LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Proving total correctness of concurrent programs without using auxiliary variables

by

Pawel Pączkowski

Proving total correctness of concurrent programs.........

LFCS Report Series

ECS-LFCS-89-100
(also published as CSR-318-89)

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

November 1989

Copyright © 1989, LFCS

Copyright © 1989, Laboratory for Foundations of Computer Science,
University of Edinburgh. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

Proving total correctness of concurrent programs without using auxiliary variables

Paweł Pączkowski

November 1989

Abstract

Auxiliary variables or location predicates are commonly used in assertional proof systems for concurrency in order to achieve completeness. When encoded in assertions the references to control flow become entangled in state predicates and can obscure the clarity of argument. We present assertional proof techniques for showing partial correctness, deadlock freedom and termination in which program locations do not need to be encoded in assertions. Instead, the separated reasoning on the flow of control becomes a mechanizable step of the proof. The proposed proof techniques are shown to be sound and complete. Our methodology is applicable to concurrent programming languages based on shared variables or message passing.

1 Introduction

This paper deals with assertional methodology of proving properties of concurrent programs. Although not as powerful as temporal logics, assertional reasoning originated in [Floyd 67] is very natural, conceptually simple and capable of handling a number of important properties like partial and total correctness.

In pursuit of compositionality the known assertional proof systems for concurrency either introduce history variables or allow reasoning about program locations. History variables [Misra Chandy 81, Soundararajan 83, ZRE 85] emerge naturally in many message-passing programs but there are examples, like set partitioning [Dijkstra 82], where introducing the whole history of computation into reasoning would seem excessive. In such cases one usually resorts to reasoning on program locations, what can be done by means of location predicates [Lamport 80], program counters [Flon Suzuki 81] or auxiliary variables [Owicki Gries 76]. In any case one faces either extension of the assertion language or modification of the text of programs one wants to reason about.

In this paper we adopt a different and simpler strategy. The assertions expressing properties of computation states will be attached to configurations of a suitably chosen transition

system called an annotation. As a part of the verification process this transition system will be shown to correspond to the control flow of the verified program. Thus, reasoning on control flow will be factored out as a separate step of the correctness proof what will eliminate the need for auxiliary variables or program locations in assertions. As an additional benefit this step of the correctness proof can be mechanized.

Adopting such an approach we develop proof techniques for partial correctness, deadlock freedom and total correctness where no information on program locations needs to be encoded in assertions. We show soundness of the proposed proof techniques. Completeness in the sense of Cook is proved for partial correctness and deadlock freedom, the completeness result for total correctness is obtained after the usual restriction to arithmetical interpretations.

The resulting proof methodology can be viewed as a generalisation of Floyd's method of program verification. This means that our approach is not compositional in this sense that proving a property of parallel composition $S_1 \parallel S_2$ is not decomposed into proofs of properties of S_1 and S_2. However, hierarchical development of proofs can be recovered by introducing action refinement. We do not address this issue in this paper.

Annotations on which we base correctness proofs are branching structures of predicates of first order logic. A similar concept was exploited by Brookes to build proof systems for partial correctness and deadlock freedom for concurrency [Brookes 85, Brookes 86]. Brookes' concern, however, is compositionality resulting in more complex annotations than presented here; in particular recursively defined trees of predicates are required for handling loops. He also does not tackle total correctness.

The proposed proof methodology will be applied to two example programming languages, one with shared variables, the other equipped with message passing primitives. This extends the results of [Flon Suzuki 81] where axioms for total correctness for a concurrent programming language with shared variables are given. There, however, the translation of a verified program into Dijkstra's language of guarded statements results in an extensive use of program counters in assertions and there is no relation between the correctness proof and the structure of the verified program.

The paper is organised as follows. We start by introducing basic notions and the example programming languages, sections 2–7. Then we define annotations, the central notion to our proof methodology. Section 9 contains a brief discussion of the expressiveness issues which are important for the completeness proofs. Then follow three main sections presenting proof methods for showing partial correctness, deadlock freedom and termination. In each case soundness and completeness are proved. We end with examples, applying our proof methodology to the set partitioning program and to a simple mutual exclusion protocol.

2 Transition systems

A labelled transition system (lts in short) is a triple $(C, Act, \longrightarrow)$, where C is a set of configurations, Act is a set of actions and $\longrightarrow \subset C \times Act \times C$ is a transition relation. We write $c \xrightarrow{\alpha} c'$ if $(c, \alpha, c') \in \longrightarrow$.

When the set of configurations is defined as a term algebra the transition relation can be defined in a so called structural way [Plotkin 81] by providing a little inference system for deriving transitions understood as "formulas" $c \xrightarrow{\alpha} c'$, for $c, c' \in C, \alpha \in Act$.

We will be also considering *extended* lts's $(C, Act, \longrightarrow, I, F, \ldots)$, where $(C, Act, \longrightarrow)$ is a lts and $I, F, \ldots$ are a finite number of distinguished subsets of C. As a typical such extension lts's will be equipped with sets of initial and final configurations (denoted by I and F, respectively). If the distinguished set has only one element, say $I = \{i\}$, instead of $\{i\}$ we will write i, possibly with some indices, in the definition of the extended lts.

Once we have defined an lts we can convert it into an extended lts by defining the distinguished subsets of configurations. Another typical operation on transition systems is restricting an extended lts with initial configurations (and possibly other extensions) to the reachable part, i.e. to those configurations and transitions which can be reached from some configuration in the initial set.

Definition A *path* in a lts is a finite or infinite sequence of its transitions $c_i \xrightarrow{\alpha_i} c_i'$, $i = 0, 1, \ldots$ such that $c_i' = c_{i+1}$.

We introduce the following important notion

Definition Let $A_i = (C_i, Act_i, \longrightarrow_i, I_i, F_i, \ldots)$, for $i = 1, 2$. We say that ρ is a *simulation* from A_1 to A_2, denoted $\rho : A_1 \rightarrow A_2$, if ρ is a function from configurations of A_1 into configurations of A_2 such that

1. ρ preserves the distinguished sets, i.e. if $c \in I_1$ $(F_1, \ldots)$ then $\rho(c) \in I_2$ $(F_2, \ldots)$
2. if $c \xrightarrow{\alpha}_1 c'$ then $\rho(c) \xrightarrow{\alpha}_2 \rho(c')$

We also say that A_2 simulates A_1 if a simulation $\rho : A_1 \rightarrow A_2$ exists.

If there is a bijective simulation from A_1 into A_2 whose converse is also a simulation we say that A_1 and A_2 are isomorphic.

3 The assertion language and its interpretation

In the following we assume some first order assertion language $\mathcal{P}$ with equality. The terms and quantifier free formulas of our assertion language will be used as, respectively, assignable expressions and boolean expressions of programming languages we will be dealing with.

We also assume some fixed interpretation of the assertion language and whenever we

talk about semantical validity or satisfaction of formulas we mean it relative to our fixed interpretation.

Valuations of variables, i.e. functions from the set of variables of $\mathcal{P}$ to the set of values of the fixed interpretation, will be called states. St will denote the set of all states and σ will range over it. For a predicate p belonging to $\mathcal{P}$ $[\![p]\!]$ will denote the set $\{\sigma \in St \mid \sigma \models p\}$. Σ will range over subsets of St. We write $\Sigma \models p$ when $\forall \sigma \in \Sigma \;\; \sigma \models p$.

4 Programming languages

The proposed program verification method will be applied to two example concurrent languages. Define $\mathcal{S}_w$, a parallel while language with the await statement whose statements have the following abstract syntax

$$S \;=\; a \mid \textbf{await } b \textbf{ then } a \mid S;S \mid S \parallel S \mid \textbf{if } b \textbf{ then } S \textbf{ else } S \mid \textbf{while } b \textbf{ do } S$$

where a belongs to a class of atomic statements which includes assignments and b ranges over boolean expressions. We allow only atomic actions in the await statement because the more general construct allowing arbitrary statements in awaits would involve two separate primitives: synchronisation and converting a statement into an interruptible atomic action. We believe those concepts should be dealt with separately and, for simplicity, we did not include the latter one. We are however free to choose the atomic actions to correspond semantically to composite statements and recover in such a way the full expressive power of the await statement.

