each component.

\[ \square \]

**Notes**

1. The basis rule by itself is sufficient to define an elementary form of \textbf{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 \textbf{transient} \( p \). To see this consider the program \( f \mid g \) in which every action of \( f \) falsifies \( p \) only if for some predicate \( q \), \( p \land q \) holds as a precondition, and every action of \( g \) falsifies \( p \) only if \( p \land \neg q \) 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 \( p \land q \) and \( p \land \neg q \) can be shown transient. As we show later, \( p \ leads\text{-to} \ \neg p \).
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:

\[
\frac{\{ r \} f \{ p \land \neg q \text{ co } p \lor q, \text{ transient } p \land \neg q \}}{\{ r \} f \{ p \text{ en} q \}}
\]

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 (page 17).

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

\[
\begin{align*}
f_j :: \\
\text{initially } old_j, \ new_j = 0, 0 \\
\{ \text{true} \} \\
\text{loop} \\
\quad \{ \text{true} \} \\
\quad \alpha_j :: new_j := old_j + 1; \\
\quad \{ new_j = old_j + 1 \} \\
\quad \text{if } \begin{cases} \\
\beta_j :: \{ new_j = old_j + 1 \} \ ctr = old_j \rightarrow \ ctr := new_j \{ \text{true} \} \\
\gamma_j :: \{ new_j = old_j + 1 \} \ ctr \neq old_j \rightarrow old_j := ctr \{ \text{true} \} \\
\end{cases} \\
\{ \text{true} \} \\
\text{forever}
\end{align*}
\]

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 A.3 (page 17). For every \( f_j \) and any \( m \) and \( N \):

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

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 if statement. Using inheritance, prove (E) for \(\beta_j\) and \(\gamma_j\). In each case, assume the precondition \(ctr = m \land nb = N\) of if and the preconditions of \(\beta_j\) and \(\gamma_j\). The postcondition \(ctr = m \land nb < N \lor ctr > m\) is easy to see in each of the following cases:

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

\[
ctr := new_j
\}

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

\[
old_j := ctr
\}

\[
\{ctr = m \land nb < N \lor ctr > m\}
\]

### 5.3 Progress Property: Leads-to

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)** \(\frac{p \quad en \quad q}{p \rightarrow q}\)

- **(transitivity)** \(\frac{p \rightarrow q, q \rightarrow r}{p \rightarrow r}\)

- **(disjunction)** For any (finite or infinite) set of predicates \(S\)

\[
(\forall p : p \in S : p \rightarrow q)
\]

\[
(\forall p : p \in S : p) \rightarrow q
\]

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 \(f\).

\[
ctr = m \land nb = N \quad en \quad ctr = m \land nb < N \lor ctr > m
\]

proven in Section 5.2

\[
ctr = m \land nb = N \rightarrow ctr = m \land nb < N \lor ctr > m
\]

Applying the basis rule of *leads-to*

\[
ctr = m \rightarrow ctr > m
\]

Induction rule, use the well-founded order < over natural numbers

\[
true \rightarrow ctr > C,\ for\ any\ integer\ C
\]

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 [11]. 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 [8] 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 [12] is a pioneering paper.

The first method to suggest proof rules in the style of Hoare [5], and thus a specification technique, is due to Jones [6, 7]. 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 [10].

A completely different approach is suggested in the UNITY theory of Chandy and Misra [2], and extended in Misra [9]. 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 [11] 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 [14] has been effective in reasoning about concurrently accessed data structures. We are studying its relationship to the work described here.
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 [9].

\[
\begin{align*}
\text{false } & \text{ co } q \\
\frac{p \text{ co } q \land p' \text{ co } q'}{p \land p' \text{ co } q \land q'} & \quad \text{(CONJUNCTION)} \\
\frac{p \text{ co } q}{p \land p' \text{ co } q} & \quad \text{(LHS STRENGTHENING)} \\
\frac{p \text{ co } q \lor p' \text{ co } q'}{p \lor p' \text{ co } q \lor q'} & \quad \text{(DISJUNCTION)} \\
\frac{p \text{ co } q}{p \text{ co } q \lor q'} & \quad \text{(RHS WEAKENING)}
\end{align*}
\]

