An Axiomatic Treatment of Partial Correctness and Deadlock in a Shared Variable Parallel Language

Stephen Brookes

June 1992
CMU-CS-92-154

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This paper is an expanded and improved version of *An Axiomatic Treatment of a Parallel Language*, which appeared in *Logics of Programs*, Springer LNCS vol. 193, 1985.

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597, in part by National Science Foundation grant CCR-9006064 and in part by DARPA/NSF grant CCR-8906483.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NSF, DARPA or the U.S. government.

Keywords: imperative programming, semantics, parallelism, shared memory, partial correctness, deadlock, logic

Abstract

We give a semantically based axiomatic treatment of partial correctness and deadlock for an imperative shared variable parallel programming language. The Owicki-Gries proof methodology for this language proves conventional Hoare-style partial correctness assertions and involves the notion of interference-freedom of proofs (to guarantee soundness), auxiliary variables (to guarantee relative completeness), and global invariants (to permit reasoning about deadlock-freedom). Our axiomatic proof system is based more explicitly on the underlying operational semantics, using assertions whose syntactic structure directly reflects the operational behavior of parallel programs at an appropriate level of abstraction. We build a proof system that requires neither interference-freedom nor auxiliary variables. Novel features include the use of a syntactic form of parallel composition of assertions, and the use of conjunction and implication as connectives on assertions. It is possible simultaneously to reason about partial correctness and deadlock-freedom using our proof system, without recourse to global invariants. We discuss some non-trivial examples, and compare our proof methodology with some other proof methods from the literature.

1 Introduction

It has become widely accepted since Hoare's classic paper [Hoa69] that formal reasoning about program correctness properties is desirable. Hoare treated partial correctness properties of commands in a sequential imperative programming language, using simple assertions based on pre- and post-conditions. The proof system was *syntax-directed*, in that axioms or rules were given for each syntactic construct; each inference rule allows the deduction of a partial correctness property of a command from premisses which are partial correctness properties of its immediate syntactic sub-commands. The partial correctness assertions (pca's) chosen by Hoare are admirably suited to the task: they are concise in structure and have a clear correlation with a state transformation semantics for the programming language. This means that fairly straightforward proofs of the soundness and (relative) completeness of Hoare's proof system can be given [Apt81, Coo78].

For more complicated programming languages the story is not so simple. Several proposed axiomatic treatments of programming languages have turned out to be either unsound or incomplete [OD82]. The task of establishing soundness and completeness of proof systems for program properties can be complicated by an excessive amount of detail used in the semantic description of the programming language. This point seems to be quite well known, and is made, for instance in [Apt81]. Similar problems can be caused by the use of an excessively intricate or poorly structured assertion language, or by overly complicated proof rules. Certainly for sequential languages with state-transformation semantics the usual Hoare-style assertions with pre- and post-conditions are suitable. But for languages which require more sophisticated semantic treatment we believe that it is inappropriate to try to force assertions to fit into the pre- and post-condition mould; such an attempt tends to lead to pre- and post-conditions with a rather complex structure, when it could be simpler to use a class of assertions with a different structure which more accurately corresponds to the semantics. The potential benefits of basing an axiomatic treatment directly on a well chosen semantics has been argued, for instance, in [Bro85], where an axiomatic treatment of aliasing was given. We believe that implicit in the design of Hoare's logic was the idea that the semantic structure of a language should determine the choice of assertion language for specifying program properties. It is this idea, rather than the use of assertions built from a pre-condition and a post-condition, that should form the basis for generalization when we axiomatize more complex programming languages. In this paper we take such an approach for a simple imperative parallel programming language.

Parallel programming languages certainly require a more sophisticated semantic model than sequential languages. Proof systems for reasoning about various forms of parallelism have been proposed by several authors, notably [AFdR80, Ash75, BKP84, Ger83, Kel76, Lam80, LS84, LG81, MP82, OG76]. Owicki and Gries [OG76] gave a Hoare-style axiom system for a programming language in which parallel commands can interact through their effects on shared variables. Their proof rule for parallel composition involved a notion of *interference-freedom* and used *proof outlines* as premisses. In order to obtain a (relatively) complete proof system Owicki found it necessary to use *auxiliary variables* and to add proof rules for dealing with them. These features have been the subject of considerable discussion in the literature, such as [Bes82, Lam80]. Some alternative proof systems, such as those of Lamport and Schneider [LS84] and the transition logic of Gerth [Ger83], employ essentially the same syntactic structure for assertions (again pca's) but modify the interpretation given to assertions. All of these approaches attempt to maintain at least a syntactic similarity of structure with Hoare's pca's. Most of the proof systems which have emerged so far have followed Owicki in imposing some form of interference-freedom condition as a constraint on the rule for parallel composition. Most of these axiomatizations have been designed with partial correctness in mind, and in cases where it is possible to adapt the proof system to deduce deadlock-freedom properties this is typically done by means of global invariants.

In this paper we present an alternative axiomatization based directly on a structural operational semantics [Plo81], essentially as given by Hennessy and Plotkin [HP79]. In view of the congruence established in [HP79] between this operational semantics and a denotational semantics based on a powerdomain of *resumptions*, our axiomatization can also be thought of as built on a resumption semantics. We define *syntactic* operations on assertions which correspond to the *semantics* of the various syntactic constructs in the programming language; in particular, we define sequential and parallel composition for assertions. This leads naturally to *compositional*, or syntax-directed, proof rules for the syntactic constructs. We do not need an interference-freedom condition in our rule for parallel composition, in contrast to Owicki's system. Similarly, we do not need an auxiliary variables rule. We show how to re-construct Owicki's rule for parallel composition with its interference-freedom condition, using our methods. Essentially, Owicki's system uses a restricted subset of our assertions and a variant form of parallel composition of assertions.

We apply our proof methodology to some non-trivial parallel programming examples from the literature, including Peterson's algorithm, a producer-consumer program, and the problem of partitioning a set. We further compare our work briefly with that of some other authors, discuss some of its limitations, and make a few suggestions for further research.

2 A Parallel Programming Language

2.1 Syntax

Our programming language is obtained by adding two features to a simple imperative while-loop language: a form of parallel composition, and a "conditional critical region" construct [Hoa72]. Parallel commands interact solely through their effects on shared variables. There are four syntactic sets: **Ide**, the set of identifiers, ranged over by I; **Exp**, the set of expressions, ranged over by E; **BExp**, the set of boolean expressions (or conditions), ranged over by B; and **Com**, the set of commands, ranged over by C. The abstract syntax for identifiers, expressions and conditions will be taken for granted; all we assume is that identifiers and expressions denote integer or boolean values, and the language contains the usual arithmetic and boolean operators and constants. For commands we specify the following abstract syntax:

$$\begin{array}{rcl} C & ::= & \mathbf{skip} \mid I{:=}E \mid C_1;C_2 \mid \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \mid \\ & & \mathbf{while}\ B\ \mathbf{do}\ C \mid [C_1 \| C_2] \mid \mathbf{await}\ B\ \mathbf{then}\ \langle C \rangle \end{array}$$

The command **skip** is an atomic action having no effect on program variables. An assignment is denoted $I{:=}E$, and is also an atomic action. Sequential composition is represented by $C_1;C_2$. A parallel composition $[C_1 \| C_2]$ is executed by interleaving the atomic actions of the component commands C_1 and C_2. A command of the form **await** B **then** $\langle C \rangle$ is a *conditional critical region* with *test* B and *body* C; we impose the syntactic constraint that the body C should be a finite sequence of assignments (or **skip**)[1]; the intended semantics will be to execute C as an atomic step when B is true. We use the abbreviations $\langle C \rangle$ for **await true then** $\langle C \rangle$, and **await** B for **await** B **then** $\langle$**skip**$\rangle$.

[1] This syntactic constraint is not theoretically necessary, and causes no significant loss in the expressive power of the programming language, but seems reasonable in order to guarantee that critical regions may be implemented easily.

2.2 Semantics

In describing the semantics of this language, we will focus mainly on commands. The set S of *states* consists simply of the functions from identifiers to denotable values:

$$S = [\mathbf{Ide} \rightarrow V],$$

where V is the set of integers and truth values. We use s to range over states, and we write $s + [I \mapsto v]$ for the state which agrees with s except that it gives identifier I the value v. As usual, the value denoted by an expression may depend on the values of its free identifiers. Thus, we assume the existence of a semantic function

$$\mathcal{E} : \mathbf{Exp} \rightarrow [S \rightarrow V].$$

Similarly, we assume given a semantic function $\mathcal{B} : \mathbf{BExp} \rightarrow [S \rightarrow T]$, where $T = \{\mathtt{tt}, \mathtt{ff}\}$ is the set of truth values. We make the simplifying assumption that all expressions and conditions denote a proper value, i.e. that evaluation of expressions and conditions always terminates. Moreover, expression evaluation does not cause any side-effects.

We specify the semantics of commands in the structural operational style [Plo81], and our presentation follows that of [HP79]. We define first an abstract machine which specifies the computations of a command. The abstract machine is given by a *labelled transition system*

$$\langle \mathbf{Conf}, \mathbf{Lab}, \rightarrow \rangle,$$

where **Conf** is a set of *configurations*, **Lab** is a set of *labels* (ranged over by α and β), and $\rightarrow$ is a family

$$\{\xrightarrow{\alpha} | \alpha \in \mathbf{Lab}\}$$

of *transition relations* $\xrightarrow{\alpha} \subseteq \mathbf{Conf} \times \mathbf{Conf}$ indexed by elements of **Lab**. We use atomic actions as labels.

For convenience we introduce a term **null** to represent termination, and we specify (purely for notational purposes) that

$$\begin{aligned} [\mathbf{null} \| C] &= [C \| \mathbf{null}] = C, \\ \mathbf{null}; C &= C. \end{aligned}$$

The set of configurations is $\mathbf{Conf} = (\mathbf{Com} \cup \{\mathbf{null}\}) \times S$. A configuration of the form $\langle C, s\rangle$ will represent a stage in a computation at which the remaining command to be executed is C, and the current state is s. A configuration of the form $\langle \mathbf{null}, s\rangle$ represents termination in the given state. A *transition* of the form

$$\langle C, s\rangle \xrightarrow{\alpha} \langle C', s'\rangle$$

represents a computation step in which the state and remaining command change as indicated, and in which the atomic action α occurs. We write $\langle C, s\rangle \rightarrow \langle C', s'\rangle$ when there is an α for which $\langle C, s\rangle \xrightarrow{\alpha} \langle C', s'\rangle$. And we use the notation $\rightarrow^*$ for the reflexive transitive closure of this relation. Thus $\langle C, s\rangle \rightarrow^* \langle C', s'\rangle$ iff there is a sequence of atomic actions from the first configuration to the second.

The transition relation is defined to be the smallest relation on configurations satisfying the syntax-directed transition rules given in Figure 1. This means that a transition is possible if and only if it can be deduced from these rules.

Note that this transition system specifies that a parallel composition terminates only when both components have terminated, because of our conventions about **null**: we have $\langle [C_1 \| C_2], s\rangle \xrightarrow{\alpha} \langle C_2, s'\rangle$ whenever $\langle C_1, s\rangle \xrightarrow{\alpha} \langle \mathbf{null}, s'\rangle$, for instance.

$$\langle \mathbf{skip}, s\rangle \xrightarrow{\mathbf{skip}} \langle \mathbf{null}, s\rangle$$

$$\langle I{:=}E, s\rangle \xrightarrow{I:=E} \langle \mathbf{null}, s + [I \mapsto \mathcal{E}[\![E]\!]s]\rangle$$

$$\frac{\langle C_1, s\rangle \xrightarrow{\alpha} \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \xrightarrow{\alpha} \langle C_1'; C_2, s'\rangle}$$

$$\frac{\langle C_1, s\rangle \xrightarrow{\alpha} \langle C_1', s'\rangle}{\langle [C_1 \| C_2], s\rangle \xrightarrow{\alpha} \langle [C_1' \| C_2], s'\rangle}$$

$$\frac{\langle C_2, s\rangle \xrightarrow{\alpha} \langle C_2', s'\rangle}{\langle [C_1 \| C_2], s\rangle \xrightarrow{\alpha} \langle [C_1 \| C_2'], s'\rangle}$$

$$\frac{\mathcal{B}[\![B]\!]s = \mathtt{tt} \quad \langle C, s\rangle \rightarrow^* \langle \mathbf{null}, s'\rangle}{\langle \mathbf{await}\ B\ \mathbf{then}\ \langle C\rangle, s\rangle \xrightarrow{\langle C\rangle} \langle \mathbf{null}, s'\rangle}$$

$$\frac{\mathcal{B}[\![B]\!]s = \mathtt{tt}}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle \xrightarrow{B} \langle C_1, s\rangle}$$

$$\frac{\mathcal{B}[\![B]\!]s = \mathtt{ff}}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle \xrightarrow{B} \langle C_2, s\rangle}$$

$$\frac{\mathcal{B}[\![B]\!]s = \mathtt{tt}}{\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \xrightarrow{B} \langle C; \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle}$$

$$\frac{\mathcal{B}[\![B]\!]s = \mathtt{ff}}{\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \xrightarrow{B} \langle \mathbf{null}, s\rangle}$$

Figure 1: Transition Rules

2.3 Examples

1. The command $[x{:=}x+1 \| x{:=}x+1]$ has two transition sequences from state $[x \mapsto 0]$, which produce the same effect:

$$\begin{aligned}\langle [x{:=}x+1 \| x{:=}x+1], [x \mapsto 0]\rangle &\xrightarrow{x:=x+1} \langle x{:=}x+1, [x \mapsto 1]\rangle \\ &\xrightarrow{x:=x+1} \langle \mathbf{null}, [x \mapsto 2]\rangle.\end{aligned}$$

2. The command $[x{:=}2 \| (x{:=}1; x{:=}x+1)]$ has three transition sequences from $[x \mapsto 0]$:

$$\begin{aligned}\langle [x{:=}2 \| (x{:=}1; x{:=}x+1)], [x \mapsto 0]\rangle &\xrightarrow{x:=2} \langle x{:=}1; x{:=}x+1, [x \mapsto 2]\rangle \\ &\xrightarrow{x:=1} \langle x{:=}x+1, [x \mapsto 1]\rangle \\ &\xrightarrow{x:=x+1} \langle \mathbf{null}, [x \mapsto 2]\rangle\end{aligned}$$

$$\begin{aligned}\langle [x{:=}2 \| (x{:=}1; x{:=}x+1)], [x \mapsto 0]\rangle &\xrightarrow{x:=1} \langle [x{:=}2 \| x{:=}x+1], [x \mapsto 1]\rangle \\ &\xrightarrow{x:=2} \langle x{:=}x+1, [x \mapsto 2]\rangle \\ &\xrightarrow{x:=x+1} \langle \mathbf{null}, [x \mapsto 3]\rangle\end{aligned}$$

$$\begin{aligned}\langle [x{:=}2 \| (x{:=}1; x{:=}x+1)], [x \mapsto 0]\rangle &\xrightarrow{x:=1} \langle [x{:=}2 \| x{:=}x+1], [x \mapsto 1]\rangle \\ &\xrightarrow{x:=x+1} \langle x{:=}2, [x \mapsto 2]\rangle \\ &\xrightarrow{x:=2} \langle \mathbf{null}, [x \mapsto 2]\rangle.\end{aligned}$$

3. From any initial state s the command

$$[(x{:=}1; \mathbf{await}\ y=1\ \mathbf{then}\ \langle x{:=}y-1\rangle) \| (y{:=}1; \mathbf{await}\ x=1)]$$

can perform the sequence $x{:=}1$, $y{:=}1$, $x{:=}y-1$ and reach the stuck configuration

$$\langle \mathbf{await}\ x=1,\ s+[x \mapsto 0, y \mapsto 1]\rangle.$$

The command also has a successfully terminating computation in which the atomic steps are $y{:=}1$, $x{:=}1$, **skip**, $x{:=}y-1$, leading to the final configuration

$$\langle \mathbf{null}, s+[x \mapsto 0, y \mapsto 1]\rangle.$$

2.4 Partial Correctness and Deadlock

A *computation* is a maximal sequence of transitions; in general a command may have both finite and infinite computations. A computation ending in a configuration of form $\langle \mathbf{null}, s\rangle$ represents successful termination. We say that a configuration $\langle C, s\rangle$ is deadlocked iff it cannot perform any atomic action but has not terminated successfully (i.e., C is not **null**). Informally, this happens iff C is an await command whose test is false in state s, or C is a sequential composition $C_1; C_2$ and $\langle C_1, s\rangle$ is deadlocked, or C is a parallel composition all of whose components are deadlocked. The following definition makes this precise.