The second language, $\mathcal{S}_c$ is a CSP-like programming language with message passing through channels. Below follows the abstract syntax of its statements S and guarded statements G, where a and b are as above and c ranges over communication actions which, as usually in CSP, can be of two kinds: $ch!t$ or $ch?x$, where ch is a channel name, t a term and x a variable.

$$G \;=\; b \rightarrow S \mid b;c \rightarrow S \mid G \,[\!]\, G$$

$$S \;=\; a \mid c \mid S;S \mid S \parallel S \mid \textbf{do } G \textbf{ od} \mid \textbf{if } G \textbf{ fi}$$

We use either parentheses or more traditional symbols **begin**, **end** when we want to avoid syntactical ambiguities.

We introduce the notion of *atomic action* for each of the above languages. For $\mathcal{S}_w$ the set of atomic actions comprises the atomic statements, await statements and boolean expressions. For $\mathcal{S}_c$ the set of atomic actions consists of atomic statements, boolean expressions and constructs $b_1;c_1 \parallel b_2;c_2$ representing a performed communication, where one of c_i is an input action and the other is an output action on the same channel (such communication actions are called *matching*).

Various notions and objects appearing later on will have subscripts w or c to point out the particular language we mean at the moment. If the subscripts are omitted, we mean both cases simultaneously.

The structural operational semantics for the above programming languages can be defined in a standard way. So defined operational semantics is a lts whose configurations are pairs $\langle \sigma, S \rangle$, where σ is a state and S is a statement which "remains to be executed". The statement part of the configuration indicates the position where control flow resides at a given moment of computation. The transition rules are defined in a structural way and describe both the state and control flow changes.

We will depart slightly from this standard procedure by defining first transition systems which describe only possible flows of control in programs. The actual operational semantics will be obtained by interpreting semantically transitions of control flow. We proceed in such a way because anyhow we will need the definition of control flow later on.

5 Control flow

We define labelled transition systems called CF_w and CF_c representing possible control flows of programs of $\mathcal{S}_w$ and $\mathcal{S}_c$.

The statements of $\mathcal{S}_w$ together with an additional symbol ε representing the end of computation constitute the set of configurations of CF_w. The transitions are labelled with atomic actions of $\mathcal{S}_w$. A transition $S \xrightarrow{\alpha} T$ is to be understood as a possibility of statement S to perform an action α after which statement T will remain to be executed. Table 1 presents the axioms and rules for deriving transitions of CF_w. As is usually done, $\dfrac{S \xrightarrow{\alpha} T_1 \mid T_2}{S' \xrightarrow{\beta} T_1' \mid T_2'}$ abbreviates the two rules $\dfrac{S \xrightarrow{\alpha} T_1}{S' \xrightarrow{\beta} T_1'}, \dfrac{S \xrightarrow{\alpha} T_2}{S' \xrightarrow{\beta} T_2'}$.

The set of configurations of CF_c consists of the statements of $\mathcal{S}_c$ plus the additional symbol ε for the finished computation. The transitions of CF_c are defined in two steps. First an auxiliary transition relation $\longrightarrow_p$ is defined, then $\longrightarrow_p$ is restricted to give $\longrightarrow$, the actual transition relation of CF_c. Transitions $\longrightarrow$ are labelled with atomic actions of $\mathcal{S}_c$, transitions $\longrightarrow_p$ can be additionally labelled with "unmatched" communication guards $b; c$. The rules for deriving transitions of CF_c are precented in Table 2. Note, that guarded statements G and the additional symbol *fail* are not configurations of CF_c although they can appear in derivations of transitions.

$$a \xrightarrow{a} \varepsilon$$

$$\textbf{await } b \textbf{ then } a \xrightarrow{\textbf{await } b \textbf{ then } a} \varepsilon$$

$$\frac{S \xrightarrow{\alpha} S' \mid \varepsilon}{S;T \xrightarrow{\alpha} S';T \mid T}$$

$$\textbf{if } b \textbf{ then } S \textbf{ else } T \xrightarrow{b} S$$

$$\textbf{if } b \textbf{ then } S \textbf{ else } T \xrightarrow{\neg b} T$$

$$\textbf{while } b \textbf{ do } S \xrightarrow{\neg b} \varepsilon$$

$$\textbf{while } b \textbf{ do } S \xrightarrow{b} S; \textbf{while } b \textbf{ do } S$$

$$\frac{S \xrightarrow{\alpha} S' \mid \varepsilon}{S \parallel T \xrightarrow{\alpha} S' \parallel T \mid T} \qquad \frac{T \xrightarrow{\alpha} T' \mid \varepsilon}{S \parallel T \xrightarrow{\alpha} S \parallel T' \mid S}$$

Table 1. Control flow derivation rules. $S, T, S', T' \in \mathcal{S}_w$.

6 Operational semantics

By operational semantics we mean an lts whose configurations are pairs $\langle \sigma , S \rangle$, where σ is a state and S a configuration of control flow. Transitions of operational semantics are obtained by interpreting semantically the transitions of Control Flow:

$$\frac{S \xrightarrow{\alpha} S' \quad (\sigma, \sigma') \in [\![\alpha]\!]}{\langle \sigma , S \rangle \xrightarrow{\alpha} \langle \sigma' , S' \rangle}$$

where $[\![\alpha]\!] \subset St \times St$ is a relational semantics of atomic action α, defined as follows for the specific cases of assignment, boolean test, communication action and the await statement:

$$\begin{array}{lll}
(\sigma, \sigma') \in [\![x := t]\!] & \text{iff} & \sigma' = \sigma[\sigma(t)/x] \\
(\sigma, \sigma') \in [\![b]\!] & \text{iff} & \sigma \models b \text{ and } \sigma' = \sigma \\
(\sigma, \sigma') \in [\![b_1; c_1 \parallel b_2; c_2]\!] & \text{iff} & \sigma \models b_1 \wedge b_2 \text{ and } \sigma' = \sigma[\sigma(t)/x] \\
 & & \text{where one of } c_i \text{ is } ch!t \text{ and the other } ch?x \\
(\sigma, \sigma') \in [\![\textbf{await } b \textbf{ then } a]\!] & \text{iff} & \sigma \models b \text{ and } (\sigma, \sigma') \in [\![a]\!]
\end{array}$$

For each atomic action a other than assignment we assume some predefined *total* relation $[\![a]\!]$.

$$(b \rightarrow S) \xrightarrow{b}_p S \qquad (b \rightarrow S) \xrightarrow{\neg b}_p \mathit{fail}$$

$$(b;c \rightarrow S) \xrightarrow{b;c}_p S \qquad (b;c \rightarrow S) \xrightarrow{\neg b}_p \mathit{fail}$$

$$\frac{G_1 \xrightarrow{\alpha}_p S}{G_1 \mathbin{[\!]} G_2 \xrightarrow{\alpha}_p S} \qquad \frac{G_2 \xrightarrow{\alpha}_p S}{G_1 \mathbin{[\!]} G_2 \xrightarrow{\alpha}_p S}$$

$$\frac{G_1 \xrightarrow{b_1}_p \mathit{fail} \qquad G_2 \xrightarrow{b_2}_p \mathit{fail}}{G_1 \mathbin{[\!]} G_2 \xrightarrow{b_1 \wedge b_2}_p \mathit{fail}}$$

$$a \xrightarrow{a}_p \varepsilon \qquad c \xrightarrow{true;c}_p \varepsilon$$

$$\frac{S \xrightarrow{\alpha}_p S' \mid \varepsilon}{S;T \xrightarrow{\alpha}_p S';T \mid T}$$

$$\frac{G \xrightarrow{\alpha}_p S}{\mathbf{if}\ G\ \mathbf{fi} \xrightarrow{\alpha}_p S}$$

$$\frac{G \xrightarrow{\alpha}_p S \mid \mathit{fail}}{\mathbf{do}\ G\ \mathbf{od} \xrightarrow{\alpha}_p S;\ \mathbf{do}\ G\ \mathbf{od} \mid \varepsilon}$$

$$\frac{S \xrightarrow{\alpha}_p S' \mid \varepsilon}{S \parallel T \xrightarrow{\alpha}_p S' \parallel T \mid T} \qquad \frac{T \xrightarrow{\alpha}_p T' \mid \varepsilon}{S \parallel T \xrightarrow{\alpha}_p S \parallel T' \mid S}$$

$$\frac{S_1 \xrightarrow{b_1;c_1}_p S_1' \qquad S_2 \xrightarrow{b_2;c_2}_p S_2'}{S_1 \parallel S_2 \xrightarrow{b_1;c_1 \parallel b_2;c_2}_p S_1' \parallel S_2'} \quad \text{for matching } c_1, c_2$$

$$\frac{S \xrightarrow{\alpha}_p S' \mid \varepsilon}{S \xrightarrow{\alpha} S' \mid \varepsilon} \quad \text{for } \alpha \text{ an atomic action of } \mathcal{S}_c$$

Table 2. Control flow derivation rules for CF_c. S, T, possibly indexed and primed, are statements of $\mathcal{S}_c$, G, G_1, G_2 are guarded statements.