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)
  \[
  \frac{p \text{ co } q, \text{ stable } r}{p \land r \text{ co } q \land r} \\
  \frac{p \lor r \text{ co } q \lor r}{p \text{ co } q}
  \]

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

- invariant p, invariant q
  \[
  \frac{\text{invariant } p \land q}{\text{invariant } p \lor q} \\
  \frac{\{r\} f \{\text{stable } p \}}{\{r \land p\} f \{p\}} \\
  \frac{\{r\} f \{\text{constant } c \}}{\{r \land c = e\} f \{e = c\}}
  \]

- (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 \( p' \land \neg \text{post}_f \) for some \( p' \). Then in some annotation of \( f \) where action \( b \rightarrow \alpha \) has the precondition \( \text{pre} \):
   (1) \( \text{pre} \land p' \Rightarrow b \), and (2) \( \{ \text{pre} \land p' \} \alpha \{ \neg p' \} \).
   (1\text{'} From predicate calculus \( \text{pre} \land p' \land q \Rightarrow b \), and
   (2\text{'} from Hoare logic \( \{ \text{pre} \land p' \land q \} \alpha \{ \neg p' \} \). Applying the basis rule, transient \( p' \land q \land \neg \text{post}_f \), i.e., transient \( p \land q \).

2. (Sequencing) In \( f ; g \), transient \( p \land \neg \text{post}_f \) in \( f \) and transient \( p \) in \( g \). Inductively, transient \( p \land q \land \neg \text{post}_f \) 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)

\[
\{r\} f \{s\}
\]

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

- (Sequencing)

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

- (Concurrency)

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

\[
\begin{align*}
\text{Given: } & (\forall i :: \{ r_i \} \ f_i \ \{ s_i \}) \\
& \{ r \} \ f \ \{ s \} \\
\text{Assert: } & (\forall i :: \{ r_i \} \ f_i \ \{ p \ \text{en} \ q \ | \ s_i \}) \\
& \{ r \} \ f \ \{ p \ \text{en} \ q \ | \ s \}
\end{align*}
\]

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

2. (rhs weakening) \[ \frac{p \ \text{en} \ q \land q'}{p' \ \text{en} \ q} \]

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

4. (lhs manipulation) \[ \frac{p \land q \Rightarrow p' \Rightarrow q'}{p \ \text{en} \ q \equiv p' \ \text{en} \ q} \]
   Observe that \( p \land q \equiv p' \land q \) and \( p \lor q \equiv p' \lor q \). So, \( p \) and \( q \) are interchangeable in all the proof rules. As special cases, \( p \land q \ \text{en} \ q \equiv p \ \text{en} \ q \equiv p \lor q \ \text{en} \ q \).

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

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

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

Proof of (PSP): From the hypotheses:

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

We have to show:
transient $p \land r \land \neg(q \land r) \land \neg(r \land s)$ \hspace{1cm} (4)

$p \land r \land \neg(q \land r) \land \neg(r \land s)$ \hspace{1cm} (5)

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

$$p \land r \land \neg q \land s \
\equiv \{\text{expand both terms in rhs}\}
\quad p \land r \land \neg q \land s \lor p \land \neg r \land s \lor q \land r \land s \lor q \land \neg r \land s \
\Rightarrow \{\text{lhs strengthening and rhs weakening}\}
\quad p \land r \land \neg q \land s \land \neg p \land \neg r \land s \lor q \land r \land s \lor q \land \neg r \land s$$

A.4 Derived Rules for leads-to

The rules are taken from Misra [9] 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) \hspace{1cm} $p \Rightarrow q$$
\quad p \mapsto q$

2. (lhs strengthening, rhs weakening)

\[
\begin{array}{c}
p \mapsto q \\
\hline
\quad p^i \land p \mapsto q \\
\quad p \mapsto q \lor \bar{q}^i
\end{array}
\]

3. (disjunction) \hspace{1cm} (\forall i :: p_i \mapsto q_i)$$
\equiv \hspace{1cm} (\forall i :: p_i) \mapsto (\forall i :: q_i)$$

where $i$ is quantified over an arbitrary finite or infinite index set, and $p_i, q_i$ are predicates.