Definition 2.1 The predicate *stuck* on configurations is the least relation satisfying:

$$\frac{\mathcal{B}[\![B]\!]s = \mathtt{ff}}{\mathrm{stuck}\langle \mathbf{await}\ B\ \mathbf{then}\ \langle C\rangle, s\rangle}$$

$$\frac{\mathrm{stuck}\langle C_1, s\rangle}{\mathrm{stuck}\langle C_1; C_2, s\rangle}$$

$$\frac{\mathrm{stuck}\langle C_1, s\rangle \quad \mathrm{stuck}\langle C_2, s\rangle}{\mathrm{stuck}\langle [C_1 \| C_2], s\rangle}$$

A configuration $\langle C, s\rangle$ is deadlocked iff $\mathrm{stuck}\langle C, s\rangle$ is provable from these rules. •

There are thus three kinds of computation: infinite, finite successful and finite deadlocked. Since we are concerned here with partial correctness and deadlock we will focus on the finite computations. To this end we now introduce a partial correctness semantic function $\mathcal{M}$ and a deadlock semantic function $\mathcal{D}$:

Definition 2.2 The semantic functions $\mathcal{M}, \mathcal{D} : \mathbf{Com} \rightarrow [S \rightarrow \mathcal{P}(S)]$ are:

$$\begin{array}{rcl}\mathcal{M}[\![C]\!]s & = & \{s' | \langle C, s\rangle \rightarrow^* \langle \mathbf{null}, s'\rangle\} \\ \mathcal{D}[\![C]\!]s & = & \{s' | \exists C'.\ \langle C, s\rangle \rightarrow^* \langle C', s'\rangle \ \&\ \mathrm{stuck}(\langle C', s'\rangle)\}.\end{array}$$

•

A conventional partial correctness assertion (pca) has the form $\{P\}\, C\, \{Q\}$, where P and Q are conditions, typically drawn from a fixed first order language. We introduce a generalization: *partial correctness and deadlock* assertions.

Definition 2.3 A *partial correctness and deadlock assertion* (pcda) has the form

$$\{P\}\, C\, \{Q\}\, [R],$$

where P,Q and R are conditions; we call P the pre-condition, Q the termination post-condition, and R the deadlock post-condition. •

The pcda $\{P\}\, C\, \{Q\}\, [R]$ is valid iff every successful computation of C from a state satisfying P ends in a state satisfying Q, and every deadlocking computation of C from a state satisfying P gets stuck in a state satisfying R. This generalizes the usual notion of validity for pca's in a natural manner. In particular, an assertion of the form $\{P\}\, C\, \{Q\}\, [\mathbf{false}]$ states that every terminating computation of C from an initial state satisfying P ends successfully in a state satisfying Q: this is a combination of a pca with deadlock-freedom.

Definition 2.4 The pcda $\{P\}\, C\, \{Q\}\, [R]$ is *valid* iff

1. $\forall s, s'.(s \models P\ \&\ s' \in \mathcal{M}[\![C]\!]s \quad \Rightarrow \quad s' \models Q);$
2. $\forall s, s'.(s \models P\ \&\ s' \in \mathcal{D}[\![C]\!]s \quad \Rightarrow \quad s' \models R).$

•

$$\vdash \{P\}\,\mathbf{skip}\,\{P\}\,[\mathbf{false}]$$

$$\vdash \{[E \setminus I]P\}\,I{:=}E\,\{P\}\,[\mathbf{false}]$$

$$\frac{\{P\&B\}\,C_1\,\{Q\}\,[R] \quad \{P\&\neg B\}\,C_2\,\{Q\}\,[R]}{\{P\}\,\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\,\{Q\}\,[R]}$$

$$\frac{\{P\&B\}\,C\,\{Q\}\,[R]}{\{P\}\,\mathbf{while}\ B\ \mathbf{do}\ C\,\{P\&\neg B\}\,[R]}$$

$$\frac{\{P\}\,C_1\,\{Q\}\,[R_1] \quad \{Q\}\,C_2\,\{R\}\,[R_2]}{\{P\}\,C_1;C_2\,\{R\}\,[R_1 \vee R_2]}$$

$$\frac{\{P\&B\}\,C\,\{Q\}\,[R]}{\{P\}\,\mathbf{await}\ B\ \mathbf{then}\ \langle C\rangle\,\{Q\}\,[P\&\neg B]}$$

Figure 2: Some Proof Rules for Partial Correctness and Deadlock

We are interested in finding a proof methodology for establishing pcda's. It is worth noting that Hoare's axioms and rules for sequential while-programs and the Owicki-Gries rule for conditional critical regions can be simply adapted to the pcda setting, as shown in Figure 2. However, it is impossible to devise a similar proof rule for parallel composition of commands. It is well known [OG76, HP79] that in order to deduce partial correctness properties of a parallel command it is necessary to take account of the context in which the command is to be run. For example, the commands $x{:=}2$ and $x{:=}1; x{:=}x+1$ clearly have the same partial correctness properties, but $[x{:=}2 \| x{:=}2]$ sets x to 2 and $[(x{:=}1; x{:=}x+1) \| x{:=}2]$ sets x to 2 or 3. In order to reason about the partial correctness of a parallel combination of commands in a manner independent of the context in which the command appears, we need to know more about the individual commands than simply their partial correctness semantics. A similar argument shows that deadlock behavior of a parallel program cannot be deduced in a straightforward syntax-directed way. For these reasons there is no general inference rule allowing the deduction of a pcda $\{P\}\,C_1 \| C_2\,\{Q\}\,[R]$ from premisses of the form $\{P_i\}\,C_i\,\{Q_i\}\,[R_i]$ $(i = 1, 2)$. Instead we will adopt a more highly structured assertion language.

3 A semantically based assertion language

We have seen that the operational semantics given above has enough detail to allow us to derive the semantic functions $\mathcal{M}$ and $\mathcal{D}$, although these functions themselves cannot be defined directly in the denotational style (that is, by syntactic structural induction). Since we presented the operational semantics by providing a collection of structural rules, we can use the transition system itself as the basis for reasoning about program behavior in a syntax-directed style. The transition system specifies, for each command and each state, the atomic actions that the command may perform next from that state and the command and updated state resulting when each such action occurs. Moreover, the transition system is finitely branching: from any $\langle C, s\rangle$ there are only finitely many immediately enabled transitions. Thus, each command and initial state determines a finitely branching, possibly infinitely deep, computation tree.

The branching structure of the transition system suggests that we adopt a class of tree-like assertions, in which the properties of intermediate stages of a computation as well as initial and final states can be specified. Moreover, in order to allow the proper treatment of deadlock and termination, we should maintain a distinction between stuck and successful final states. This suggests that we employ assertions rather like Milner's *labelled synchronization trees* [Mil80], with atomic actions labelling the arcs, but with pre- and post-conditions for each arc and with two forms of leaf node: • for termination and ∘ for deadlock. Following Milner, we will also allow the use of μ-notation for describing infinite trees with recursive structure, so that we can handle loops in a straightforward manner.

However, there is another important semantic consideration that will influence our choice of assertion language. Hoare's logic for sequential while-programs enjoys the property that, for any valid pca $\{P\}\,C\,\{Q\}$ there is a proof that employs a single pca for each syntactic subcommand of C. This "single-threadedness" property is the key to the definition of "proof outline" and to Cook's proof of relative completeness [Coo78]. It is easy to see that parallel programs do not enjoy an analogous single-threadedness property. For example, to show the validity of the assertion

$$\{x=0\}\,[x{:=}x+1\|x{:=}x+2]\,\{x=3\}$$

one needs (the information conveyed by) the following four pca's, two for each subcommand:

$$\begin{aligned} &\{x=0\}\,x{:=}x+1\,\{x=1\}, \\ &\{x=2\}\,x{:=}x+1\,\{x=3\}, \\ &\{x=0\}\,x{:=}x+2\,\{x=2\}, \\ \text{and }\ &\{x=1\}\,x{:=}x+2\,\{x=3\}. \end{aligned}$$

The single-threadedness property will also fail for tree-structured assertions, unless we allow the use of *conjunctions* of assertions. We will, therefore, include conjunction as a logical connective on assertions. It is perhaps worth remarking that Hoare's logic as usually formulated for sequential programs does not use (or need) conjunctions, precisely because of the single-threadedness property.

With these considerations in mind, we are now ready to present an assertion language.

Definition 3.1 The class of *transition assertions* (over a given first-order condition language) is given by the following grammar:

$$\phi ::= P\sum_{i=1}^{n}\alpha_i P_i \phi_i \mid P\circ \mid P\bullet \mid \theta \mid \mu\theta.\phi \mid \phi_1\&\phi_2,$$

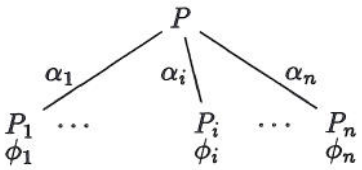
where P and the P_i are conditions, θ is a meta-variable ranging over transition assertions, and the α_i are atomic actions. •

Assertions of the form $P\bullet$ and $P\circ$ will be called terminal assertions; we use • to represent termination and ∘ to represent deadlock. An assertion of the form $\mu\theta.\phi$ is called *recursive*; μ is a binding operator. We will mainly be concerned with *closed* assertions, *i.e.* assertions containing no free assertion variables.

Our notation for assertions is similar to Milner's linear notation for synchronization trees, but with conditions built into the tree structure and with two types of NIL tree[2]. For presentation

[2] Like Milner, we regard the sum notation as denoting a *set* of branches, so that we treat as equivalent assertions such as $P(\alpha Q\phi+\alpha Q\phi+\beta R\psi)$ and $P(\beta R\psi+\alpha Q\phi)$.

purposes, we will sometimes prefer to draw an assertion as a tree, or as a DAG when several branches lead to identical sub-assertions. For example, the generic assertion $P\sum_{i=1}^{n}\alpha_i P_i \phi_i$ may be represented as:



3.1 Validity

In order to express the property that a command C *satisfies* a (closed) assertion ϕ we write

$$C \textbf{ sat } \phi.$$

When ϕ is the assertion $P\sum_{i=1}^{n}\alpha_i P_i \phi_i$ we interpret C **sat** ϕ in the following way. If the command is started in a state satisfying P, then its initial action must be an α_i drawn from the set of initial labels of the assertion, and these labels are precisely the initial actions possible for the command. If the command starts with an α_i action it reaches a state where P_i is true and where the remaining command satisfies ϕ_i. Formally, we say that

$$\models C \textbf{ sat } P\sum_{i=1}^{n}\alpha_i P_i \phi_i$$

iff, for all s, α, C', s':

$$(s \models P \ \& \ \langle C, s\rangle \xrightarrow{\alpha} \langle C', s'\rangle \ \Rightarrow \ \exists i \leq n.\ \alpha = \alpha_i \ \& \ s' \models P_i \ \& \ C' \models \phi_i), \tag{1}$$

and, for all s, i,

$$(s \models P \ \Rightarrow \ \exists C_i, s_i.\ \langle C, s\rangle \xrightarrow{\alpha_i} \langle C_i, s_i\rangle), \tag{2}$$

so that all of the actions specified in ϕ are indeed possible for C when the initial state satisfies P.

We interpret the two terminal cases as follows:

$$\models C \textbf{ sat } P\bullet \quad \Longleftrightarrow \quad C = \textbf{null},$$

since **null** represents unequivocal termination; and

$$\models C \textbf{ sat } P\circ \quad \Longleftrightarrow \quad \forall s.(s \models P \Rightarrow \text{stuck}\langle C, s\rangle),$$

since deadlock depends on the current state.

We interpret a recursive assertion $\mu\theta.\phi$ as follows. Let

$$\begin{aligned} \theta_0 &= \{\textbf{false}\}\bullet, \\ \text{and } \theta_{n+1} &= [\theta_n \setminus \theta]\phi \quad \text{for } n \geq 0. \end{aligned}$$

Each θ_n is obtained by unfolding the recursive assertion a finite number of times. The logical properties of $\mu\theta.\phi$ are uniquely determined from those of its unfoldings: we specify that

$$\models C \textbf{ sat } \mu\theta.\phi \quad \Longleftrightarrow \quad \forall n \geq 0.(\models C \textbf{ sat } \theta_n).$$

Finally, $\models C$ **sat** $(\phi_1 \ \& \ \phi_2)$ iff $\models C$ **sat** ϕ_1 and $\models \ C$ **sat** ϕ_2.