7 Behaviours of programs

We omit here indices c, w as following definitions apply to both cases.

Definition Control flow *of a program* S, denoted $CF(S)$, is an extended lts obtained by adding to CF the initial configuration S and restricting CF to the reachable part. We will also sometimes need to distinguish the single final configuration ε and, later, the deadlockable ones.

Definition For a program S and a set of states Σ the *behaviour of* S *from initial states* Σ, denoted $Beh(S, \Sigma)$, is defined as an extended lts obtained by adding to the operational semantics the set of initial configurations $\{\langle \sigma , S \rangle \mid \sigma \in \Sigma\}$ and restricting this lts to the reachable part. Again we sometimes additionally distinguish the set of final configurations $\{\langle \sigma , \varepsilon \rangle \mid \langle \sigma , \varepsilon \rangle$ is reachable from some initial configuration$\}$.

Proposition 1 *$CF(S)$ simulates $Beh(S, \Sigma)$ for any S, Σ.*

Proof. First we check that if $\langle \sigma , T \rangle$ is a configuration of $Beh(S, \Sigma)$ then T is a configuration of $CF(S)$, i.e. T is reachable in $CF(S)$ from the initial configuration S. This can be done by induction on the length of a minimal path from some initial configuration of $Beh(S, \Sigma)$ to $\langle \sigma , T \rangle$.

Consequently, $\rho(\langle \sigma , T \rangle) = T$ is a well defined function and it is easy to check that ρ is a simulation. □

Input-output semantics of programs can be derived from their operationally defined behaviours. Let $[\![S]\!] \subset St \times St$ denote the input output relation of a program S.

Definition $(\sigma, \sigma') \in [\![S]\!]$ iff $\langle \sigma' , \varepsilon \rangle$ is a (final) configuration of $Beh(S, \{\sigma\})$.

Note that for an atomic action α the defined above $[\![\alpha]\!]$ coincides with the assumed relational semantics for atomic actions, hence the same notation.

8 Annotations

We propose to characterise behaviours by *annotations.* As the name suggests, annotations are a generalisation of annotated programs or proof outlines which frequently appear in Hoare style program correctness proofs. The central role of proof outlines as a basis for Hoare style logics for concurrency was exposed in [de Roever 85] and [Schneider Andrews 86].

Let *Ind* be a set of indices.

Definition An annotation is a finite extended lts $(P, Act, \longrightarrow, i, \ldots)$ whose configurations are indexed predicates of the assertion language, $P \subset \mathcal{P} \times Ind$, the actions *Act* are the atomic actions of a considered programming language and i is a distinguished initial configuration. Apart from the initial configuration we sometimes distinguish the final or deadlockable configurations.

The indices are used to allow the same predicate to appear in different configurations of an annotation. Otherwise the indices are not important; in particular we say that a configuration is satisfied on a state σ when the predicate part of configuration is satisfied on σ.

We distinguish a class of *Floyd annotations.* Annotation A is called a Floyd annotation of program S if A is isomorphic to $CF(S)$. Such an annotation can be seen as control flow of S annotated with predicates of the assertion language which recalls Floyd's program verification technique.

Annotations underlie our proof technique for verifying partial correctness, termination and deadlock freedom. The use of such branching structures of assertions allows us to obtain soundness and completeness without resorting to auxiliary variables.

9 Expressiveness

As was pioneered in [Cook 78], the usual requirement made on the assertion language in order to assure completeness of various program logics is that the assertion language is expressive.

Definition An assertion language $\mathcal{P}$ is called expressive wrt to a class of programs $\mathcal{S}$ if for any $p \in \mathcal{P}$ and $S \in \mathcal{S}$ the weakest liberal precondition of p wrt S is definable in $\mathcal{P}$.

It is known [Olderog 83] that if $\mathcal{P}$ is a first order language with equality then expressibility of weakest liberal preconditions is equivalent to expressibility of strongest postconditions. This result applies also to concurrent programming languages whose input-output semantics is nondeterministic.

For the completeness proof in the next sections we will need a slightly different assumption than expressibility of pre-/post- conditions. Namely, for a program S and a predicate p we will require that for any configuration T of $CF(S)$ the set of states $\Sigma_T = \{\sigma \mid \langle \sigma, T \rangle$ is a configuration of $Beh(S, [\![p]\!])\}$ is definable in $\mathcal{P}$.

Σ_T contains all possible states of computation which may appear when the flow of control resides at configuration T and, roughly, corresponds to the strongest postcondition of some initial part of S. This initial part is not, in general, a substatement of statement S, nevertheless the following lemma assures that requirement of expressibility of Σ_T is not a stronger requirement than the usual expressiveness assumption.

Lemma 1 *Each Σ_T is definable in an expressive assertion language.*

Proof. Although Σ_T need not be a postcondition of p wrt a substatement of S we show that it can be still presented as a postcondition of some statement from $\mathcal{S}$ (where, to shorten the notation $\mathcal{S}$ stands for $\mathcal{S}_w$ or $\mathcal{S}_c$) so Cook's expressiveness implies definability of Σ_T.

First, we observe that Σ_T is a postcondition of some *regular program* over the set of the atomic actions of S. To see this, consider $CF(S)$ as a nondeterministic finite automaton over the alphabet consisting of atomic actions of S. Using the automata theory methods [AHU 74], with each configuration of $CF(S)$ we can associate a regular expression over the alphabet of atomic actions which represents the set (language) of action sequences appearing on all possible paths from the initial configuration of $CF(S)$ to configuration T.

Let α_T be a regular expression associated with configuration T as a result of the procedure above. Denoting by $[\![\alpha_T]\!]$ the usual relational semantics of α_T we have: $\langle\sigma, T\rangle$ is a configuration of $Beh(S, [\![p]\!])$ iff $(\sigma_0, \sigma) \in [\![\alpha_T]\!]$ for some $\sigma_0 \in [\![p]\!]$. Consequently, p_T is the strongest postcondition of p wrt $[\![\alpha_T]\!]$.

Now it is not difficult to show that that for any regular program α over the set of atomic actions of $\mathcal{S}$ there exists a $\mathcal{S}$ program defining the same input-output relation as α. We proceed by induction on the structure of α. For example the statements corresponding to α^* and belonging to $\mathcal{S}_w$ and $\mathcal{S}_c$ respectively can be defined as follows

$$b := \mathit{true}\ ;\ ((\textbf{while}\ b\ \textbf{do}\ S_\alpha) \parallel b := \mathit{false})$$

$$b := \mathit{true}\ ;\ (\textbf{do}\ b \rightarrow S_\alpha\ [\!]\ b \rightarrow b := \mathit{false}\ \textbf{od}),$$

where S_α is a statement corresponding to α and b is a fresh variable not appearing in α. □

10 Partial correctness

Extended lts's used in this section and representing behaviours, annotations and control flow are assumed to have sets of initial and final configurations. The (single) initial and final configurations of an annotation A will be denoted by i_A and f_A, respectively.