4. (cancellation) \hspace{1cm} $p \mapsto q \lor r, r \mapsto s$$
\quad p \mapsto q \land s$

A.4.2 Heavyweight rules

1. (impossibility) $\quad p \mapsto \text{false}$$
\quad \neg p$

2. (PSP) \hspace{1cm} $\quad p \mapsto q$$
\quad \text{stable } p^i$$
\quad p \land p^i \mapsto q \land \bar{p}^i$
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.

\[
(\forall i :: p_i \implies q_i \lor b)
\]

\[
(\forall i :: q_i \lor b)
\]

\[
(\forall i :: p_i) \implies (\forall i :: q_i) \lor b
\]

**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 \( f \).

(L) \[
p \implies q \text{ in } f
\]

\[
r \land x = X \implies \text{co } x = X \lor \neg r \text{ in } g
\]

\[
p \implies q \lor \neg r \text{ in } f \parallel g
\]

An informal justification of this rule is as follows. Any \( p \)-state in which \( \neg r \) holds, \( q \lor \neg r \) holds. We show that in any execution of \( f \parallel g \) starting in a \( (p \land r) \)-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

(L') \[
p \implies q \text{ in } f
\]

\[
p \land x = M \implies q \lor x \neq M \text{ in } f \parallel g
\]

The formal proof of (L) is by induction on the structure of the proof of \( p \implies q \text{ in } f \). See

http://www.cs.utexas.edu/users/psp/unity/notes/UnionLiftingRule.pdf

for details.

**B Example: Mutual exclusion**

We prove a coarse-grained version of a 2-process mutual exclusion program due to Peterson [13]. 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, $try$ 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 $turn$.

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 $turn$ 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 $turn := false$ in $M'$ as $turn' := true$; similarly the test for $turn$ is replaced by the dual test for $turn'$. Regard $turn'$ as the complement of $turn$. 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 $M$ 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 $cs, cs' = false, false$ — global initialization

$M$: initially $try = false$

\{¬try, ¬cs\}

loop — unrelated computation that may not terminate;

\{¬try, ¬cs\} $\alpha$: try, turn := true, true;

\{try, ¬cs\} $\beta$: ¬try' ∧ turn' → cs := true; — Enter critical section

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

forever

Remarks on the program The given program is based on a simplification of an algorithm in Peterson [13]. In the original version the assignment in $\alpha$ may be decoupled to the sequence $try := true; turn := true$. The tests in $\beta$ may be made separately for each disjunct in arbitrary order. Action $\gamma$ 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.

B.2 Safety and progress properties

It is required to show (1) the safety property: both $M$ and $M'$ are never simultaneously within their critical sections, i.e., invariant $\neg(cs \land cs')$, 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.

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

We prove:

\begin{align*}
\text{invariant } cs & \Rightarrow try \text{ in } M \sqcup M' \quad \text{(S1)} \\
\text{invariant } cs' \land try & \Rightarrow turn \text{ in } M \sqcup M' \quad \text{(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 $M$, which constitute its safety specification, from which (S1) and (S2) follow.

\begin{align*}
\text{stable } cs & \Rightarrow try \text{ in } M \quad \text{(S3)} \\
\text{stable } try & \Rightarrow turn \text{ in } M \quad \text{(S4)} \\
\text{stable } cs \land try' & \Rightarrow turn' \text{ in } M \quad \text{(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' ⇒ turn' and execution of β affects neither try' nor turn', so try' ⇒ turn' is a postcondition, and so is cs ∧ try' ⇒ turn'.

|   |   |   |   |
|---|---|---|---|
|(S3) | cs ⇒ try | post: ¬cs | α |
|(S4) | try ⇒ turn | post: turn | β |
|(S5) | cs ∧ try' ⇒ turn' | post: ¬cs | γ |