3.2 Some Examples

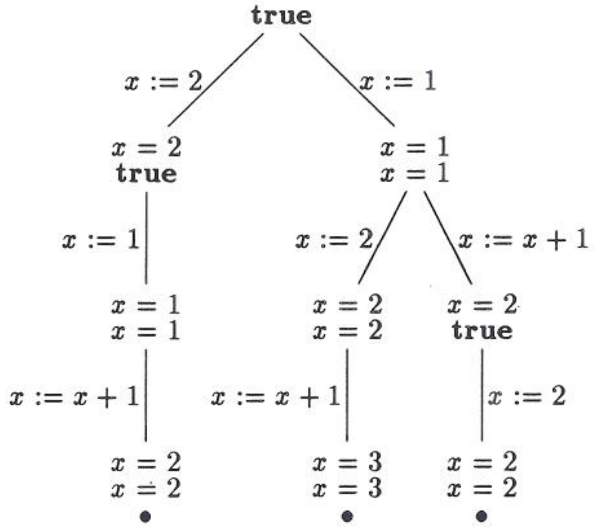
1. The command $[x{:=}x+1 \| x{:=}x+1]$ satisfies the assertion

$$\{x=0\}\, x{:=}x+1\, \{x=1\}\, \{x=1\}\, x{:=}x+1\, \{x=2\}\, \{x=2\} \bullet$$

It also satisfies

$$\{x=0\}\, x{:=}x+1\, \{x=1\}\, \{x=99\}\, x{:=}x+1\, \{x=100\}\, \{\mathbf{true}\} \bullet .$$

2. The command $[x{:=}2 \| (x{:=}1; x{:=}x+1)]$ satisfies the assertion



3. The loop **while** $x > 0$ **do** $x{:=}x-1$ satisfies the recursive assertion

$$\mu\theta.\, \{x>0\}\, x>0\, \{x>0\}\, \{x>0\}\, x{:=}x-1\, \{x \geq 0\}\, \theta$$

and also the assertion $\{x \leq 0\}\, x>0\, \{x \leq 0\}\, \{\mathbf{true}\} \bullet$. It also satisfies the assertion

$$\begin{array}{ll}\mu\theta.(& \{x>0\}\, x>0\, \{x>0\}\, \{x>0\}\, x{:=}x-1\, \{x \geq 0\}\, \theta \\ \quad \& & \{x \leq 0\}\, x>0\, \{x \leq 0\}\, \{\mathbf{true}\} \bullet).\end{array}$$

3.3 Parallel composition of assertions

We now define a syntactic parallel composition for assertions. The definition is given inductively.

When one of the assertions is terminal, we specify that

$$\begin{array}{rcl}[P \circ \| Q \sum_{j=1}^{m} \beta_j Q_j \psi_j] & = & \{P\&Q\} \sum_{j=1}^{m} \beta_j Q_j [P \circ \| \psi_j], \\ {[P \bullet \| Q \sum_{j=1}^{m} \beta_j Q_j \psi_j]} & = & \{P\&Q\} \sum_{j=1}^{m} \beta_j Q_j [P \bullet \| \psi_j],\end{array}$$

and similarly when the two terms are exchanged. When both assertions are terminal we put

$$\begin{array}{l}P \bullet \| Q \bullet = \{P\&Q\} \bullet \\ P \circ \| Q \bullet = P \bullet \| Q \circ = P \circ \| Q \circ = \{P\&Q\} \circ .\end{array}$$

These definitions express the fact that, according to the operational semantics, a parallel program terminates when all of its components have terminated and deadlocks when all of its components deadlock. Note for instance that, for all ψ,

$$[\{\mathbf{true}\} \bullet \| \psi] \ = \ [\psi \| \{\mathbf{true}\} \bullet] \ = \ \psi.$$

(Strictly speaking, these are logical equivalences rather than syntactic identities.) This corresponds to the fact that once a parallel component of a program terminates successfully the rest of the program can continue executing.

The inductive clause is an extension of the *interleaving* operation on synchronization trees [6,24,28], modified to handle the conditions and conjuncts in an appropriate manner. For assertions ϕ and ψ of the form

$$\phi \ = \ P(\sum_{i=1}^{n} \alpha_i P_i \phi_i), \quad \psi \ = \ Q(\sum_{j=1}^{m} \beta_j Q_j \psi_j),$$

we define

$$[\phi \| \psi] \ = \ \{P \& Q\} \, (\sum_{i=1}^{n} \alpha_i P_i [\phi_i \| \psi] + \sum_{j=1}^{m} \beta_j Q_j [\phi \| \psi_j]).$$

Note that as far as the action sequences are concerned the operation corresponds exactly to interleaving of trees.

We extend this definition to deal with conjunction as follows. When ϕ and ψ are conjunctions, $[\phi \| \psi]$ is defined to be a conjunction. For each conjunct $P \sum_{i=1}^{n} \alpha_i P_i \phi_i$ of ϕ and each conjunct $Q \sum_{j=1}^{m} \beta_j Q_j \psi_j$ of ψ we include in $[\phi \| \psi]$ a conjunct of the form:

$$\{P \& Q\} \, (\sum_{i=1}^{n} \alpha_i P_i [\phi_i \| \psi] \ + \ \sum_{j=1}^{m} \beta_j Q_j [\phi \| \psi_j]).$$

When ϕ and ψ are simple assertions this is exactly the same definition as before.

The following result shows that parallel composition of assertions does indeed have the correct effect: if C_1 satisfies ϕ and C_2 satisfies ψ then $[C_1 \| C_2]$ satisfies $[\phi \| \psi]$.

Theorem 3.2 *If* $\models C_1$ **sat** ϕ *and* $\models C_2$ **sat** ψ *then* $\models [C_1 \| C_2]$ **sat** $[\phi \| \psi]$.

3.4 Sequential composition of assertions

We also define a sequential composition $\phi; \psi$ for assertions, whose effect is to graft a copy of ψ onto the terminated nodes of ϕ. The definition is again inductive.

In the base cases, we put

$$P \bullet; \ Q \sum_{j=1}^{m} \beta_j Q_j \psi_j \ = \ \{P \& Q\} \sum_{j=1}^{m} \beta_j Q_j \psi_j$$

and

$$P \circ; \ \psi \ = \ P \circ .$$

When ϕ is $P(\sum_{i=1}^{n} \alpha_i P_i \phi_i)$ we put

$$\phi; \psi \ = \ P \sum_{i=1}^{n} \alpha_i P_i (\phi_i; \psi).$$

To handle conjunctions we let $(\phi_1 \& \phi_2);\psi \;=\; (\phi_1;\psi)\&(\phi_2;\psi)$ and similarly when we have a conjunction in the second place: in other words, sequential composition distributes over conjunction.

Again it is easy to show that this syntactic operation on assertions correctly models sequential composition of programs.

Theorem 3.3 *If* $\models C_1$ **sat** ϕ *and* $\models C_2$ **sat** ψ *then* $\models\; (C_1;C_2)$ **sat** $(\phi;\psi)$.

3.5 Implication

We define a notion of *implication* $\phi \Rightarrow \psi$ between assertions, with the intention that whenever ϕ implies ψ then any command satisfying ϕ will also satisfy ψ. The definition relies on the usual form of implication $P \Rightarrow Q$ between conditions. The main idea is that ϕ implies ψ when ψ arises from ϕ by strengthening pre-conditions and/or weakening post-conditions. This is closely related to the familiar Rule of Consequence in Hoare's logic for while-programs, which is usually written in the form:

$$\frac{P' \Rightarrow P \quad \{P\}C\{Q\} \quad Q \Rightarrow Q'}{\{P'\}C\{Q'\}}$$

This rule can equally well be viewed as a rule of *modus ponens* for pca's, if we define a form of implication on pca's by:

$$\{P\}C\{Q\} \;\Rightarrow\; \{P'\}C\{Q'\} \quad\Longleftrightarrow\quad (P' \Rightarrow P)\ \&\ (Q \Rightarrow Q').$$

The following definitions adapt this idea to our assertion language.

For $\phi = P\sum_{i=1}^{n}\alpha_i P_i \phi_i$ and $\psi = Q\sum_{i=1}^{n}\alpha_i Q_i \psi_i$, we define:

$$(\phi \Rightarrow \psi) \quad\Longleftrightarrow\quad (Q \Rightarrow P)\ \&\ \bigwedge_{i=1}^{n}(P_i \Rightarrow Q_i)\ \&\ \bigwedge_{i=1}^{n}(\phi_i \Rightarrow \psi_i).$$

For the base cases, we define

$$\begin{array}{lcl} P\bullet \Rightarrow Q\bullet & \Longleftrightarrow & Q \Rightarrow P, \\ P\circ \Rightarrow Q\circ & \Longleftrightarrow & Q \Rightarrow P. \end{array}$$

We extend implication to conjunctions in the obvious way:

$$\begin{array}{c} (\phi_1\ \&\ \phi_2) \Rightarrow \phi_1 \\ (\phi_1\ \&\ \phi_2) \Rightarrow \phi_2 \end{array}$$

For a recursive assertion we define:

$$\mu\theta.\phi \iff [(\mu\theta.\phi)\setminus\theta]\phi,$$

where $[\psi\setminus\theta]\phi$ denotes the result of substituting ψ for each free occurrence of θ in ϕ, with renaming of bound variables if necessary to avoid capturing any of the free assertion variables of ψ.

We also employ the following inference rule, based on the usual kind of μ-induction rule (see for instance [Sco76, Par70]):

$$\frac{\psi \Rightarrow \theta \;\vdash\; \psi \Rightarrow \phi}{\psi \Rightarrow \mu\theta.\phi}$$

where ψ is closed and θ may occur free in ϕ.

The following *forward propagation* rule is useful for strengthening an assertion, by using partial correctness properties of its atomic actions:

$$\frac{\{P\&Q\}\,\alpha_i\,\{Q_i\}\,,\quad i=1,\ldots,n}{P\sum_{i=1}^{n}\alpha_i P_i\phi_i \;\Rightarrow\; \{P\&Q\}\sum_{i=1}^{n}\alpha_i\,\{P_i\&Q_i\}\,\phi_i}$$

This rule may be applied either locally, at a particular sub-assertion of a general assertion ϕ, or systematically, beginning by conjoining a condition Q to the root node of ϕ and repeating the operation with condition $P_i\&Q_i$ on the ϕ_i, and so on recursively.

The following result shows that this form of implication behaves logically as expected:

Theorem 3.4 *For all C, ϕ and ψ, if $\models C$ **sat** ϕ and $\phi \Rightarrow \psi$, then $\models C$ **sat** ψ.*

3.6 Safe assertions

In the syntactic definition of the class of transition assertions, we did not require that any logical connection exist between adjacent "intermediate" conditions inside an assertion. Assertions in which every internal post-condition logically implies the immediately adjacent condition can be regarded as describing the behavior of a command executed in isolation, since intuitively the command can progress along the paths described by the assertion without requiring interference or outside intervention. We will call such assertions *safe*. There are good semantic reasons for not making this constraint part of the syntactic definition of our assertion language, since in general information solely about the safe assertions satisfied by a program may be insufficient to characterize the behavior of that program in all parallel contexts. Nevertheless, since safe assertions are closely related to pcda's we find it convenient to subject them to some special treatment.

Definition 3.5 For each assertion ϕ the "root pre-condition" root(ϕ) is defined by:

$$\begin{array}{l}
\mathrm{root}(P\sum_{i=1}^{n}\alpha_i P_i\phi_i) \;=\; P\\
\mathrm{root}(P\circ) = \mathrm{root}(P\bullet) \;=\; P\\
\mathrm{root}(\phi_1\&\phi_2) = \mathrm{root}(\phi_1) \vee \mathrm{root}(\phi_2)\\
\mathrm{root}(\mu\theta.\phi) = \mathrm{root}(\phi)\\
\mathrm{root}(\theta) = \mathbf{false}
\end{array}$$

The "termination post-condition" term(ϕ) and "deadlock post-condition" dead(ϕ) are defined by:

$$\begin{array}{ll}
\mathrm{term}(P\sum_{i=1}^{n}\alpha_i P_i\phi_i) \;=\; \bigvee_{i=1}^{n}\mathrm{term}(\phi_i) & \mathrm{dead}(P\sum_{i=1}^{n}\alpha_i P_i\phi_i) = \bigvee_{i=1}^{n}\mathrm{dead}(\phi_i)\\
\mathrm{term}(P\circ) = \mathbf{false} & \mathrm{dead}(P\circ) = P\\
\mathrm{term}(P\bullet) = P & \mathrm{dead}(P\bullet) = \mathbf{false}\\
\mathrm{term}(\phi_1\&\phi_2) = \mathrm{term}(\phi_1) \vee \mathrm{term}(\phi_2) & \mathrm{dead}(\phi_1\&\phi_2) = \mathrm{dead}(\phi_1) \vee \mathrm{dead}(\phi_2)\\
\mathrm{term}(\theta) = \mathbf{false} & \mathrm{dead}(\theta) = \mathbf{false}\\
\mathrm{term}(\mu\theta.\phi) = \mathrm{term}(\phi) & \mathrm{dead}(\mu\theta.\phi) = \mathrm{dead}(\phi)
\end{array}$$

•

Informally, root(ϕ) is the pre-condition at the start of ϕ, term(ϕ) is the disjunction of the conditions at terminated nodes of ϕ, and dead(ϕ) is the disjunction of the conditions at deadlocked nodes of ϕ.

A terminal assertion is safe, and $P\sum_{i=1}^{n}\alpha_i P_i\phi_i$ is safe iff for each i, $P_i \Rightarrow \mathrm{root}(\phi_i)$ and ϕ_i is safe. A recursive assertion $\mu\theta.\phi$ is safe if the condition immediately preceding each occurrence of θ implies $\mathrm{root}(\phi)$ and ϕ is safe at all other nodes.

A precise definition of what it means for an assertion to be safe follows:

Definition 3.6 The predicate *safe* on assertions is the least predicate satisfying:

$$\vdash \mathrm{safe}(P\circ)$$

$$\vdash \mathrm{safe}(P\bullet)$$

$$\frac{\bigwedge_{i=1}^{n}(P_i \Rightarrow \mathrm{root}(\phi_i)) \quad \bigwedge_{i=1}^{n}\mathrm{safe}(\phi_i)}{\mathrm{safe}(P\sum_{i=1}^{n}\alpha_i P_i\phi_i)}$$

$$\frac{\mathrm{safe}(\phi_1) \quad \mathrm{safe}(\phi_2)}{\mathrm{safe}(\phi_1 \& \phi_2)}$$

$$\frac{\mathrm{safe}([\mathrm{root}(\phi)\bullet\backslash\theta]\phi)}{\mathrm{safe}(\mu\theta.\phi)}$$

●

Safe assertions are fundamental in reasoning about conditional critical regions, because when **await** B **then** $\langle C\rangle$ is executed from a state in which B is true, the effect is to execute C without allowing interruption. Thus, whenever $\models C$ **sat** ϕ and $\mathrm{safe}(\phi)$, the command **await** B **then** $\langle C\rangle$ will satisfy the assertion

$$\{\mathrm{root}(\phi)\ \&\ B\}\,\langle C\rangle\,\{\mathrm{term}(\phi)\}\,\{\mathbf{true}\}\bullet.$$

Safe assertions are of particular importance since they can be used directly to derive partial correctness and deadlock assertions, as shown by the following result. The proof is straightforward.

Theorem 3.7 *If* $\models C$ **sat** ϕ *and* $\mathit{safe}(\phi)$*, then* $\models \{\mathit{root}(\phi)\}\,C\,\{\mathit{term}(\phi)\}\,[\mathit{dead}(\phi)]$.