Definition A program S is partially correct wrt an input predicate p and an output predicate q, denoted $\{p\}\,S\,\{q\}$, if whenever $\sigma \models p$ and $(\sigma, \sigma') \in [\![S]\!]$ then $\sigma' \models q$.

The above defined partial correctness Hoare triple $\{p\}\,S\,\{q\}$ will always be understood here semantically, with respect to our fixed interpretation of the assertion language.

Partial correctness relates only the initial and final configurations of a behaviour. A more general notion, useful particularly in case of nonterminating programs will give some information also on intermediate stages of computation.

Definition $Beh(S, \Sigma) \models A$ (behaviour satisfies an annotation) if there exists a simulation $\rho : Beh(S, \Sigma) \rightarrow A$ such that $\sigma \models \rho(\langle\sigma, T\rangle)$ for any configuration $\langle\sigma, T\rangle$ of $Beh(S, \Sigma)$

Partial correctness can be now seen as a derived notion:

Proposition 2 *If $Beh(S, [\![p]\!]) \models A$ and $f_A \supset q$ is satisfied then $\{p\}\,S\,\{q\}$ holds.*

Proof. If $\sigma \models p$ and $(\sigma, \sigma') \in [\![S]\!]$ then σ' is a final configuration of $Beh(S, [\![p]\!])$ so $\sigma' \models f_A$.
□

We aim towards a convenient procedure for checking that a behaviour satisfies an annotation. The following definition isolates an important class of annotations.

Definition Annotation A is *locally correct* if for any transition $p \stackrel{\alpha}{\longrightarrow} p'$ of A the partial correctness triple $\{p\}\,\alpha\,\{p'\}$ holds.

For simplicity we used the semantical condition $\{p\}\,\alpha\,\{p'\}$ above (recall that Hoare triples are understood here semantically). Instead we could assume some proof system for partial correctness of atomic actions. For example such a proof system would comprise the familiar axiom for the assignment $\{p[t/x]\}\,x := t\,\{p\}$ and the consequence rule $\frac{p \supset p' \quad \{p'\}\,\alpha\,\{q'\} \quad q' \supset q}{\{p\}\,\alpha\,\{q\}}$.

Proposition 3 *If A simulates $Beh(S, \Sigma)$, A is a locally correct annotation and $\Sigma \models i_A$ then $Beh(S, \Sigma) \models A$.*

Proof. Let $\rho : Beh(S, \Sigma) \to A$ be a simulation and $\langle \sigma , T \rangle$ a configuration of $Beh(S, \Sigma)$. By induction on the length of minimal path from some initial configuration to $\langle \sigma , T \rangle$ we can show that $\sigma \models \rho(\langle \sigma , T \rangle)$. □

By Proposition 1 $CF(S)$ simulates $Beh(S, \Sigma)$ so we have

Corollary 1 *If A simulates $CF(S)$, A is a locally correct annotation and $\Sigma \models i_A$ then $Beh(S, \Sigma) \models A$.*

This is an important observation since in this case only finite transition systems are involved so checking the existence of a simulation can be done automatically.

The developed procedure for checking satisfaction of an annotation by a behaviour can be used, by Proposition 2, to derive partial correctness property.

Corollary 2 *If A simulates $Beh(S, [\![p]\!])$, A is a locally correct annotation, $p \supset i_A$ and $f_A \supset q$ are satisfied then $\{p\}\,S\,\{q\}$ holds.*

Again, $Beh(S, [\![p]\!])$ can be replaced by $CF(S)$.

Corollary 3 *If A simulates $CF(S)$, A is a locally correct annotation, $p \supset i_A$ and $f_A \supset q$ are satisfied then $\{p\}\,S\,\{q\}$ holds.*

Corollary 3 when applied to an annotation which is isomorphic to $CF(S)$ implies the soundness of Floyd's verification methodology.

Now we show completeness of the weakest of the presented verification principles, the one justified by Corollary 3 above. This is in fact equivalent to showing completeness

of Floyd's verification principle and has been already done in purely relational setting in [de Bakker Meertens 75]. Similar result can be found in [Cousot 81]. Here we are also concerned with expressiveness issues which were not addressed in the mentioned references.

Theorem 1 (Cook completeness of Floyd annotations for partial correctness) *Assume that the assertion language is expressive. If $\{p\}\, S\, \{q\}$ then there exists a locally correct annotation A simulating $CF(S)$ such that $p \supset i_A$ and $f_A \supset q$ are satisfied.*

Proof. The required annotation A is obtained by attaching to each configuration T of $CF(S)$ a predicate p_T such that $\sigma \models p_T$ iff $\langle \sigma , T \rangle$ is a configuration of $Beh(S, [\![p]\!])$. Formally, configurations of $CF(S)$ play the role of indices in A, configurations of A are pairs (p_T, T), the transition relation as well as initial and final configurations are induced from $CF(S)$. The existence of predicates p_T is guaranteed by expressiveness of the assertion language, as shown in Lemma 1.

Obviously, A simulates $CF(S)$. For the local correctness, let $p_T \xrightarrow{\alpha} p_{T'}$ be a transition in A (from now on we write p_T instead of (p_T, T)). Let $\sigma \models p_T$ and $(\sigma, \sigma') \in [\![\alpha]\!]$. By definition of p_T $\langle \sigma , T \rangle$ is a configuration of $Beh(S, [\![p]\!])$. Transition $p_T \xrightarrow{\alpha} p_{T'}$ can take place only when $T \xrightarrow{\alpha} T'$ is a transition in $CF(S)$, which together with $(\sigma, \sigma') \in [\![\alpha]\!]$ gives $\langle \sigma , T \rangle \xrightarrow{\alpha} \langle \sigma' , T' \rangle$. Thus $\langle \sigma' , T' \rangle$ is a configuration of $Beh(S, [\![p]\!])$ and, as result, $\sigma' \models p_{T'}$.

p_S is the initial configuration of A as S is the initial configuration of $CF(S)$. $\sigma \models p_S$ iff $\langle \sigma , S \rangle$ is a configuration of $Beh(S, [\![p]\!])$ so if $\sigma \models p$ then $\sigma \models p_S$ which means that p implies the initial predicate of A.

p_ε, the final configuration of A satisfies $\sigma \models p_\varepsilon$ iff $\langle \sigma , \varepsilon \rangle$ is a configuration of $Beh(S, [\![p]\!])$. We assumed partial correctness $\{p\}\, S\, \{q\}$ so for any $\langle \sigma , \varepsilon \rangle$ in $Beh(S, [\![p]\!])$ $\sigma \models q$. This assures that the final predicate of A implies q. □

Our treatment of partial correctness is not structural in whatever senses apply to the proof systems in [Owicki Gries 76, Levin Gries 81, AFR 80]. Note however, that for the above completeness proof we did not need to extend our programs with any auxiliary statements and variables nor did we need to use any extra global invariant. Moreover, we can simply extend our approach to obtain completeness for deadlock freedom and total correctness; such extensions are not known to be possible for the approaches above.

11 Deadlock freedom

Definition Behaviour $Beh(S, \Sigma)$ is *deadlock free* if for any its nonfinal configuration $\langle \sigma , T \rangle$ there exists a transition originating from $\langle \sigma , T \rangle$.

Our proof technique will take into account the fact that only some configurations can potentially become deadlocked so it is enough to assure their nonblocking.

First some notation. With each atomic action α we associate a predicate $cond(\alpha)$ defined

$$cond(a) = true$$
$$cond(b) = b$$
$$cond(\textbf{await } b \textbf{ then } a) = b$$
$$cond(b_1; c_1 \parallel b_2; c_2) = b_1 \wedge b_2$$

Now, for a configuration c of an lts we define $Act(c)$ as the set of all those actions which label transitions originating from c, and, in case the transitions are labelled with atomic actions of our programming languages, we define a predicate $cond(c) = \bigvee\{cond(\alpha) \mid \alpha \in Act(c)\}$. The empty alternative is understood here as falsehood.

Definition T is a *deadlockable configuration* of control flow if T is not ε and $cond(T)$ is not valid (i.e. $\exists\sigma \in St \; \sigma \not\models cond(T)$).