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

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) stable cs ⇒ try in \( M \parallel M' \):
  - stable cs ⇒ try in \( M \), shown as (S3)
  - stable cs ⇒ try in \( M' \), cs and try are constant in \( M' \)
  - stable cs ⇒ try in \( M \parallel M' \), Inheritance rule

- (S2) stable cs' ∧ try ⇒ turn in \( M \parallel M' \):
  - stable try ⇒ turn in \( M \), shown as (S4)
  - stable cs' ∧ try ⇒ turn in \( M \), cs' constant in \( M \)
  - stable cs' ∧ try ⇒ turn in \( M' \), dual of (S5)
  - stable cs' ∧ try ⇒ turn in \( M \parallel M' \), Inheritance rule

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

First, we prove one safety and one ensures property of \( M \parallel M' \).

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

Proofs of (S6) and (P1): To prove (S6): prove try ∧ ¬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 (P1.1) and (S7), which constitute the progress specification of \( M \) (dual specification for \( M' \)).

\[
\begin{align*}
\text{try} \land (\neg \text{try}' \lor \text{turn}') & \ \text{en} \ \neg \text{try} \ \text{in} \ \( M \) \quad \text{(P1.1)} \\
\text{stable} \ \neg \text{try' \lor turn'} & \ \text{in} \ \( M \) \quad \text{(S7)}
\end{align*}
\]

The proof of (P1.1) uses the sequencing rule and (S7) is straight-forward. Then, using duality on (S7), stable ¬try' ∨ 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.

\[
\begin{align*}
\text{try} \land (\neg \text{try'} \lor \text{turn'}) & \mapsto \neg \text{try} \\
\text{try} \land \neg \text{try'} & \mapsto \neg \text{try} \\
\text{try} \land \text{turn'} & \mapsto \neg \text{try} \\
\end{align*}
\]

(P2)

(P2.1)

(P2.2)

The main proof:

\[
\begin{align*}
\text{try} \land \text{turn'} & \mapsto \neg \text{try} \quad \text{, from (P2.2)} \\
\text{try'} \land \text{turn} & \mapsto \neg \text{try'} \quad \text{, duality} \\
\text{try} \land \text{try'} \land \text{turn} & \mapsto \neg \text{try} \lor \text{try} \land \neg \text{try'} \\
\text{try} & \mapsto \neg \text{try} \quad \text{, disjunction with (P2)} \\
\text{try} \land \neg \text{cs co cs} & \mapsto \neg \text{try} \quad \text{, (S6)} \\
\text{try} \land \neg \text{cs} & \mapsto \text{cs} \quad \text{, PSP of above two} \\
\text{try} & \mapsto \text{cs} \quad \text{, disjunction with } \text{try} \land \text{cs} \mapsto \text{cs}
\end{align*}
\]

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 get and put are given by:

\[
\begin{align*}
\{u' = u\} \text{get}(z) & \{u' = u \cup \{z\}\} \\
\{u' = u\} \text{put}(z) & \{u = u' \cup \{z\}\}
\end{align*}
\]

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

\[
\begin{align*}
f_1 = \begin{cases} 
|u| > 0 & \rightarrow \text{get}(x); \ |u| > 0 & \rightarrow \text{get}(y); \\
\text{put}(x \oplus y)
\end{cases} \\
f_{k+1} = \begin{cases} 
f_1 \parallel f_k, 
\end{cases} \quad k \geq 1
\end{align*}
\]
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 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 put operations. A get does not affect $np_k$ and a put increments it. Initially $np_k = 0$.

The modified program is as follows where $⟨\cdots⟩$ is an action.

$$f_1 = |u| > 0 \rightarrow \langle \text{get}(x); w_1 := w_1 \cup \{x\} \rangle;$$

$$|u| > 0 \rightarrow \langle \text{get}(y); w_1 := w_1 \cup \{y\} \rangle;$$

$$\langle \text{put}(x \oplus y); w_1 := w_1 - \{x, y\}; np_1 := 1 \rangle$$

$$f_{k+1} = f_1 \parallel f_k, \quad k \geq 1$$

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 $k$, $k \geq 1$:

$$\{u = u_0\} f_k \{\Sigma u = \Sigma u_0, \quad |u| + k = |u_0|\}$$

$$|u| + |w_k| + np_k > k \rightarrow np_k = k \text{ in } f_k$$

(T)

$$|u| + |w_k| + np_k > k \rightarrow np_k = k \text{ in } f_k$$

(P)