Note that parallel composition is not guaranteed to produce safe assertions, even if the component assertions are safe. The same holds for sequential composition. Nevertheless, the following simple result is useful:

Theorem 3.8 *If* ϕ *and* ψ *are safe and* $(\mathit{term}(\phi) \Rightarrow \mathit{root}(\psi))$ *then* $\phi;\psi$ *is safe.*

It is obvious that an assertion in which every internal post-condition is identical to the adjacent pre-condition will be safe. There is clearly some redundancy in using the full syntax given earlier for assertions like this. It is therefore convenient to adopt an abbreviated notation for this sub-class of the safe assertions, in which we only mention each internal condition once. We therefore introduce the following syntax.

Definition 3.9 The class of abbreviated safe assertions, ranged over by meta-variable σ, is defined by the following grammar:

$$\sigma ::= P\sum_{i=1}^{n} \alpha_i \sigma_i \mid P\circ \mid P\bullet \mid \theta \mid \mu\theta.\sigma \mid \sigma_1 \& \sigma_2,$$

in which P and the P_i are conditions. $\bullet$

3.7 Program skeletons

Every program has a characteristic safe assertion that describes the possible control paths through the program, using conjunction to deal with cases where the flow of control may be affected by interference. We call this kind of assertion a *program skeleton.*

Definition 3.10 The skeleton skel(C) of a program C is defined, by induction on the syntactic structure of C, by:

$$\begin{array}{rl}
\mathrm{skel}(\mathbf{skip}) = & \{\mathbf{true}\}\mathbf{skip}\{\mathbf{true}\}\{\mathbf{true}\}\bullet \\
\mathrm{skel}(I{:=}E) = & \{\mathbf{true}\}I{:=}E\{\mathbf{true}\}\{\mathbf{true}\}\bullet \\
\mathrm{skel}(C_1;C_2) = & \mathrm{skel}(C_1);\mathrm{skel}(C_2) \\
\mathrm{skel}([C_1\|C_2]) = & \mathrm{skel}(C_1)\|\mathrm{skel}(C_2) \\
\mathrm{skel}(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2) = & \{B\}B\{\mathbf{true}\}\mathrm{skel}(C_1) \\
& \quad \& \\
= & \{\neg B\}B\{\mathbf{true}\}\mathrm{skel}(C_2) \\
\mathrm{skel}(\mathbf{await}\ B\ \mathbf{then}\ \langle C\rangle) = & \{B\}\langle C\rangle\{\mathbf{true}\}\{\mathbf{true}\}\bullet \\
& \quad \& \\
= & \{\neg B\}\circ \\
\mathrm{skel}(\mathbf{while}\ B\ \mathbf{do}\ C) = & \mu\theta.[\{B\}B\{\mathbf{true}\}(\mathrm{skel}(C);\theta) \\
& \qquad\qquad \& \\
& \qquad \{\neg B\}B\{\mathbf{true}\}\{\mathbf{true}\}\bullet]
\end{array}$$

$\bullet$

The following result, easily proved by induction on C, is indicative of the utility of program skeletons.

Theorem 3.11 *For every command C, the assertion skel(C) is safe and is satisfied by C.*

3.8 Examples, revisited

We return briefly to some assertions discussed earlier.

1. The assertion

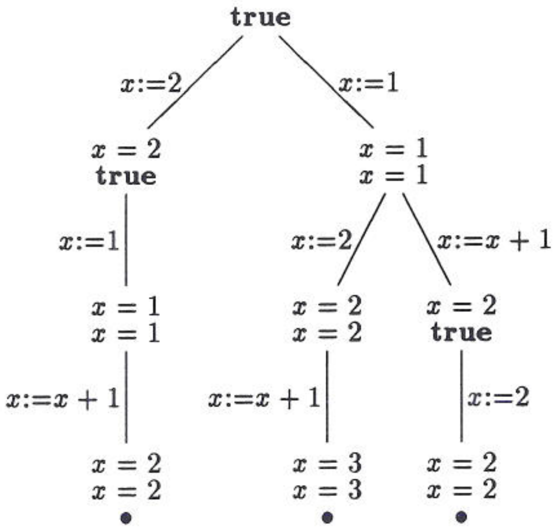
   $$\{x = 0\}x{:=}x + 1\{x = 1\}\{x = 1\}x{:=}x + 1\{x = 2\}\{x = 2\}\bullet$$

   is safe, has root condition $x = 0$, termination condition $x = 2$, and deadlock condition **false**. There is also an obvious abbreviated form for this assertion. In contrast, the assertion

   $$\{x = 0\}x{:=}x + 1\{x = 1\}\{x = 99\}x{:=}x + 1\{x = 100\}\{\mathbf{true}\}\bullet$$

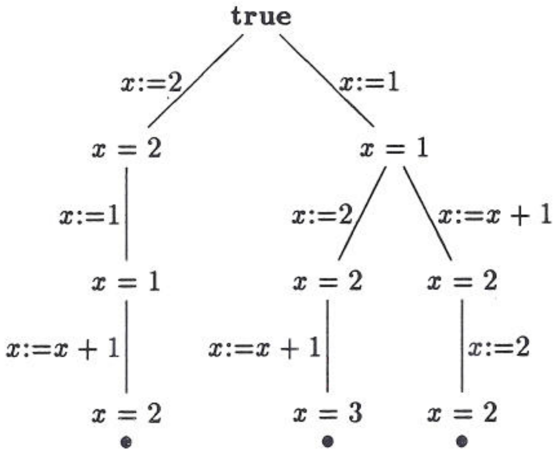
   is not safe, but is nevertheless satisfied by $x{:=}x + 1; x{:=}x + 1$.

2. The assertion

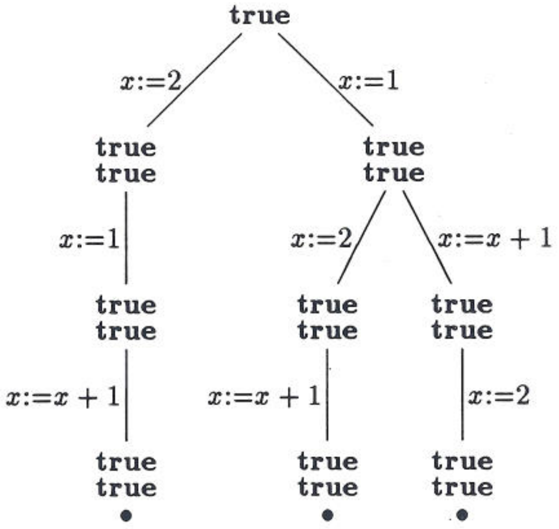


is safe, has root **true**, termination condition $x = 2 \vee x = 3$, and deadlock condition **false**. At the internal nodes where adjacent conditions do not coincide the reason is that the next step behavior of the program once it has reached the corresponding point in its execution does not depend on there being no interference before the program resumes; hence the use of **true** as the "resumption pre-condition".

An abbreviated form of safe assertion satisfied by this command is:



3. The skeleton of the command $[x{:=}2 \| (x{:=}1; x{:=}x+1)]$ is the assertion:



The safe assertions given above for this command may be easily derived by forward propagation.

4. The assertion

$$\mu\theta.(\{x > 0\}x > 0\{x > 0\}\{x > 0\}x{:=}x - 1\{x \geq 0\}\theta)$$

is not safe, because $x \geq 0$ does not imply $x > 0$. However, the assertion

$$\begin{aligned} \mu\theta.(\quad & \{x > 0\}\, x > 0\, \{x > 0\}\, \{x > 0\}\, x{:=}x - 1\, \{x \geq 0\}\, \theta \\ \& \quad & \{x \leq 0\}\, x > 0\, \{x \leq 0\}\, \{\mathbf{true}\}\, \bullet) \end{aligned}$$

is safe. Its root is $x > 0 \vee x \leq 0$, its termination condition is **true**, and its deadlock condition is $\mathtt{ff}$.

3.9 A Proof System for Transition Assertions

Using the definitions and results accumulated so far, we are led to the collection of axioms and inference rules summarized in Figure 3. Validity of each axiom is clear, the soundness of most of the inference rules has already been discussed, and the remaining cases (for instance, dealing with if-then-else) are obviously sound. These rules and axioms are based closely on the operational semantics presented in Figure 1.

$$\vdash\ \textbf{skip sat}\ \ \{P\}\,\textbf{skip}\,\{P\}\,\{\textbf{true}\}\,\bullet$$

$$\vdash\ I{:=}E\ \textbf{sat}\ \ \{[E \setminus I]P\}\,I{:=}E\,\{P\}\,\{\textbf{true}\}\,\bullet$$

$$\vdash\ \textbf{await}\ B\ \textbf{then}\ \langle C\rangle\ \textbf{sat}\ \ \{\neg B\}\,\circ$$

$$\frac{C\ \textbf{sat}\ \phi \quad \mathrm{safe}(\phi)}{\textbf{await}\ B\ \textbf{then}\ \langle C\rangle\ \textbf{sat}\ \ \{\mathrm{root}(\phi)\&B\}\,\langle C\rangle\,\{\mathrm{term}(\phi)\}\,\{\textbf{true}\}\,\bullet}$$

$$\frac{C_1\ \textbf{sat}\ \phi_1}{\textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{sat}\ \{P\&B\}\,B\,\{P\}\,\phi_1}$$

$$\frac{C_2\ \textbf{sat}\ \phi_2}{\textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{sat}\ \ \{P\&\neg B\}\,B\,\{P\}\,\phi_2}$$

$$\frac{C\ \textbf{sat}\ \phi}{\textbf{while}\ B\ \textbf{do}\ C\ \textbf{sat}\ \mu\theta.[\{P\&B\}\,B\,\{P\}\,(\phi;\theta)\ \&\ \ \{P\&\neg B\}\,B\,\{P\}\,\{\textbf{true}\}\,\bullet]}$$

$$\frac{C_1\ \textbf{sat}\ \phi \qquad C_2\ \textbf{sat}\ \psi}{[C_1\|C_2]\ \textbf{sat}\ [\phi\|\psi]}$$

$$\frac{C_1\ \textbf{sat}\ \phi \quad C_2\ \textbf{sat}\ \psi}{C_1;C_2\ \textbf{sat}\ \phi;\psi}$$

$$\frac{C\ \textbf{sat}\ \phi \quad C\ \textbf{sat}\ \psi}{C\ \textbf{sat}\ (\phi\&\psi)}$$

$$\frac{C\ \textbf{sat}\ \phi \quad \phi \Rightarrow \psi}{C\ \textbf{sat}\ \psi}$$

$$\frac{C\ \textbf{sat}\ \phi \quad \mathrm{safe}(\phi)}{\{\mathrm{root}(\phi)\}\,C\,\{\mathrm{term}(\phi)\}\,[\mathrm{dead}(\phi)]}$$

Figure 3: Axioms and Proof Rules for Transition Assertions

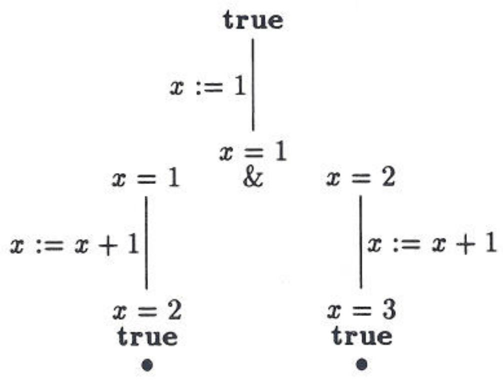
4 Examples

We begin with a few very simple example programs to illustrate the ideas.

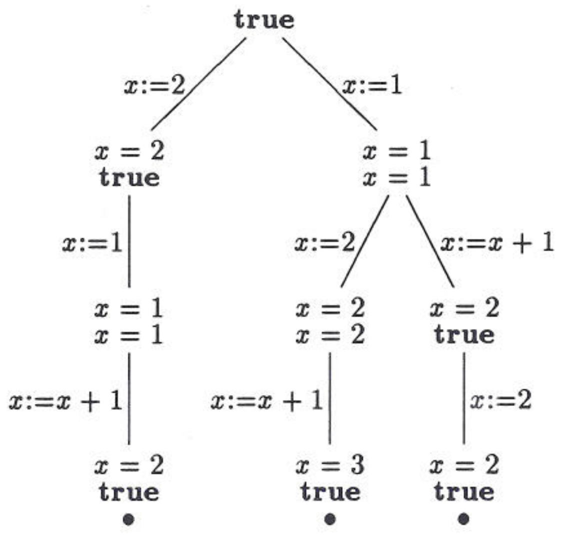
1. We wish to prove that $[x{:=}x+1 \| x{:=}x+1]$ satisfies the assertion
$$\{x=0\}\,x{:=}x+1\,\{x=1\}\,\{x=1\}\,x{:=}x+1\,\{x=2\}\,\{x=2\}\bullet.$$
It is easy to prove the following assertion for $x{:=}x+1$:
$$\begin{array}{ll} & \{x=0\}\,x{:=}x+1\,\{x=1\}\,\{x=1\}\bullet \\ \& & \{x=1\}\,x{:=}x+1\,\{x=2\}\,\{x=2\}\bullet \end{array}$$
The parallel composition of this assertion (say, ϕ) with itself has four conjuncts, one of which is logically equivalent to:
$$\{x=0\}\,x{:=}x+1\,\{x=1\}\,\phi.$$
But $\phi \;\Rightarrow\; \{x=1\}\,x{:=}x+1\,\{x=2\}\,\{x=2\}\bullet$. Hence, $[\phi\|\phi]$ implies the desired assertion.

2. We can prove that the command $x{:=}x+1; x{:=}x+1$ satisfies the assertion
$$\{x=0\}\,x{:=}x+1\,\{x=1\}\,\{x=99\}\,x{:=}x+1\,\{x=100\}\,\{\mathbf{true}\}\bullet,$$
by forming the sequential composition of the assertions
$$\{x=0\}\,x{:=}x+1\,\{x=1\}\,\{\mathbf{true}\}\bullet, \qquad \{x=99\}\,x{:=}x+1\,\{x=100\}\,\{\mathbf{true}\}\bullet.$$

3. It is easy to prove that $x{:=}2$ satisfies the assertion
$$\{\mathbf{true}\}x{:=}2\{x=2\}\{\mathbf{true}\}\bullet.$$
Similarly, $x{:=}1$ satisfies
$$\{\mathbf{true}\}x{:=}1\{x=1\}\{\mathbf{true}\}\bullet$$
and, using the rule for &-introduction, $x{:=}x+1$ satisfies
$$\{x=1\}x{:=}x+1\{x=2\}\{\mathbf{true}\}\bullet \;\&\; \{x=2\}x{:=}x+1\{x=3\}\{\mathbf{true}\}\bullet$$
By the rule for sequential composition it follows that $x{:=}1;\ x{:=}x+1$ satisfies the assertion:



The parallel composition of these assertions implies the following assertion, which can therefore be derived for the program $[x{:=}2 \| (x{:=}1;\ x{:=}x+1)]$:



This assertion obviously implies the assertion given earlier for the same program.

Now we progress to some more complicated examples whose correctness proofs require larger and more intricate assertions. These examples are based on classic problems in parallel programming: coordination of producer and consumer processes, guaranteeing mutual exclusion, and partition of a set. For the origins of these problems and classic programming solutions see for instance [Dij68] and the more recent discussions in [Bar85, BA82, AO91].