To justify the name note the following

Proposition 4 *Let $\langle\sigma, T\rangle$ be a nonfinal configuration of some behaviour $Beh(S, \Sigma)$. If there is no transition from $\langle\sigma, T\rangle$ then T is a deadlockable configuration of $CF(S)$.*

Thus to assure deadlock freedom of a behaviour it is enough to make sure that there are transitions from configurations $\langle\sigma, T\rangle$ with deadlockable T.

Deadlockable configurations can be found just by syntactical considerations or by looking at their transitions, without checking validity of predicates $cond(T)$.

Proposition 5 *The sets defined in 1 and 2 below contain all deadlockable configurations of CF_w. The sets defined in 3 and 4 contain all deadlockable configurations of CF_c.*

1. *The least such subset D of S_w that*

await b **then** $a \in D$	*for any boolean expression b and atomic action a*
$T; S \in D, T \parallel T' \in D$	*for any $T, T' \in D$, $S \in S_w$.*

2. *The set of those configurations of CF_w from which only transitions labelled with await statements originate.*

3. *The least such subset D of S_c that*

$c \in D$	*for any communication command c*
if G **fi** $\in D$	*for any guarded statement G of S_c*
do G **od** $\in D$	*for any guarded statement G with at least one communication guard*
$T; S \in D, T \parallel T' \in D$	*for any $T, T' \in D$, $S \in S_c$*

4. *The set of those configurations of CF_c from which there are no transitions labelled with atomic statements.*

Proof. Induction on the structure of configurations. □

Throughout this section the extended lts $CF(S)$ will have the initial configuration, and a distinguished set D containing all deadlockable configurations of $CF(S)$, i.e. $CF(S) = (C, Act, \longrightarrow, S, D)$.

Annotations used in this section are also assumed to have initial and deadlockable configurations, $A = (P, Act, \longrightarrow, i_A, D_A)$, where D_A is just a distinguished subset of P.

Definition An annotation A is *deadlock free* if whenever $p \in D_A$ then $p \supset cond(p)$ is satisfied.

Definition Simulation $\rho : CF(S) \rightarrow A$, where both $CF(S)$ and A are extended lts's with deadlockable configurations, is *deadlock preserving* if for any $T \in D$ $Act(\rho(T)) = Act(T)$. Recall that $\rho(T) \in D_A$ since ρ is a simulation.

Proposition 6 *If there is a deadlock preserving simulation from $CF(S)$ to a locally correct deadlock free annotation A and $p \supset i_A$ is satisfied then $Beh(S, [\![p]\!])$ is deadlock free.*

Proof. Let ρ be the deadlock preserving simulation. Let $\langle\sigma, T\rangle$ be a nonfinal configuration of $Beh(S, [\![p]\!])$ and let $\rho(T) = q$. If T is not a deadlockable configuration of $CF(S)$ then we know that there is a transition from $\langle\sigma, T\rangle$. If T is deadlockable, then q is a deadlockable configuration of A (because ρ is a simulation), so $q \supset cond(q)$ (because A is deadlock free). Since ρ is deadlock preserving $cond(q) \equiv cond(T)$. By local correctness (Corollary 1) $\sigma \models q$. Hence $\sigma \models cond(T)$ what assures that there is a transition from $\langle\sigma, T\rangle$. □

Theorem 2 (Cook completeness of Floyd annotations for deadlock freedom) *Assume that the assertion language is expressive. If $Beh(S, [\![p]\!])$ is deadlock free then there exists a locally correct deadlock free annotation A with a deadlock preserving simulation $CF(S) \rightarrow A$ such that $p \supset i_A$ is satisfied.*

Proof. Let A be the same configuration as in Theorem 1 extended with the set of deadlockable configurations induced from $CF(S)$. The isomorphism from $CF(S)$ to A is obviously a deadlock preserving simulation. In order to show that A is deadlock free we prove that for any nonfinal configuration p_T of A, i.e. where T is not ε, $p_T \supset cond(p_T)$. Let $\sigma \models p_T$. This means that $\langle\sigma, T\rangle$ is a configuration of $Beh(S, [\![p]\!])$. The behaviour is deadlock free so there is a transition $\langle\sigma, T\rangle \xrightarrow{\alpha} \langle\sigma', T'\rangle$. This can only happen when $\alpha \in Act(T)$ and $\sigma \models cond(\alpha)$. Hence $\sigma \models cond(T)$. By definition of A $cond(T) = cond(p_T)$ so $\sigma \models cond(p_T)$. □

Our approach to proving deadlock freedom involves analysing potentially blocking configurations. Proof methods proposed in related papers [Owicki Gries 76, Levin Gries 81, AFR 80] also depend on distinguishing the sets of "blocking configurations" and, essentially, assure that such configurations cannot be reached during the computation by showing that a conjunction of involved preconditions is false. In contrast to the mentioned references, however, we were able to show the completeness result.

12 Termination

Definition $Beh(S, \Sigma)$ is *terminating* if there are no infinite paths in it.

Such definition of termination is consistent with intuition only under the assumption, which we adopt, that the atomic actions represent terminating computations.

We adopt the familiar idea of well founded loop counters. We assume that the assertion language and its interpretation allow to define counters ranging over a well founded set, i.e. there are

a well founded set WF in the domain of the interpretation

an unary predicate wf such that $\sigma \models wf(l)$ iff $\sigma(l) \in WF$

a binary predicate $\prec$ interpreted as the ordering relation on WF

It is assumed that all predicates appearing in configurations of annotations used in this section have a designated free variable l which will be interpreted over WF (to assure this, each predicate p has a conjunct $wf(l)$).

Definition An annotation A is *decreasing in l* if for each transition $p \xrightarrow{\alpha} q$ the partial correctness triple $\{p\}\, \alpha\, \{\exists l'\ l' \prec l\ \wedge\ q[l'/l]\}$ is valid. The later condition means that $\forall w \in WF\ \exists v \in WF\ v \prec w\ \wedge\ \{p[w/l]\}\, \alpha\, \{q[v/l]\}$.

Proposition 7 *If A simulates $Beh(S, [\![p]\!])$, A is decreasing in l and $p \supset \exists l\ i_A$ is satisfied then $Beh(S, [\![p]\!])$ is terminating.*

Proof. Consider $\langle \sigma_0\, , S\rangle \xrightarrow{\alpha_1} \langle \sigma_1\, , S_1\rangle \xrightarrow{\alpha_2} \cdots$, a sequence of transitions of $Beh(S, [\![p]\!])$. Since A simulates $Beh(S, [\![p]\!])$ and A is a decreasing in l there exist elements $w_i \in WF$ and predicates p_i in A such that $i_A \xrightarrow{\alpha_1} p_1 \xrightarrow{\alpha_2} \cdots$ and $\sigma_0 \models i_A[w_0/l]$, $\sigma_i \models p_i[w_i/l]$ and $w_{i+1} \prec w_i$. If $\langle \sigma_0\, , S\rangle \xrightarrow{\alpha_1} \langle \sigma_1\, , S_1\rangle \xrightarrow{\alpha_2} \cdots$ were an infinite path then the sequence $w_0, w_1, \ldots$ would be infinite, which is excluded. Thus there are no infinite paths in $Beh(S, \Sigma)$ $\square$

Corollary 4 *If A simulates $CF(S)$, A is decreasing in l and $p \supset \exists l\ i_A$ is satisfied then $Beh(S, [\![p]\!])$ is terminating.*

We show completeness of our proof technique for arithmetical interpretations. Our proof follows the idea used in [Apt 84] to deal with nondeterministic programs. For the definition of arithmetical interpretation see [Harel 79]. Here we briefly summarise that the assertion language is assumed to comprise the symbols of arithmetic which get their standard meaning in the arithmetical interpretation. Additionally, there exists a predicate R which allows to encode finite sequences of elements of the interpretation domain in single elements.

Theorem 3 (Completeness of Floyd annotations for termination in arithmetical interpretations) *Assume that the interpretation is arithmetical. If $Beh(S, [\![p]\!])$ is terminating*

then there exists an annotation A such that A is decreasing in l, A simulates $CF(S)$ and $p \supset \exists l\, i_A$ is satisfied.