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 3.4, to establish (T). First, we show for all $k$, $k \geq 1$:

$$\{w_k = \{\}, \quad np_k = 0\} f_k$$

$$\{\text{constant } \Sigma (u \cup w_k), \quad \text{constant } |u| + |w_k| + np_k \mid w_k = \{\}, \quad np_k = k\}$$

(T')
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 \( \{ a_k \} f_k \{ b_k \mid c_k \} \). Proof is by induction on \( k \).

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

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

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

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

\[ \{ a_k \} f_k \{ b_{k+1} \mid c_k \} \]
\[ \{ a_k \} f_k \{ b_{k+1} \mid c_k \}, \text{ from the annotation of } f_1 \]
\[ \{ a_k \} f_k \{ b_{k+1} \mid c_k \}, \text{ w_k, np_k constant in } f_k \] \hspace{1cm} (1)
\[ \{ a_k \} f_k \{ b_{k+1} \mid c_k \}, \text{ inductive hypothesis} \]
\[ \{ a_k \} f_k \{ b_{k+1} \mid c_k \}, \text{ w_1, np_1 constant in } f_k \] \hspace{1cm} (2)
\[ \{ a_k \} f_k \{ b_{k+1} \mid c_k \}, \text{ join proof rule on (1,2) } \]
\[ \{ a_k \} f_k \{ b_{k+1} \mid c_k \}, \text{ inheritance on (1,2) } \]
\[ \{ a_k \} f_k \{ b_{k+1} \mid c_{k+1} \}, \text{ a_{k+1} = a_k \text{ and } c_{k+1} = c_k } \]

**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 \( k, k \geq 1 \).

\[ \{ a_k \} f_k \{ b_k \mid c_k \} \]
\[ \{ u = u_0, a_k \} f_k \{ b_k \mid c_k \}, \text{ lhs strengthening from Section 3.4} \]
\[ \{ u = u_0, a_k \} f_k \]
\[ \{ \text{invariant } \Sigma(u \cup w_k) = \Sigma u_0, \text{ invariant } |u| + |w_k| + np_k = |u_0| \mid c_k \} \]
\[ \{ u = u_0, a_k \} \text{ definition of } a_k \text{ and invariant} \]
\[ \{ u = u_0, a_k \} f_k \{ \Sigma(u \cup w_k) = \Sigma u_0, |u| + |w_k| + np_k = |u_0|, c_k \} \]
\[ \{ u = u_0, a_k \} \text{ invariance rule, Section 3.4, in postcondition} \]
\[ \{ u = u_0, w_k = \{ \} \land np_k = 0 \} f_k \{ \Sigma u = \Sigma u_0, |u| + k = |u_0| \} \]
\[ \{ u = u_0, a_k \} \text{ definitions of } a_k \text{ and } c_k \]
C.2 Proof of Progress property (P)

Abbreviations: Abbreviate \(|u| + |w_1| + np_k > k\) by \(p_k\) and \(np_k = k\) by \(q_k\) so that for all \(k, k \geq 1\), Property (P) is:

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

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

C.2.1 Progress proof \(p_1 \rightarrow q_1\) in \(f_1\)

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

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

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 \(p_1\)):

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

C.2.2 Progress proof, \(p_{k+1} \rightarrow q_{k+1}\) 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 \(\prec\) over pairs of sets such as \(z_k\): \((a, b) \prec (a', b')\) 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 \(w_1\) 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, \( p_{k+1} \Rightarrow q_{k+1} \) 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).

\[
\begin{align*}
p_{k+1} \Rightarrow p_{k+1} \land (q_1 \lor q_k) & \text{ in } f_{k+1} \\
p_{k+1} \land (q_1 \lor q_k) \Rightarrow q_{k+1} & \text{ in } f_{k+1} \quad \text{(P1)}
\end{align*}
\]

\[
\begin{align*}
p_{k+1} \Rightarrow p_{k+1} \land (q_1 \lor q_k) & \text{ in } f_{k+1} \\
p_{k+1} \land (q_1 \lor q_k) \Rightarrow q_{k+1} & \text{ in } f_{k+1} \quad \text{(P2)}
\end{align*}
\]