4.1 Producer and Consumer

Suppose a producer process and a consumer process interact by means of a buffer, modelled as a queue on which insertion and deletion operations may be performed. We discuss a parallel program using semaphores, based on Dijkstra's solution to the producer-consumer problem in [Dij68]. A semaphore, as introduced by Dijkstra, is an integer-valued variable, say s, on which the only allowed operations after initialization are:

$$\begin{aligned} P(s) &= \textbf{await } s > 0 \textbf{ then } s{:=}s-1, \\ V(s) &= s{:=}s+1. \end{aligned}$$

A binary semaphore is a semaphore whose value is constrained to be either 0 or 1.

Our program uses three semaphores e, f, and b, of which b is binary. Intuitively, $e > 0$ means that the buffer is not empty, $f > 0$ means that the buffer is not full, and the value of b is either 0 (buffer being accessed) or 1 (buffer not being accessed). Suppose the protocols by means of which the processes gain access to the buffer are:

```
CON ::    await $e > 0$ then $e := e - 1$;
          await $b > 0$ then $b := b - 1$;
          $rem(x)$;
          $f := f + 1$;
          $b := b + 1$

PRO ::    await $f > 0$ then $f := f - 1$;
          await $b > 0$ then $b := b - 1$;
          $ins(y)$;
          $e := e + 1$;
          $b := b + 1$
```

This program design is based on the idea that a consumer should wait until the buffer is not empty before trying to gain exclusive access to the buffer for removing an item, and a producer should likewise wait until the buffer is not full before requesting access. The details of how the buffer and its insertion and deletion operations are implemented do not matter for this analysis.

If we execute CON$\|$PRO from an initial state in which the buffer is available ($b = 1$), full ($f = 0$) and not empty ($e > 0$), the producer must wait until the consumer has finished removing an item before gaining access for an insertion. This property is expressed by the safe assertion in Figure 4, which is presented in the abbreviated notation and uses comma for conjunction of conditions.

This assertion may itself be proven by forming the parallel composition of assertions ϕ_{CON} (Figure 5) and ϕ_{PRO} (Figure 6), which are easy to prove for CON and PRO respectively. Again the figures use the abbreviated notation.

The pcda

$$\{e > 0 \;\&\; f = 0 \;\&\; b = 1\}\mathrm{CON}\|\mathrm{PRO}\{e > 0 \;\&\; f = 0 \;\&\; b = 1\}[\mathbf{false}]$$

is derivable, showing absence of deadlock.

However, if the producer uses a poorly designed protocol such as:

```
PRO′ ::   await $b > 0$ then $b := b - 1$;
          await $f > 0$ then $f := f - 1$;
          $ins(y)$;
          $e := e + 1$;
          $b := b + 1$
```

there is a possible deadlock, caused when the producer grabs access to the buffer even though the buffer is still full, thereby preventing the consumer from gaining access. This is shown by the safe assertion in Figure 7. We leave it as an exercise to find assertions for PRO and CON′ whose parallel composition implies this assertion. Clearly the pcda

$$\{e > 0 \;\&\; f = 0 \;\&\; b = 1\}\mathrm{CON}\|\mathrm{PRO}'\{e > 0 \;\&\; f = 0 \;\&\; b = 1\}[e > 0 \;\&\; f = 0 \;\&\; b = 0]$$

is derivable.

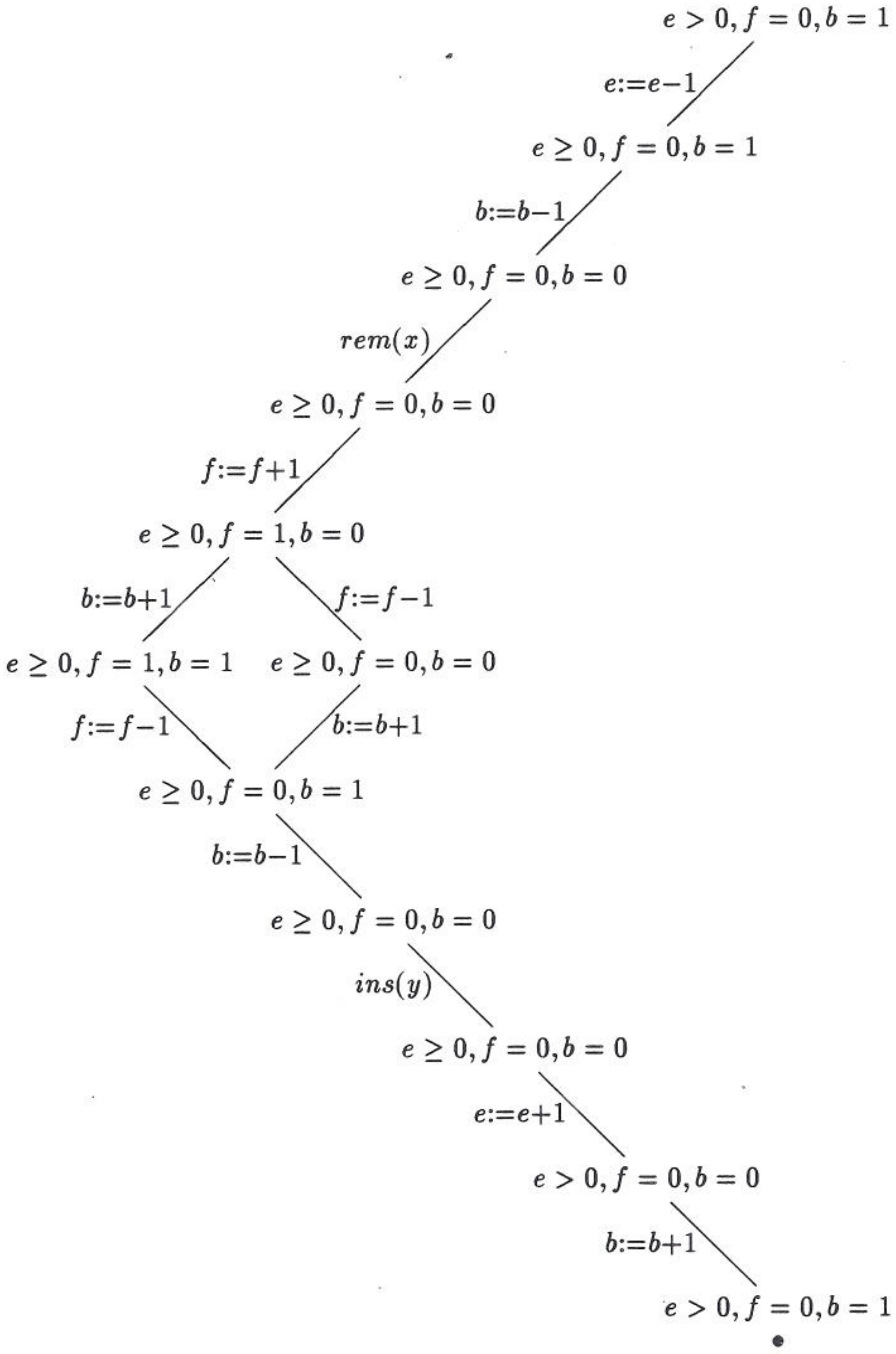


Figure 4: An assertion satisfied by CON||PRO

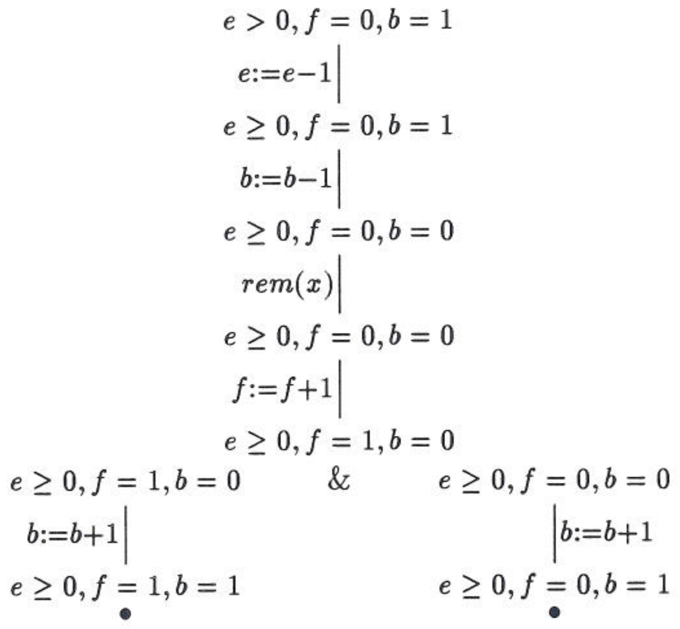


Figure 5: An assertion ϕ_{CON} satisfied by CON

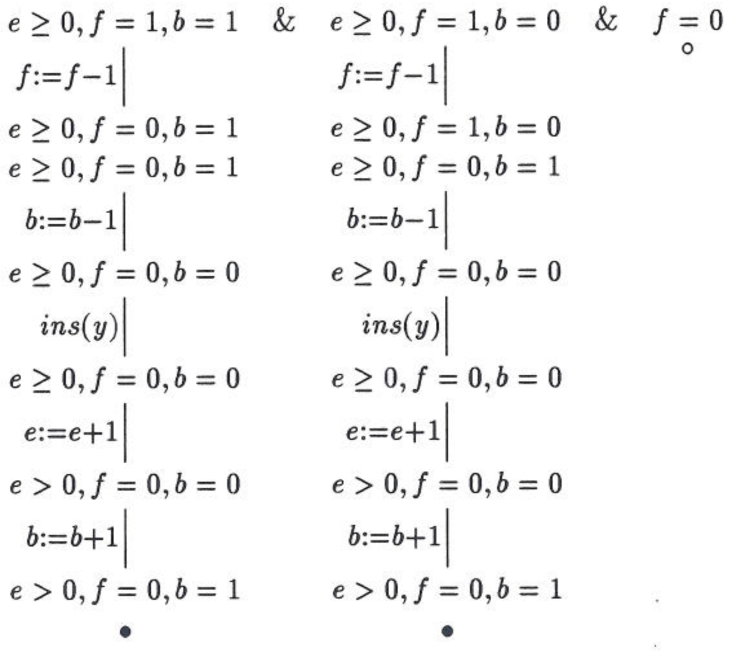


Figure 6: An assertion ϕ_{PRO} satisfied by PRO

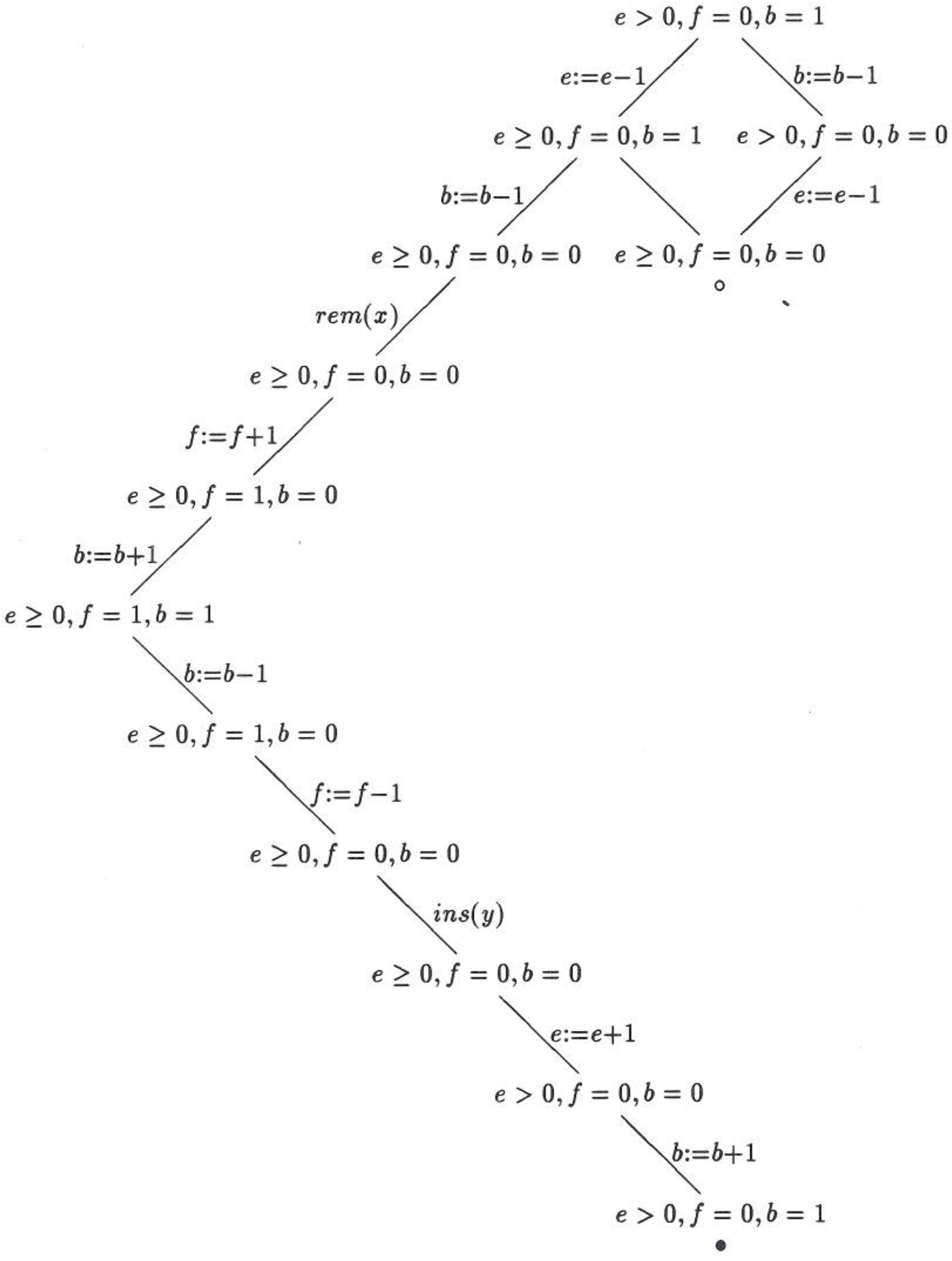


Figure 7: An assertion satisfied by CON||PRO′

4.2 Peterson's algorithm for mutual exclusion

This example is based on Peterson's mutual exclusion algorithm [Pet81], in which two processes repeatedly execute a loop containing a critical section and we must guarantee that they are never simultaneously inside their critical sections. A correctness proof for a version of this algorithm, given in the Owicki-Gries style, occurs in [AO91].

In Peterson's algorithm, each process sets a boolean "readiness" variable (here r_1, respectively, r_2) when it intends to enter its critical section then temporarily gives the other process a chance to enter (by setting the shared variable t), and waits until either it has the right to enter or the other process is not ready. Upon leaving the critical section a process signals that it is no longer ready, and begins the cycle again. The value of t indicates which process gets to enter its critical section in case both are ready simultaneously.