Proof. When we work with arithmetical interpretation, *wf* and $\prec$ will be replaced by more suggestive symbols *nat*, $<$. Obviously, the numerals contained in the domain of our arithmetical interpretation are taken as the well founded set needed for termination proofs.

Denote by x_1 the vector of all free variables that appear in S or p. Let n, l be fresh variables. For each configuration T of $CF(\mathrm{S})$ we will define a predicate $comp_T(x_1, n)$, which is satisfied of a state σ if and only if there is a path of length $\sigma(n)$ in $Beh(S, [\![p]\!])$ starting from $\langle \sigma , T \rangle$.

The required annotation A can be then obtained by attaching to each configuration T of $CF(\mathrm{S})$ a predicate

$$p_T = nat(l) \wedge (\forall n > l\ \neg comp_T(x_1, n)).$$

In other words, the above formula is satisfied of a state σ if there are no paths of length greater than $\sigma(l)$ from $\langle \sigma , T \rangle$.

Let us check that $p \supset \exists l\, p_S$ (p_S is the initial predicate of A). Let $\sigma \models p$. $Beh(S, [\![p]\!])$ is terminating so $Beh(S, \{\sigma\})$ also does not have infinite paths from $\langle \sigma , S \rangle$. Thus $Beh(S, \{\sigma\})$ is a tree (has no loops). Since $Beh(S, \{\sigma\})$ is finitely branching, by Königs Lemma $Beh(S, \{\sigma\})$ is finite and as a result there exists a bound on the length of path in $Beh(S, \{\sigma\})$, so $\sigma \models \exists l\, p_S$.

It is easy to see that A is decreasing in l.

It remains to define $comp_T$. In order to facilitate the notation let us number the configurations of $CF(\mathrm{S})$ with naturals. Let u_1, z_1 be vectors of fresh variables of the same length as x_1 and let x_0, u_0, z_0 be yet another fresh (single) variables which will be interpreted as configurations of control flow (encoded by natural numbers). Finally u, z, x will stand for vectors (u_0, u_1), (z_0, z_1), (x_0, x_1). Thus the configurations of $Beh(S, [\![p]\!])$ can be encoded in x, u, z.

We define an auxiliary predicate $trans(u, z)$ expressing 'there is a transition in $Beh(S, [\![p]\!])$ from configuration encoded in z to configuration encoded in u'. Precisely,

$$\sigma \models trans(u, z) \;\text{ iff }\; \langle \sigma[\sigma(u_1)/x_1] , \sigma(u_0) \rangle \xrightarrow{\alpha} \langle \sigma[\sigma(z_1)/x_1] , \sigma(z_0) \rangle.$$

The predicate $trans(u, z)$ is defined as a disjunction of the following formulas: for each transition $i \xrightarrow{\alpha} j$ in $CF(\mathrm{S})$ take as a disjunct

$$nat(u_0) \wedge nat(z_0) \wedge u_0 = i \wedge z_0 = j \wedge sp(x_1 = u_1 , \alpha)[z_1/x_1],$$

where $sp(x_1 = u_1 , \alpha)$ denotes the formula expressing the strongest postcondition of $x_1 = u_1$ wrt α. The postcondition is expressible because the interpretation is arithmetical.

Accordingly to the definition of arithmetical interpretation there exists a predicate R such that for any natural number n the following formula holds

$$\forall v_1 \ldots v_n \ \exists y (\forall v \ \forall i (nat(i) \wedge n \geq i) \supset (R(v,i,y) \ \equiv \ v = v_i)).$$

As a shorthand we will allow $v_1, \ldots, v_n, v$ to be tuples rather than single variables.

Predicate R enables to define $comp_i(x_1, n)$ by encoding computations in single elements of the interpretation domain. Namely,

$$\begin{array}{rcl} comp_i(x_1, n) & = & nat(n) \ \wedge \ \exists y \ (R(x,1,y) \ \wedge \ x_0 = i \ \wedge \\ & & \forall k < n \ \exists u \exists z \ R(u,k,y) \wedge R(z,k+1,y) \wedge trans(u,z)). \end{array}$$

□

13 Examples

We work out one of standard examples, partitioning of a set, [Dijkstra 82, AFR 80, Barringer 85]. Given two disjoint sets of integers S_0 and T_0, $S_0 \cup T_0$ has to be partitioned into two subsets S and T such that $|S| = |S_0|$, $|T| = |T_0|$ and $max(S) < min(T)$.

The program *Set_Part* presented in Table 3 is a solution of the above problem. In contrast to [Dijkstra 82, AFR 80] we do not adopt the distributed termination convention but we achieve an equivalent effect by using the same boolean condition for termination of loops in both parallel components, i.e. we claim that our solution behaves in the same way as its counterpart which uses the distributed termination convention.

```
Set_Part ≡ Small || Large

Small ≡  mx := max(S) ;                      Large ≡
         ch ! mx ;                                    ch ? y ;
         S := S − {mx} ;                              T := T ∪ {y} ; mn := min(T) ;
         ch ? x ;                                     ch ! mn ;
         S := S ∪ {x} ; mx := max(S) ;                T := T − {mn} ;
         do                                           do
         mx > x ; ch ! mx →                           mx > x ; ch ? y →
               S := S − {mx} ;                              T := T ∪ {y} ; mn:= min(T) ;
               ch ? x ;                                     ch ! mn ;
               S := S ∪ {x} ; mx := max(S)                  T := T − {mn}
          od                                           od
```

Table 3. Program for set partitioning.

We will show that *Set_Part* is partially correct and terminating (deadlock freedom is trivial).

The partial correctness property we want to prove is $\{p_0\}$ *Set_Part* $\{p_{14}\}$, where

$p_0 \equiv |S| = |S_0| \wedge |T| = |T_0| \wedge S \cup T = S_0 \cup T_0 \wedge S \cap T = \emptyset$

$p_{14} \equiv p_0 \wedge max(S) < min(T)$.

We show this property following the pattern given in Corollary 3. We give a locally correct annotation A_{pc} which simulates $CF(Set_Part)$ and has p_0 and p_{14} as initial and final predicates, respectively.

Figure 1 contains the graphical presentation of transitions of A_{pc}. The predicates $p_1, \ldots, p_{14}$ are listed below:

$$\begin{array}{lll}
p_1 & \equiv & mx \in S \wedge p_0 \\
p_2 & \equiv & mx \in S \wedge mx = y \wedge p_0 \\
p_3 & \equiv & mx \in S \wedge \\
 & & |S| = |S_0| \wedge |T| = |T_0| + 1 \wedge (S - \{mx\}) \cup T = S_0 \cup T_0 \wedge (S - \{mx\}) \cap T = \emptyset \\
p_4 & \equiv & mn = min(T) \wedge p_3 \\
p_5 & \equiv & y \notin T \wedge \\
 & & |S| = |S_0| - 1 \wedge |T| = |T_0| \wedge S \cup T \cup \{y\} = S_0 \cup T_0 \wedge S \cap (T \cup \{y\}) = \emptyset \\
p_6 & \equiv & |S| = |S_0| - 1 \wedge |T| = |T_0| + 1 \wedge S \cup T = S_0 \cup T_0 \wedge S \cap T = \emptyset \\
p_7 & \equiv & mn = min(T) \wedge p_6 \\
p_8 & \equiv & mn = x = min(T) \wedge p_6 \\
p_9 & \equiv & mn = x = min(T) \wedge \\
 & & |S| = |S_0| \wedge |T| = |T_0| + 1 \wedge S \cup (T - \{mn\}) = S_0 \cup T_0 \wedge S \cap (T - \{mn\}) = \emptyset \\
p_{10} & \equiv & mx = max(S) \wedge p_9 \\
p_{11} & \equiv & x < min(T) \wedge x \notin S \wedge \\
 & & |S| = |S_0| - 1 \wedge |T| = |T_0| \wedge S \cup \{x\} \cup T = S_0 \cup T_0 \wedge (S \cup \{x\}) \cap T = \emptyset \\
p_{12} & \equiv & x < min(T) \wedge p_0 \\
p_{13} & \equiv & x < min(T) \wedge mx = max(S) \wedge p_0
\end{array}$$

Local correctness of A_{pc} can be easily checked by examining all transitions. It is also easy to see that A_{pc} simulates $CF(Set_Part)$, besides, checking this could be automatised as finite transition systems are involved. In order to avoid the above list of predicates we could define just p_{13} and say that the remaining ones can be obtained by pushing p_{13} backwards, i.e. by doing appropriate substitutions in p_{13}. We listed explicitly all the predicates in a readable form because we believe they give information on how the program behaves.