We have \( p_1 \Rightarrow q_1 \) and, inductively, \( p_k \Rightarrow q_k \). The proofs mostly use the derived rules of \textit{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 \( f_{k+1} \). Lifting rule refers to rule (L’) of Section A.4.3.

\[
\begin{align*}
p_1 \land z_{k+1} = n \Rightarrow q_1 \lor z_{k+1} \neq n & \quad \text{Lifting rule on } p_1 \Rightarrow q_1 \text{ in } f_1 \\
p_k \land z_{k+1} = n \Rightarrow q_k \lor z_{k+1} \neq n & \quad \text{Lifting rule on } p_k \Rightarrow q_k \text{ in } f_k \\
(p_1 \lor p_k) \land z_{k+1} = n \Rightarrow (q_1 \lor q_k) \lor z_{k+1} \neq n & \quad \text{disjunction} \\
(p_1 \lor p_k) \land z_{k+1} = n \Rightarrow (q_1 \lor q_k) \lor z_{k+1} \neq n & \quad \text{(PSP) with } \text{stable } z_{k+1} \leq n \\
p_{k+1} \land (p_1 \lor p_k) \land z_{k+1} = n \Rightarrow p_{k+1} \land (q_1 \lor q_k) \lor p_{k+1} \land z_{k+1} < n & \quad \text{conjunction with } \text{stable } p_{k+1} \\
p_{k+1} \land z_{k+1} = n \Rightarrow p_{k+1} \land z_{k+1} < n \lor p_{k+1} \land (q_1 \lor q_k) & \quad \text{, } p_{k+1} \Rightarrow (p_1 \lor p_k) \\
p_{k+1} \Rightarrow p_{k+1} \land (q_1 \lor q_k) & \quad \text{, induction rule of leads-to}
\end{align*}
\]

**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 \( f_{k+1} \).

\[
\begin{align*}
p_1 \land z_{k+1} = n \Rightarrow q_1 \lor z_{k+1} \neq n & \quad \text{Lifting rule on } p_1 \Rightarrow q_1 \text{ in } f_1 \\
p_1 \land z_{k+1} = n \Rightarrow q_1 \lor z_{k+1} < n & \quad \text{conjunction with } \text{stable } z_{k+1} \leq n \\
q_k \land p_1 \land z_{k+1} = n \Rightarrow q_k \land q_1 \lor q_k \land z_{k+1} < n & \quad \text{conjunction with } \text{stable } q_k \\
p_{k+1} \land q_k \land p_1 \land z_{k+1} = n \Rightarrow p_{k+1} \land q_k \land q_1 \lor p_{k+1} \land q_k \land z_{k+1} < n & \quad \text{conjunction with } \text{stable } p_{k+1} \\
p_{k+1} \land q_k \land z_{k+1} = n \Rightarrow p_{k+1} \land q_k \land z_{k+1} < n \lor p_{k+1} \land q_k \land q_1 & \quad \text{, } p_{k+1} \land q_k \Rightarrow p_1 \\
p_{k+1} \land q_k \Rightarrow p_{k+1} \land q_1 \land q_k & \quad \text{, induction rule of leads-to} \\
p_{k+1} \land q_k \Rightarrow q_1 \land q_k & \quad \text{, rhs weakening} \\
p_{k+1} \land q_1 \Rightarrow q_1 \land q_k & \quad \text{, similarly} \\
p_{k+1} \land (q_1 \lor q_k) \Rightarrow q_{k+1} & \quad \text{disjunction and } q_1 \land q_k \Rightarrow q_{k+1}
\end{align*}
\]
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] Edsger W. Dijkstra. Guarded commands, nondeterminacy, and the formal derivation of programs. *Communications of the ACM*, 8:453–457, 1975.

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

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

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

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

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

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

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

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

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

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

[14] 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.