We consider processes whose critical sections are $x{:=}x+1; x{:=}x+1$ and $y{:=}x$ respectively, so we are treating the variable x as a critical resource. We focus on one parallel iteration of the two processes, after they have both set their readiness flag and t is initially 1. It would be straightforward to extend our analysis to the full algorithm, which corresponds to the parallel composition of two while-loops.

Consider, then, the execution of the program fragment LEFT||RIGHT, where

```
LEFT ::    t := 2;
           await ¬r₂ ∨ (t = 1);
           x := x + 1;
           x := x + 1;
           r₁ := false

RIGHT::    t:=1;
           await ¬r₁ ∨ (t = 2);
           y := x;
           r₂ := false
```

and where we assume that initially r_1 and r_2 have been set to **true** and $t = 1$.

The assertion for LEFT||RIGHT given in Figure 8 can be proven. In the diagram we use the abbreviations e_x and e_y for $even(x)$ and $even(y)$ respectively. From this (safe) assertion the pcda

$$\{r_1 \ \& \ r_2 \ \& \ t = 1 \ \& \ even(x)\}\,[\mathrm{LEFT}\|\mathrm{RIGHT}]\,\{\neg r_1 \ \& \ \neg r_2 \ \& \ even(y)\}\,[\mathbf{false}]$$

is easily derivable. Note that if mutual exclusion were violated it would be possible for the program to terminate with y odd. We have also proven absence of deadlock.

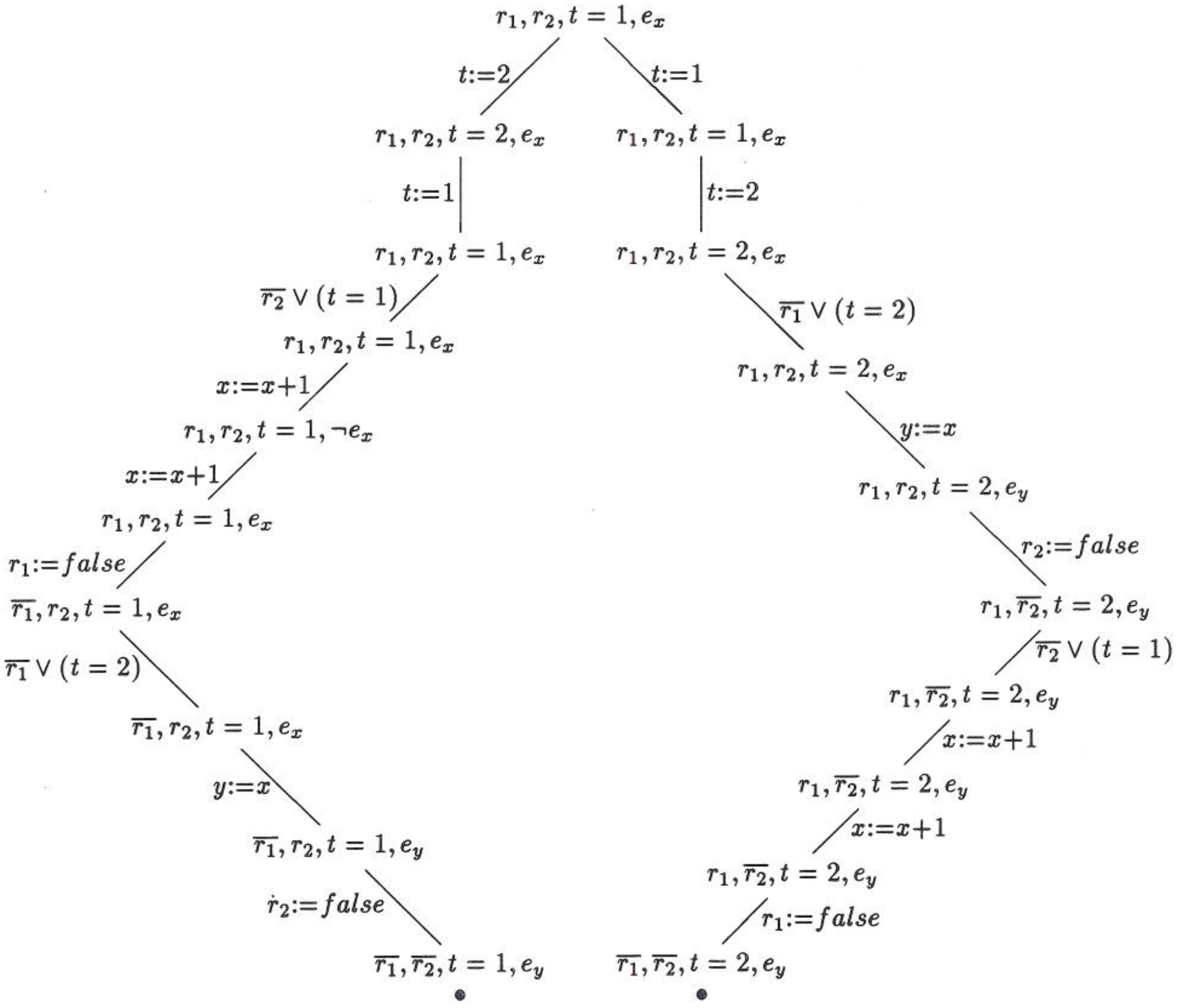

$r_1, r_2, t = 1, e_x$
$t{:=}2$
$t{:=}1$
$r_1, r_2, t = 2, e_x$
$r_1, r_2, t = 1, e_x$
$t{:=}1$
$t{:=}2$
$r_1, r_2, t = 1, e_x$
$r_1, r_2, t = 2, e_x$
$\overline{r_2} \vee (t = 1)$
$\overline{r_1} \vee (t = 2)$
$r_1, r_2, t = 1, e_x$
$x{:=}x{+}1$
$r_1, r_2, t = 2, e_x$
$r_1, r_2, t = 1, \neg e_x$
$y{:=}x$
$x{:=}x{+}1$
$r_1, r_2, t = 2, e_y$
$r_1, r_2, t = 1, e_x$
$r_1{:=}false$
$r_2{:=}false$
$\overline{r_1}, r_2, t = 1, e_x$
$r_1, \overline{r_2}, t = 2, e_y$
$\overline{r_1} \vee (t = 2)$
$\overline{r_2} \vee (t = 1)$
$r_1, \overline{r_2}, t = 2, e_y$
$\overline{r_1}, r_2, t = 1, e_x$
$x{:=}x{+}1$
$y{:=}x$
$r_1, \overline{r_2}, t = 2, e_y$
$x{:=}x{+}1$
$\overline{r_1}, r_2, t = 1, e_y$
$r_1, \overline{r_2}, t = 2, e_y$
$r_2{:=}false$
$r_1{:=}false$
$\overline{r_1}, \overline{r_2}, t = 1, e_y$
$\overline{r_1}, \overline{r_2}, t = 2, e_y$

Figure 8: An assertion satisfied by LEFT||RIGHT

4.3 The Set Partition Problem

The Set Partition Problem, essentially a parallelized version of one of the fundamental operations in the quicksort algorithm, was used by Barringer [Bar85] as a standard example illustrating the utility of various proof systems for parallel programs. Barringer's program is based on a solution of Dijkstra. Here we show briefly how this problem can be treated in our proof system.

Suppose we have two disjoint finite sets of integers represented by the variables S and T, whose union is initially the set A. Assume that we are provided with an atomic insert/delete operation $S{:}{-}x{+}y$ whose effect (provided x is in the set represented by S) is to remove x and insert y.

We want to transform S and T so that the union of the two sets is preserved, disjointness is maintained, the size of each set is preserved, and every member of S is smaller than every member of T. We want to do this by running two loops in parallel, one operating on each set, sharing variables which are used for ensuring correct cooperation. We want the parallel program to be partially correct with respect to the precondition PRE and post-condition $POST$ given by:

$$PRE: \quad S \cup T = A \ \& \ S \cap T = \{\} \ \& \ |S| = s \ \& \ |T| = t$$

$$POST: \quad S \cup T = A \ \& \ S \cap T = \{\} \ \& \ |S| = s \ \& \ |T| = t \ \& \ max(S) < min(T)$$

We also want the program to be free of deadlock[3].

The following program solves this problem:

```
$mx := max(S)$; $mn := min(T)$;
$rmx :=$ true; $rmn :=$ true;
$doneS :=$ false; $doneT :=$ false;
SMALL||LARGE
```

where

```
SMALL :: await $rmn$;
         while $mx > mn$ do
             $rmn :=$ false;
             $S{:}{-}mx{+}mn$;
             $doneS :=$ true;
             await $doneT$;
             $doneT :=$ false;
             $mx := max(S)$;
             $rmx :=$ true;
             await $rmn$
```

```
LARGE :: await $rmx$;
         while $mx > mn$ do
             $rmx :=$ false;
             $T{:}{-}mn{+}mx$;
             $doneT :=$ true;
             await $doneS$;
             $doneS :=$ false;
             $mn := min(T)$;
             $rmn :=$ true;
             await $rmx$
```

Figure 9 shows an assertion ϕ_S satisfied by the SMALL command; the corresponding assertion ϕ_T for the LARGE command is obtained in the obvious way. It is easy to derive ϕ_S from the skeleton assertion skel(SMALL), using the rules for implication[4].

[3] Barringer's parallel shared variable program for this problem assumes that insertion and deletion are separate atomic actions; it would be easy to adapt our solution to use these primitives instead, at the expense of slightly complicating the structure of the assertions used in our proofs. Another difference between our solution and Barringer's is that the shared variable program in [Bar85] is not deadlock-free.

[4] The rule for μ-induction is used crucially in this proof, in addition to the forward propagation rule.

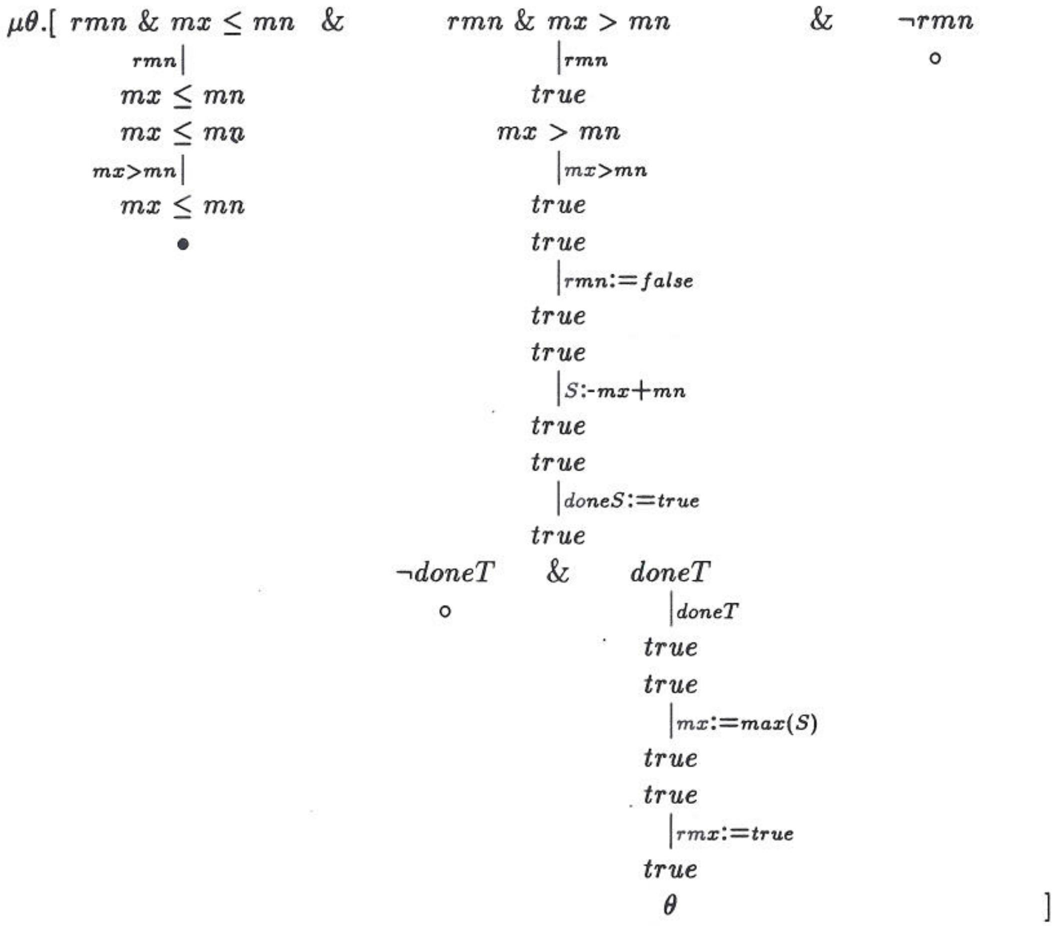


Figure 9: An assertion for SMALL

Figure 10 shows an assertion satisfied by SMALL||LARGE, using the following abbreviations:

$$\begin{aligned}
L &= mx < mn \\
D &= \neg doneS \;\&\; \neg doneT \\
D_S &= doneS \;\&\; \neg doneT \\
D_T &= \neg doneS \;\&\; doneT \\
D_{ST} &= doneS \;\&\; doneT \\
R &= \neg rmx \;\&\; \neg rmn \\
R_S &= rmx \;\&\; \neg rmn \\
R_T &= \neg rmx \;\&\; rmn \\
R_{ST} &= rmx \;\&\; rmn \\
M &= mx \in S \cap \overline{T} \;\&\; mn \in \overline{S} \cap T \\
M_S &= mx \in \overline{S} \cap \overline{T} \;\&\; mn \in S \cap T \\
M_T &= mx \in S \cap T \;\&\; mn \in \overline{S} \cap \overline{T} \\
M_{ST} &= mx \in \overline{S} \cap T \;\&\; mn \in S \cap \overline{T} \\
M_{mx} &= mx \in S \cap \overline{T} \\
M_{mn} &= mn \in \overline{S} \cap T \\
dS &= doneS \\
dT &= doneT
\end{aligned}$$

This assertion (Figure 10) may be derived from $\phi_S || \phi_T$ by the rules for implication in the following manner. The parallel composition of ϕ_S and ϕ_T implies a recursive assertion with the same shape as Figure 10, by retaining only the relevant conjuncts inside the assertion body. Then we conjoin the condition $D \& R_{ST} \& M \& L$ to the root of the first conjunct of the assertion body and use the rule for forward propagation to derive the conditions at the remaining nodes; similarly, we conjoin $R_{ST} \& M \& \overline{L}$ to the root of the second conjunct and propagate. Most of the propagation steps are obviously valid.

We write $mx \geq S$ as an abbreviation for $\forall x \in S. mx \geq x$, and similarly for $mn \leq T$. The following condition:

$$|S| = s \;\&\; |T| = t \;\&\; S \cap T \subseteq \{mx, mn\} \;\&\; S \cup T \cup \{mx, mn\} = A \;\&\; mx \geq S \;\&\; mn \leq T$$

is easily shown to be a global invariant for this program, since it is clearly preserved by every atomic action in the assertion of Figure 10, given the corresponding pre-condition. It may therefore be conjoined to every node of the assertion by forward propagation. Let us use the abbreviation INV for this condition.