Next, we show that $Beh(Set_Part, [\![true]\!])$ is terminating. We follow the pattern set in Corollary 4. The transitions of the required annotation A_t are again given in Figure 1. A counter l will appear in predicates of A_t ranging over the well founded set of integers greater than -9. The idea behind A_t is that during execution of *Set_Part* either the action $S := S - \{mx\}$ decreases the number of elements of S which are greater than $min(T)$, or the next execution of the loop guards terminates both loops. The predicates

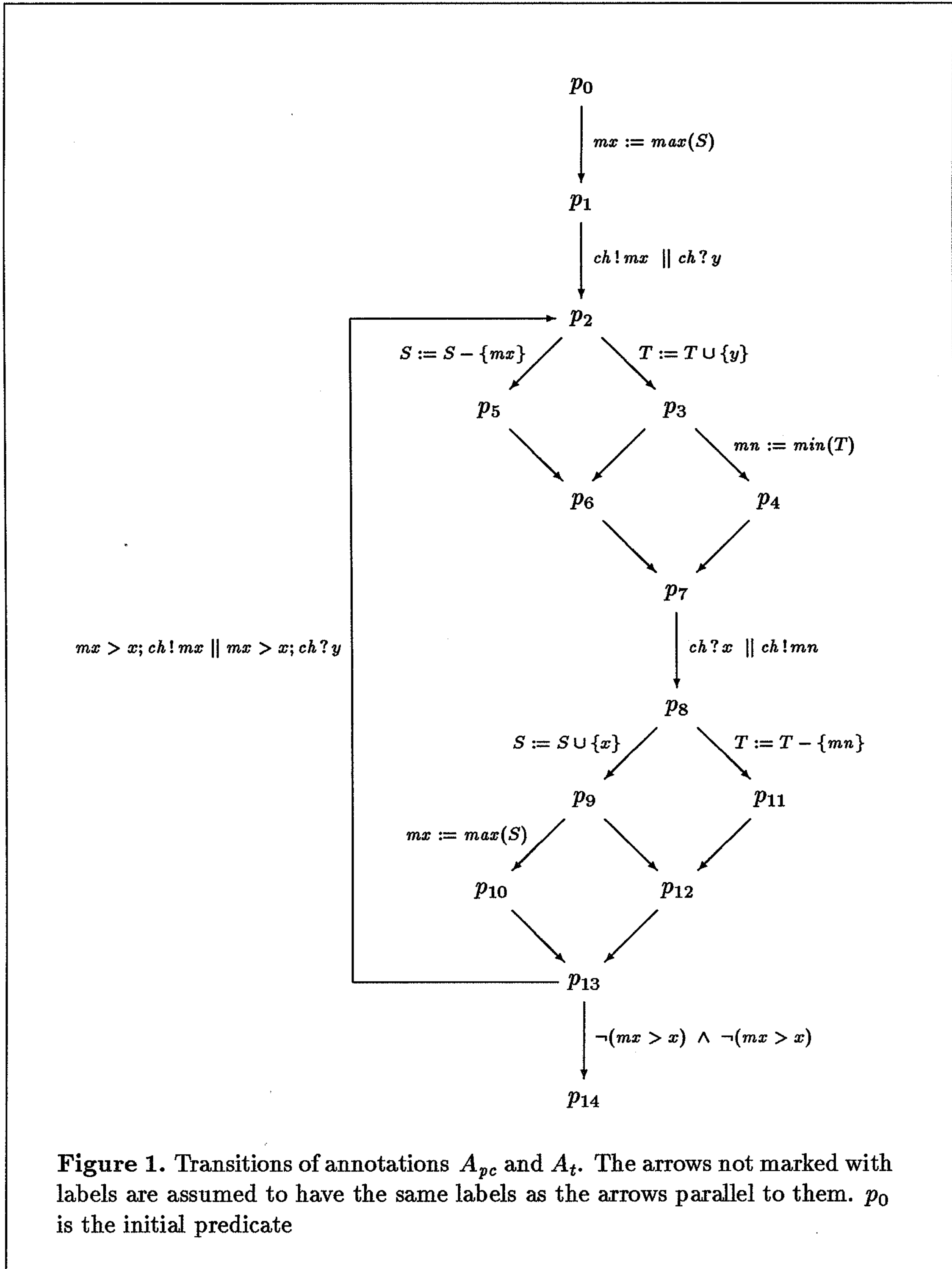

Figure 1. Transitions of annotations A_{pc} and A_t. The arrows not marked with labels are assumed to have the same labels as the arrows parallel to them. p_0 is the initial predicate

appearing in A_t handle those two cases differently.

In order to shorten the notation define

$$d(S,T) = 8 \cdot |\{s \in S \mid s > min(T)\}|, \quad \text{where } |X| \text{ denotes the cardinality of a set } X$$

The predicates of A_t are listed below:

$$
\begin{array}{lll}
p_0 & \equiv & l = d(S,T) + 2 \\
p_1 & \equiv & l = d(S,T) + 1 \ \wedge \ mx = max(S) \\
p_2 & \equiv & l = d(S,T) \qquad \wedge \ mx = y = max(S) \\
p_3 & \equiv & l = d(S,T) - 1 \ \wedge \ mx = max(S) \\
p_4 & \equiv & l = d(S,T) - 2 \ \wedge \ mx = max(S) \ \wedge \ mn = min(T) \\
p_5 & \equiv & (l = d(S,T) + 7 \ \vee \ (max(S) \leq min(T) \ \wedge \ l = -1)) \ \wedge \ y > max(S) \\
p_6 & \equiv & (l = d(S,T) + 6 \ \vee \ (max(S) \leq min(T) \ \wedge \ l = -2)) \\
p_7 & \equiv & (l = d(S,T) + 5 \ \vee \ (max(S) \leq min(T) \ \wedge \ l = -3)) \ \wedge \ mn = min(T) \\
p_8 & \equiv & (l = d(S,T) + 4 \ \vee \ (max(S) \leq x \ \wedge \ l = -4)) \ \wedge \ x = mn = min(T) \\
p_9 & \equiv & (l = d(S,T) + 3 \ \vee \ (max(S) = x \ \wedge \ l = -5)) \\
p_{10} & \equiv & (l = d(S,T) + 2 \ \vee \ (mx = x \ \wedge \ l = -6)) \ \wedge \ mx = max(S) \\
p_{11} & \equiv & (l = d(S,T) + 3 \ \vee \ (max(S) \leq x \ \wedge \ l = -5)) \ \wedge \ x < min(T) \\
p_{12} & \equiv & (l = d(S,T) + 2 \ \vee \ (max(S) = x \ \wedge \ l = -6)) \\
p_{13} & \equiv & (l = d(S,T) + 1 \ \vee \ (mx = x \ \wedge \ l = -7)) \ \wedge \ mx = max(S) \\
p_{14} & \equiv & l = -8
\end{array}
$$

Annotation A_t defined above is decreasing in l. For example, let us check that for the transition $p_2 \xrightarrow{S:=S-\{mx\}} p_5$ the triple $\{p_2\}\ S := S - \{mx\}\ \{\exists l'\ l' < l \ \wedge \ p_5[l'/l]\}$ is satisfied, i.e.

$$
\begin{array}{c}
\{l = d(S,T) \ \wedge \ mx = y = max(S)\} \\
S := S - \{mx\} \\
\{\exists l'\ l < l' \ \wedge \ (l' = d(S,T) + 7 \ \vee \ (max(S) < min(T) \ \wedge \ l' = -1)) \ \wedge \ y > max(S)\}.
\end{array}
$$

This can be readily verified by considering two cases: either $d(S,T) > 0$ and then the statement $S := S - \{mx\}$ decreases $d(S,T)$ (at least by 8) or $d(S,T) = 0$ what implies $max(S) < min(T)$ after the execution of $S := S - \{mx\}$ assuring that l' can be taken equal to -1.