The desired partial correctness property, and absence of deadlock, are then deducible, since we have obtained a safe assertion with root condition INV & D & R_{ST} & M, termination condition INV & M & $\overline{L}$, and deadlock condition **false**. It is clear that the initialization part of the overall program will terminate in a state satisfying INV & D & R_{ST} & M if begun in a state satisfying PRE; and POST is implied by the above termination condition.

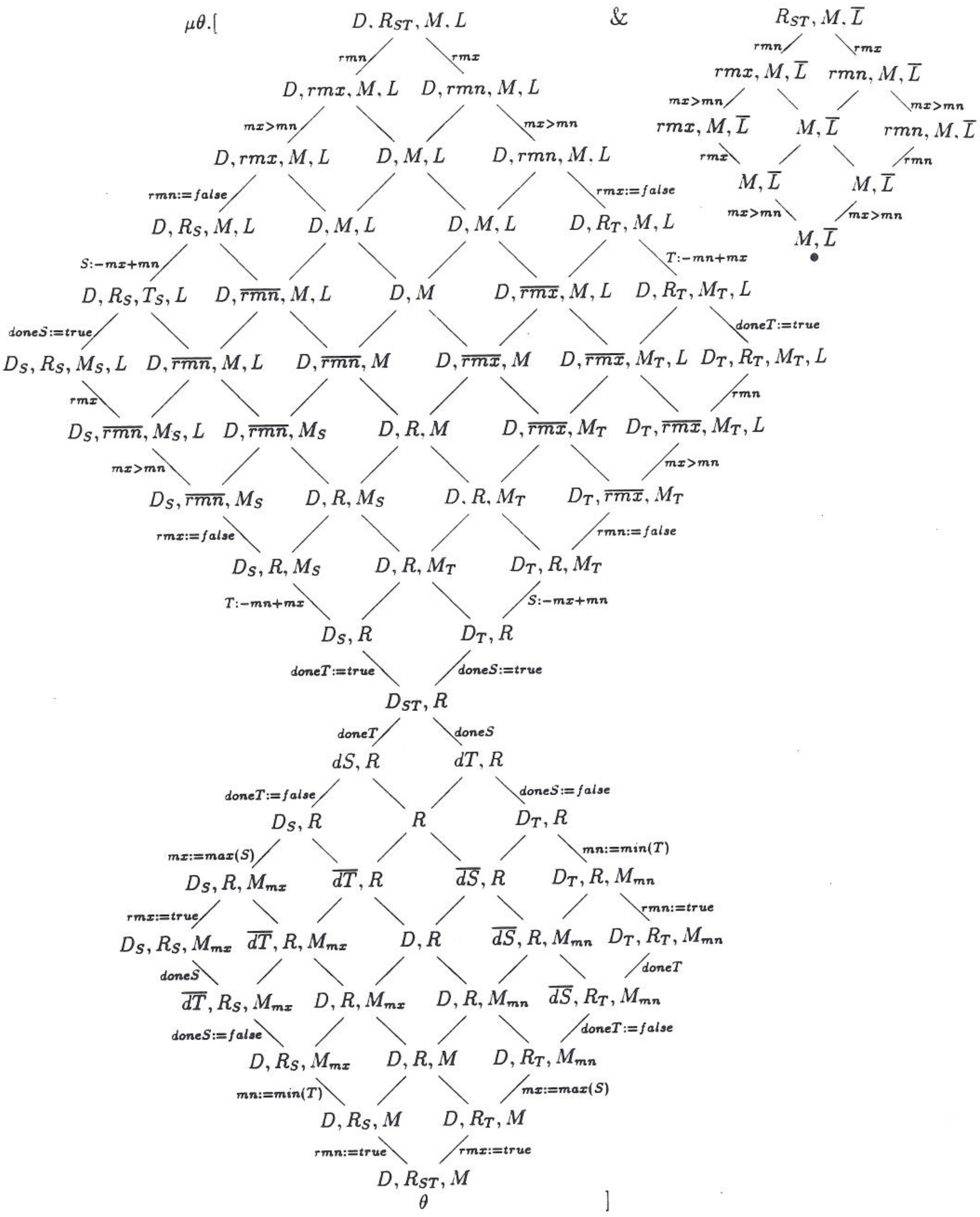


Figure 10: An assertion satisfied by SMALL||LARGE

5 Relationship to other proof systems

In this section we place our proof system in context by examining its relationship with some proof systems and proof methodologies proposed by other authors. We show that the Owicki-Gries proof rule can be seen as being based on a different form of syntactic parallel composition of assertions, and that the inference rule for parallel composition in the "generalized Hoare's logic" of Lamport and Schneider can be derived from our rule when the assertions used in their system are represented appropriately inside our assertion language. The same is true of the parallel proof rule in Gerth's Transition Logic.

5.1 The Owicki-Gries proof system

The Owicki-Gries proof system used conventional pca's of the form $\{P\}\,C\,\{Q\}$, although the parallel composition rule requires the use of *proof outlines* as premisses. A proof outline is a command text annotated with conditions, one before and one after each syntactic occurrence of an atomic action. At least for sequential commands, *safe* assertions in our assertion language correspond precisely with such proof outlines because computations of sequential commands follow the syntactic structure of the command. The analogy can be extended to parallel commands too, although the syntactic structure of a proof outline is no longer so close to that of the corresponding safe assertion.

Owicki's parallel composition rule essentially corresponds to a slightly different form of parallel composition of assertions. This may be defined as follows.

For $\phi = P\sum_{i=1}^{n}\alpha_i P_i\phi_i$ and $\psi = Q\sum_{j=1}^{m}\beta_j Q_j\psi_j$ we put

$$[\phi\|_O\psi] \;=\; \{P\&Q\}\,(\sum_{i=1}^{n}\alpha_i\,\{P_i\&Q\}\,[\phi_i\|_O\psi] + \sum_{j=1}^{m}\beta_j\,\{P\&Q_j\}\,\beta_j[\phi\|_O\psi_j]).$$

We also specify that

$$\begin{aligned}
[P\bullet\|_O Q\textstyle\sum_{j=1}^{m}\beta_j Q_j\psi_j] &= \{P\&Q\}\textstyle\sum_{j=1}^{m}\beta_j\,\{P\&Q_j\}\,[P\bullet\|_O\psi_j],\\
[P\circ\|_O Q\textstyle\sum_{j=1}^{m}\beta_j Q_j\psi_j] &= \{P\&Q\}\textstyle\sum_{j=1}^{m}\beta_j\,\{P\&Q_j\}\,[P\circ\|_O\psi_j],
\end{aligned}$$

together with the symmetric versions. We also put

$$\begin{aligned}
[P\bullet\|_O Q\bullet] &= \{P\&Q\}\bullet\\
[P\bullet\|_O Q\circ] &= \{P\&Q\}\circ\\
[P\circ\|_O Q\bullet] &= \{P\&Q\}\circ\\
[P\circ\|_O Q\circ] &= \{P\&Q\}\circ
\end{aligned}$$

as before. However, this form of composition does not always produce an assertion which correctly describes the behavior of a parallel composition of commands. We need the notion of *interference-freedom* to guarantee this [OG76].

Definition 5.1 The set atoms(ϕ) of *atomic sub-assertions* of ϕ is defined by induction on the depth of ϕ:

$$\begin{aligned}
&\text{atoms}(P\circ) = \text{atoms}(P\bullet) = \{\}\\
&\text{atoms}(P\textstyle\sum_{i=1}^{n}\alpha_i P_i\phi_i) = \{\{P\}\,\alpha_i\,\{P_i\}\,|\,1\le i\le n\}\cup\bigcup_{i=1}^{n}\text{atoms}(\phi_i)\\
&\text{atoms}(\phi_1\&\phi_2) = \text{atoms}(\phi_1)\cup\text{atoms}(\phi_2)\\
&\text{atoms}(\theta) = \{\}\\
&\text{atoms}(\mu\theta.\phi) = \text{atoms}(\phi).
\end{aligned}$$

●

Definition 5.2 Two assertions ϕ and ψ are interference-free, written int-free(ϕ, ψ), iff for every pair of atomic assertions

$$\{p\}\,\alpha\,\{p'\} \in \text{atoms}(\phi), \qquad \{q\}\,\beta\,\{q'\} \in \text{atoms}(\psi),$$

the pca's

$$\{p\&q\}\,\alpha\,\{q\}\,,\{p\&q\}\,\beta\,\{p\}\,,\{p\&q'\}\,\alpha\,\{q'\}\,,\{p'\&q\}\,\beta\,\{p'\}$$

are valid. •

Theorem 5.3 *If ϕ and ψ are interference-free then*

$$\models\; C_1 \;\textbf{sat}\; \phi \;\textit{and}\; \models\; C_2 \;\textbf{sat}\; \psi \quad \textit{imply} \quad \models\; [C_1 \| C_2] \;\textbf{sat}\; [\phi \|_O \psi].$$

In view of this result the following inference rule is valid:

$$\frac{C_1 \;\textbf{sat}\; \phi \quad C_2 \;\textbf{sat}\; \psi}{[C_1 \| C_2] \;\textbf{sat}\; [\phi \|_O \psi]} \quad \text{if int-free}(\phi, \psi)$$

Note that this theorem and the proof rule are stated in a form applicable to *all* assertions, not just to safe assertions. This can, therefore, be regarded as a slight extension of Owicki's ideas to encompass a more expressive assertion language. The following result shows that interference-freedom guarantees the preservation of safeness.

Theorem 5.4 *If ϕ and ψ are safe and interference-free, then $[\phi \|_O \psi]$ is safe.*

The root and termination conditions of this form of parallel composition satisfy the following logical equivalences:

$$\begin{aligned} \text{root}(\phi \|_O \psi) &\equiv \text{root}(\phi) \;\&\; \text{root}(\psi) \\ \text{term}(\phi \|_O \psi) &\equiv \text{term}(\phi) \;\&\; \text{term}(\psi). \end{aligned}$$

This may be shown by an inductive argument. There is then an obvious link with Owicki's proof rule for parallel composition. This rule, taken from [OG76], is:

$$\frac{\text{proofs of } \{P_1\}\,C_1\,\{Q_1\}\,,\; \{P_2\}\,C_2\,\{Q_2\} \;\text{ int-free}}{\{P_1\&P_2\}\,[C_1\|C_2]\,\{Q_1\&Q_2\}}$$

Now proof outlines for the Hoare assertions $\{P_i\}\,C_i\,\{Q_i\}$ correspond to safe assertions ϕ_i such that C_i **sat** ϕ_i, with root$(\phi_i) = P_i$ and term$(\phi_i) = Q_i$. The interference-freedom of these proof outlines corresponds to interference-freedom of ϕ_1 and ϕ_2. Then $[\phi_1 \|_O \phi_2]$ is a safe assertion satisfied by $[C_1\|C_2]$, and has root $P_1\&P_2$ and termination condition $Q_1\&Q_2$. Thus, a proof using Owicki's rule can be represented in our system using the above inference rule for $\phi\|_O\psi$.

Auxiliary variables and auxiliary critical regions

It is well known [OG76] that a proof system using the Owicki-Gries proof rule for parallel composition is not complete for partial correctness assertions. As a simple example, it is impossible even to prove the obviously valid assertion

$$\{x = 0\}\,[x{:=}x + 1\|x{:=}x + 1]\,\{x = 2\}$$

using the rules for parallel composition and assignment and the Rule of Consequence alone. We chose to avoid this problem by introducing conjunctions and implication. Owicki achieved completeness by adding "auxiliary variables" to programs and adding new proof rules to allow their use.

A set X of identifiers is *auxiliary* for a command C if all free occurrences of identifiers from this set in C are inside assignments to identifiers also in X. Thus, for instance, for the command

$$x{:=}x+1; y{:=}z; a{:=}x$$

the sets $\{y\}, \{y,z\}, \{a,x\}$ and $\{x,y,z,a\}$ are auxiliary, but $\{x\}$ is not. Let us write

$$C \text{ aux } X$$

when X is an auxiliary set of identifiers for C.

Given any set X of identifiers and any command C, let $C \setminus X$ be the command resulting from the deletion in C of all assignments to identifiers in X. It is clear that if X is auxiliary for C then $C \setminus X$ has the same partial correctness effect on identifiers outside X as C does, and $C \setminus X$ leaves the values of all identifiers in X fixed.

Let free$[\![P, Q]\!]$stand for the set of identifiers having a free occurrence in either P or Q. Owicki's auxiliary variables rule is:

$$\frac{\{P\}\, C\, \{Q\}}{\{P\}\, C \setminus X\, \{Q\}} \quad \text{if } C \text{ aux } X \text{ \& free}[\![P, Q]\!] \cap X = \{\}.$$

In addition to this rule, for completeness of the Owicki proof system one also needs a rule for eliminating "unnecessary" critical regions and irrelevant atomic actions which have been inserted merely to cope with auxiliary variables.

As an example, we can now prove (as in [OG76]) the assertion

$$\{x=0\}\, [x{:=}x+1 \| x{:=}x+1]\, \{x=2\}$$

by first introducing auxiliary variables a and b to tag the two assignments and establishing the assertion

$$\{x=0\}\, a{:=}0;\ b{:=}0;\ [\langle a{:=}1;\ x{:=}x+1\rangle \| \langle b{:=}1;\ x{:=}x+1\rangle]\, \{x=2\}\,.$$

Then we eliminate the auxiliary variables and the extra critical regions. This augmented assertion can be proved by first proving the following assertions for the two parallel components:

$$\{P_a\}\, \langle a{:=}1;\ x{:=}x+1\rangle\, \{Q_a\}\,,$$
$$\{P_b\}\, \langle b{:=}1;\ x{:=}x+1\rangle\, \{Q_b\}\,,$$

where

$$\begin{aligned} P_a &= (b=0 \,\&\, x=0) \vee (b=1 \,\&\, x=1), \\ Q_a &= (b=0 \,\&\, x=1 \,\&\, a=1) \vee (b=1 \,\&\, x=2 \,\&\, a=1), \\ P_b &= (a=0 \,\&\, x=0) \vee (a=1 \,\&\, x=1), \\ Q_b &= (a=0 \,\&\, x=1 \,\&\, b=1) \vee (a=1 \,\&\, x=2 \,\&\, b=1). \end{aligned}$$

These two proof outlines are interference-free (this requires the verification of four conditions), and their use in the parallel rule enables us to conclude

$$\{P_a \,\&\, P_b\}\, [\langle a{:=}1;\ x{:=}x+1\rangle \| \langle b{:=}1;\ x{:=}x=1\rangle]\, \{Q_a \,\&\, Q_b\}\,.$$

Since we have

$$\begin{aligned}&\{x=0\}\,a{:=}0;\ b{:=}0\,\{x=0\ \&\ a=0\ \&\ b=0\}\,,\\&x=0\ \&\ a=0\ \&\ b=0\ \Rightarrow\ P_a\ \&\ Q_a,\\&Q_a\ \&\ Q_b\ \Rightarrow\ x=2,\end{aligned}$$

the desired result follows by the usual Hoare rules for sequential composition and the Rule of Consequence. In contrast, the desired pcda is deducible in our proof system by means of conjunction and the parallel composition rule, without needing to modify the program text.

Owicki and Gries suggested a method of proving deadlock-freedom for a parallel program by means of a global invariant P such that P is false in all deadlock states. The verification that P is indeed false in deadlock states, and the calculation of the set of possible deadlock states (or a syntactically determinable superset of the set of potential deadlock states), is performed in a separate phase of the analysis. In a sense, then, this kind of methodology requires stepping outside of the proof system and performing a "meta-analysis". In contrast, our assertion language can express deadlock properties directly and one can carry out reasoning about both partial correctness and deadlock-freedom entirely within the proof system.

5.2 Lamport's proof rules

Lamport [Lam80] proposed using assertions of the form $\{P\}\,C\,\{Q\}$ with the interpretation that in every execution which starts somewhere inside C with P true, P remains true until C terminates, when Q will be true; in particular, it follows that P implies Q. Such an assertion corresponds to one of our transition assertions $P\sum_{i=1}^{n}\alpha_i P_i\phi_i$ in which each P_i (and all other intermediate conditions) are identical to P and all leaf conditions are identical to Q. The proof rule for parallel composition given in [Lam80] was:

$$\frac{\{P\}\,C_1\,\{Q\}\quad \{P\}\,C_2\,\{Q\}}{\{P\}\,[C_1\|C_2]\,\{Q\}}$$

But our definition of parallel composition of assertions preserves this uniformity property: the parallel composition of (the assertions representing) $\{P\}\,C_1\,\{Q\}$ and $\{P\}\,C_2\,\{Q\}$ will again have leaf Q and each intermediate condition will be P. (In fact, this uniformity property is preserved both by our $\|$ and by the other form $\|_O$). Thus, Lamport's rule is derivable from our rule for parallel composition.

Instead of adding auxiliary variables, Lamport suggested the use of program labels such as λ_i for the control points of a program, and including in the condition language expressions of the form at(λ), inside(λ), after(λ). Lamport's system requires reasoning about control points and the flow of control between them. Since in a Lamport-style assertion the same P has to represent more than one control point at a time, the conditions can get rather large: typically, P takes the form of a conjunction like $\bigwedge_{\lambda\in\Lambda}(\mathrm{at}(\lambda)\Rightarrow P_\lambda)$. Indeed, it could be argued that since the same P is serving a multitude of purposes it is natural to split it up into its components and to attach these components to the control points at which they are intended to hold; this is more in line with our notation, with control points corresponding to nodes in a tree.

The Generalized Hoare Logic of Lamport and Schneider [LS84] used a similar type of assertion to those of [Lam80], except that they insisted that the post-condition coincide with the pre-condition: they used invariant assertions $\{P\}\,C\,\{P\}$. The interpretation is as before, that whenever an execution begins somewhere inside C with P true, P will remain true until termination. Again, their proof rule for parallel composition (essentially, a special case of the one from [Lam80], given above) is representable in our system. Again, control conditions are used inside invariants, so that an invariant is really serving a multitude of purposes and could profitably be split up and distributed to the separate control points.

5.3 Gerth's Transition Logic

The *Transition Logic* of Gerth [Ger83] also has some connection with our work. Gerth's assertions, written $[P]C[Q]$, are interpreted: every transition that begins somewhere in C from a state satisfying P ends in a state satisfying Q. Again, the conditions need to involve control point expressions. Gerth's rule for parallel composition is:

$$\frac{[P]C_1[Q] \quad [P]C_2[Q]}{[P][C_1\|C_2][Q].}$$

But the assertion $[P]C[Q]$ can again be rendered in our assertion language as an assertion with a simple structure (alternating P and Q along each branch), and again our parallel composition of assertions has the required effect, producing an assertion representating $[P][C_1\|C_2][Q]$ from representations of $[P]C_1[Q]$ and $[P]C_2[Q]$. This means that Gerth's rule can be derived in our system.

5.4 Other related work

The proof methodology and program development method advocated by Jones [Jon83] uses *rely* and *guarantee* conditions in addition to pre- and post-conditions. Roughly speaking, a rely condition corresponds to a pre-condition assumed by every atomic action in an assertion, and a guarantee condition is implied by all post-conditions of the relevant atomic actions.

Stirling describes a compositional reformulation of the Owicki-Gries logic involving Hoare quintuples $(\Gamma, \Delta) \vdash \{P\}\, C\, \{Q\}$, where Γ and Δ are sets of first-order conditions. He employs a proof rule for parallel composition which does not need an interference-freedom side-condition; instead, the rule may only be applied when a simple "interlocking" constraint is satisfied. This system still requires an auxiliary variables rule.

Other authors, for example [BKP84, MP82], have proposed compositional proof systems for concurrent programs in which the underlying assertions are temporal in nature. In contrast to these methods, we have avoided temporal assertions at the expense of using conjunction and implication as operations on more highly structured assertions built from conventional pre- and post-conditions. We still obtained a compositional proof system. In fact, our assertions do have some similarity with temporal logic in the sense that an assertion has built into it a specification of the possible atomic actions and the behavior of the command after each of them, so that one could indeed represent one of our assertions ϕ in a more conventional temporal or dynamic logic.

6 Conclusions

We have demonstrated a new proof methodology for shared variable parallel programs which, unlike earlier proof systems of [LG81, OG76], does not require the manipulation of auxiliary variables purely for proof-theoretical purposes and which does not require a notion of interference-freedom to guarantee soundness. However, we do not claim that we have found a panacea. Just as it is necessary to exercise skill in the choice and use of auxiliary variables in Owicki's system, our system requires a judicious choice of conjunctions. Moreover, our proof system exhibits the problems inherent in our underlying programming language. In particular, since parallel commands may share variables and their interaction can be extremely complex, it may be prohibitively difficult to keep track of the various threads of control when trying to reason about programs. This is true of informal attempts as well as of formal methods using any of the proposed proof systems, ours included. Specifically, as is well known, the state space of a parallel program may grow exponentially with the number of parallel components; this manifests itself in our methodology with a possibly exponential growth in assertion size. Similar drawbacks plague other approaches, although sometimes manifested in different ways. For example, in the Owicki-Gries methodology the number of side-conditions that need to be verified to establish interference-freedom may be large, and in Lamport's generalized Hoare Logic the global invariant may itself be large since it must typically contain a conjunct for each combination of control points. It would be worthwhile to develop machine-assisted tools to help deal with these problems.

In favor of our approach we note that a careful choice of transition assertion proof may reduce the number of interactions between program fragments that need to be considered, because we need only take into account potential interference between program phrases which actually may be executed in parallel dynamically, since the assertion structure is detailed enough to permit these cases to be determined. Thus, for example, in the Set Partition program we never needed to analyze the effect of actions in the second part of SMALL's loop body on pre- or post-conditions used in the first part of LARGE's loop body. In contrast, a proof using interference-freedom matches up for such an analysis all pairs of atomic actions occurring in the text of the two processes.

Our assertion language was chosen to permit reasoning about deadlock and partial correctness together, and by basing it on a resumption-style semantics we end up with tree-structured assertions. This means that the assertion language is rather expressive, and it could perhaps be argued that it is too expressive: it allows us make certain logical distinctions between program phrases which are actually irrelevant to the generalized partial correctness and deadlock properties of program built from those phrases. For instance, the commands **skip** and **skip**; **skip** satisfy exactly the same generalized pca's but not the same transition assertions. This issue is related to the full abstraction problem for our programming language [HP79]. It would be interesting to investigate the possibility of basing a proof methodology on a streamlined version of our assertion language, in which irrelevant **skip** actions are absorbed. Moreover, although we made the syntax of atomic actions explicit in the assertion structure, this was largely done to improve the readability of assertions and is not really necessary for axiomatic purposes.

Similar ideas to those used in this paper may be adopted in an axiomatic treatment of other forms of parallel programming. In particular, a communication-based distributed programming language such as CSP [Hoa78] may be axiomatized if we modify the class of assertions to represent the potential for communication and if we design a suitable parallel composition of assertions. When adapting these ideas to a language like CSP it becomes vital to record at least a vestige of the syntax of actions in assertion structure, so that we may ensure that parallel composition of assertions models synchronization appropriately. An outline of such an axiomatization is presented in [Bro86], and we plan to develop this further and to place this work in the context of existing proof systems for CSP based on the Owicki-Gries style [AFdR80, LG81].

Acknowledgements

The author is grateful for discussions with Eike Best, Ed Clarke, Rob Gerth, Michal Grabowski, Cliff Jones, Jay Misra, Francesmary Modugno, Jeannette Wing, Glynn Winskel, and Guo Qiang Zhang.

References

[AFdR80] K. R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, 1980.

[AO91] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991.

[Apt81] K. R. Apt. Ten years of Hoare's logic: A survey. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.

[Ash75] E. A. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10:110–135, 1975.

[BA82] M. Ben-Ari. *Principles of Concurrent Programming.* Prentice/Hall International, 1982.

[Bar85] H. Barringer. *A Survey of Verification Techniques for Parallel Programs*, volume 191 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[Bes82] E. Best. A relational framework for concurrent programs using atomic actions. In *Proc. IFIP TC2 Conference*, 1982.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic assertions. In *Sixteenth Annual ACM Symposium on Theory of Computing*, May 1984.

[Bro85] S. D. Brookes. A fully abstract semantics and proof system for an ALGOL-like language with sharing. In A. Melton, editor, *Proceedings of 1st International Conference on Mathematical Foundations of Programming Semantics*, volume 239 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[Bro86] S. D. Brookes. A semantically based proof system for partial correctness and deadlock in CSP. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 58–65. IEEE Computer Society Press, June 1986.

[Coo78] S. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, February 1978.

[Dij68] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[Ger83] R. Gerth. Transition logic. In *Proceedings of the 16th ACM STOC Conference*, 1983.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, October 1969.

[Hoa72] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*, pages 61–71. Academic Press, 1972.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, August 1978.

[HP79] M. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming language. In *Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 108–120. Springer Verlag, 1979.

[Jon83] C. B. Jones. Tentative steps towards a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, October 1983.

[Kel76] R. M. Keller. Formal verification of parallel programs. *CACM*, 19(7):371–384, July 1976.

[Lam80] L. Lamport. The 'Hoare Logic' of concurrent programs. *Acta Informatica*, 14:21–37, 1980.

[LG81] G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[LS84] L. Lamport and F. Schneider. The 'Hoare Logic' of CSP, and all that. *ACM TOPLAS*, 6(2):281–296, April 1984.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.

[MP82] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, 1982.

[OD82] M. O' Donnell. A critique of the foundations of Hoare-style programming logic. *CACM*, 25(12):927–934, December 1982.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[Par70] D. M. R. Park. Fixpoint induction and proofs of program properties. In B. Meltzer and D. Michie, editors, *Machine Intelligence, Vol. 5*, pages 59–78. Edinburgh University Press, 1970.

[Pet81] G. L Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):223–252, 1981.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. DAIMI Report FN-19, Aarhus University, 1981.

[Sco76] D. S. Scott. Data types as lattices. *SIAM J. on Computing*, 5:522–587, 1976.

Appendix: Remarks on Relative Completeness

We have shown that the proof system introduced in this paper is sound, with justification provided by reference to the underlying operational semantics. In other words, whenever C **sat** ϕ is provable it follows that ϕ correctly describes (some aspect of) the operational behavior of C. The proof system is not complete, for a rather trivial reason: every command satisfies an assertion ϕ whose root is **false**, but this is not deducible from the above rules. One solution is to add a logical rule to this effect:

$$\frac{\neg\mathrm{root}(\phi)}{C\ \mathbf{sat}\ \phi}$$

Such a rule is obviously sound, but plays no useful role in program verification.

We are primarily interested in this paper in the derivation of valid pcda's. As is the case for Hoare's original logic, the best we can achieve is *relative completeness*, or completeness in the sense of Cook. Let **Th** be the set of valid conditions. We write

$$\mathbf{Th}\ \vdash\ C\ \mathbf{sat}\ \phi$$

if C **sat** ϕ can be proved using assumptions from **Th**. We assume that the condition language is *expressive*, in that for every command C and every condition Q there is a condition P expressing the weakest pre-condition for C and Q, or equivalently, for every command and every condition the strongest post-condition is expressible in the condition language.

Theorem 6.1 *For every valid pcda $\{P\}\,C\,\{Q\}\,[R]$ there is a safe assertion ϕ for which* **Th** $\vdash$ C **sat** ϕ *and $P \Rightarrow root(\phi)$, $term(\phi) \Rightarrow Q$, and $dead(\phi) \Rightarrow R$.*

The key idea in the proof of this result is to generalize the notion of strongest post-condition to the parallel setting. We have already introduce safe assertions, program skeletons, and the forward propagation rule. It is easy to show that for every command C the assertion skel(C) is provable from our axioms and rules. One can then show that every valid pcda $\{P\}\,C\,\{Q\}\,[R]$ may be derived, using rules for implication (including forward propagation) from skel(C). One way to do this is to systematically transform the assertion skel(C) from the root downwards to propagate P as follows. Since this is essentially forward propagation of strongest post-conditions, we refer to the resulting assertion as the *strongest post-assertion* of C and P.

Definition 6.2 Let spc(P, α) denote the strongest post-condition for atomic action α and pre-condition P. The strongest post-assertion of ϕ and P is defined by induction on the syntactic structure of ϕ:

$$\begin{aligned}
\mathrm{spa}(P, Q\circ) &= \{P\ \&\ Q\}\circ \\
\mathrm{spa}(P, Q\bullet) &= \{P\ \&\ Q\}\bullet \\
\mathrm{spa}(P, Q \textstyle\sum_{i=1}^{n} \alpha_i Q_i \phi_i) &= \{P\ \&\ Q\} \textstyle\sum_{i=1}^{n} \alpha_i \{R_i\}\mathrm{spa}(R_i, \phi_i) \\
\mathrm{spa}(P, \phi_1\ \&\ \phi_2) &= \mathrm{spa}(P, \phi_1)\ \&\ \mathrm{spa}(P, \phi_2) \\
\mathrm{spa}(\mu\theta.\phi) &= \mu\theta.\mathrm{spa}(P, \phi) \\
\mathrm{spa}(P, \theta) &= \theta,
\end{aligned}$$

where each R_i is of form spc$(P\ \&\ Q, \alpha_i)$. •

Intuitively, the assertion spa$(P, \mathrm{skel}(C))$ is the strongest safe assertion with root condition P that C satisfies. It is clear that since C **sat** skel(C) is provable, so is C **sat** spa$(P, \mathrm{skel}(C))$. If $\{P\}C\{Q\}[R]$ is valid it will then be deducible from this strongest post-assertion, since the termination condition of spa$(P, \mathrm{skel}(C))$ will imply Q and the deadlock condition of spa$(P, \mathrm{skel}(C))$ will imply R.