Finally, for each $i = 1, \ldots, 14$ $\ p_i \supset (l > -9)$ what assures that l ranges over a well founded set.

As the next example we take the mutual exclusion problem. Processes $S_1, \ldots, S_n$ run in parallel, each alternatively executing a critical and a noncritical section. No two processes are ever allowed to execute their critical sections concurrently. A simple solution to this problem expressed in the language $\mathcal{S}_w$ and using the await statement to synchronise S_i is presented in Table 4.

$S \;=\; S_1 \,\|\, \ldots \,\|\, S_n$

$S_i \;=\;$ **while** *true* **do**
begin
N_i ;
await b **then** $b := false$;
C_i ;
$b := true$
end

N_i	-	noncritical part of S_i
C_i	-	critical section of S_i
b	-	a variable not appearing in any of C_i, N_i.

Table 4.

We show mutual exclusion in executing critical sections and deadlock freedom of $Beh(S, [\![b]\!])$, the behaviour of our solution S initiated with $b = true$. The annotations needed to verify these properties, A_{me} and A_{df}, respectively, are defined in Figure 2 and Figure 3.

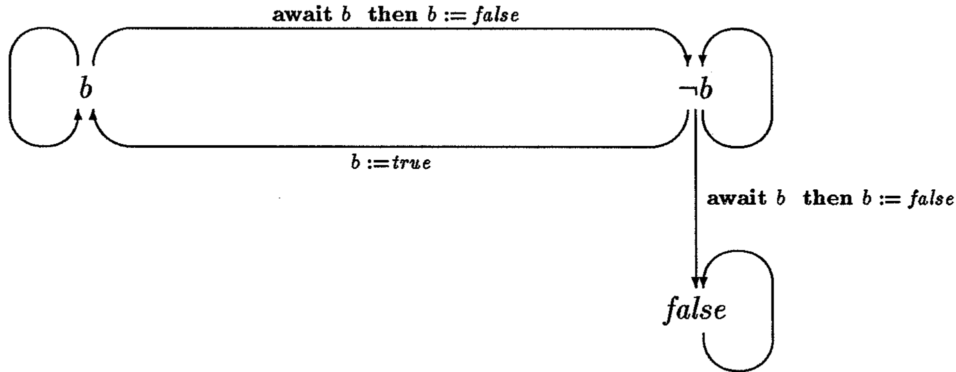

Figure 2. Annotation A_{me}. The not marked arcs represent transitions labelled with all other atomic actions of S (i.e. different from **await** b **then** $b := false$ and $b := true$). b is the initial predicate.

Both A_{me} and A_{df} are locally correct: this can be checked by examining all transitions and taking into account that the actions of N_i and C_i do not change b.

Before we prove mutual exclusion let us formalise this property first. "Mutual exclusion" means that some configurations of control flow never appear in behaviours of S. Those prohibited configurations have the shape $T_{i_1} \,\|\, \ldots \,\|\, T_{i_k}$, $k \leq n$, where each T_{i_j} is a configuration of $CF(S_{i_j})$ and at least two of T_{i_j} represent control flow positions inside the critical sections, or to put it formally $T_{i_j} \equiv C'_{i_j}$; $b := false$; S_{i_j}, where C'_{i_j} is a

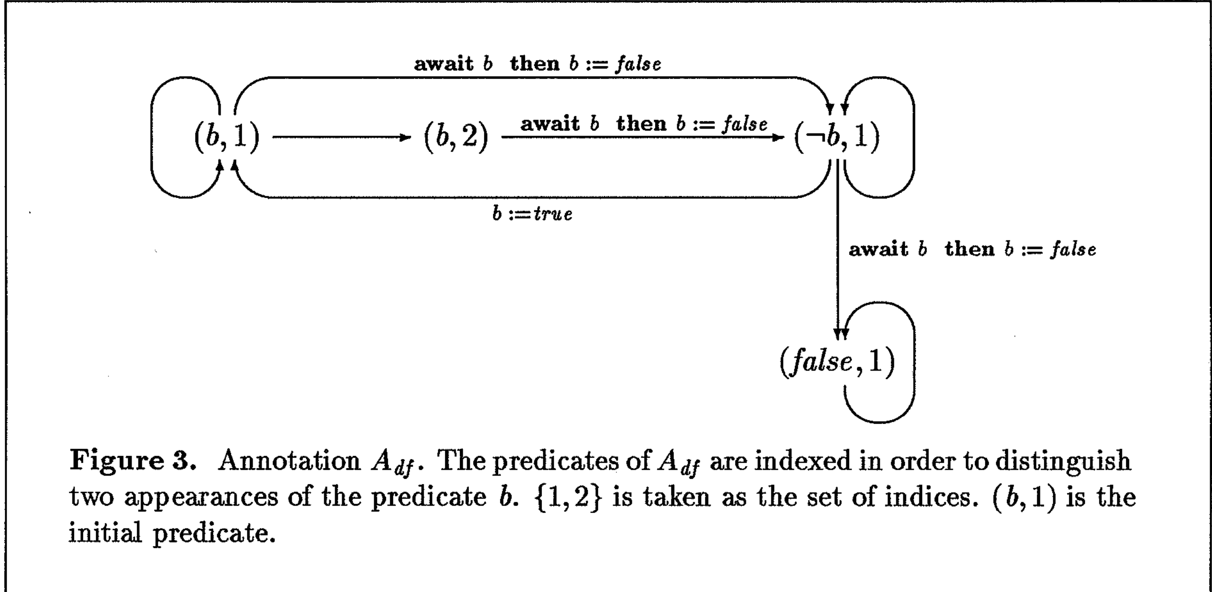

Figure 3. Annotation A_{df}. The predicates of A_{df} are indexed in order to distinguish two appearances of the predicate b. $\{1,2\}$ is taken as the set of indices. $(b,1)$ is the initial predicate.

configuration of $CF(C_{i_j})$. Let *Crit* be the set of all such prohibited configurations of $CF(S)$.

The proof of mutual exclusion goes now as follows. First, we claim that there exists a simulation $\rho : CF(S) \to A_{me}$ such that $\rho(T) = \mathit{false}$ for all $T \in \mathit{Crit}$. This can be easily argued or checked mechanically for a fixed n. A_{me} is locally correct, so $Beh(S, [\![b]\!]) \models A_{me}$ by Proposition 1. Hence $\sigma \models \rho(\langle \sigma , T \rangle)$ for configurations $\langle \sigma , T \rangle$ of $Beh(S, [\![b]\!])$, so there can be no configurations $\langle \sigma , T \rangle$ in $Beh(S, [\![b]\!])$ with $T \in \mathit{Crit}$ what is the required mutual exclusion property.

For the deadlock freedom, we distinguish the set of deadlockable configurations of $CF(S)$ as described in point 2 of Proposition 5 assuming for simplicity that there are no await statements in N_i and C_i. We take $(b,2)$ as a single deadlockable configuration of A_{df}. Again it can be argued that there is a deadlock preserving simulation from $CF(S)$ to A_{df}. Simply, all deadlockable configurations of $CF(S)$ have to be mapped into $(b,2)$. A_{df} is deadlock free because $b \supset \mathit{cond}((b,2)) = b$ so by Proposition 6 we declare that S is deadlock free.

Note that by showing the mutual exclusion property we have demonstrated that out proof technique has a larger scope of applications than verification of partial correctness deadlock freedom and termination.

Acknowledgements

I would like to thank Colin Stirling for bringing deadlock freedom and total correctness into my attention and also for many useful discussions and detailed comments on the

drafts of this paper. This work was supported by University of Edinburgh Studentship and ORS Award. University of Gdańsk granted a leave from didactic duties.

References

[AHU 74] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Company, 1974.

[Apt 84] K.R. Apt, Ten years of Hoare's Logic: a survey — Part II: nondeterminism, Theoretical Computer Science 28, pp. 83-109 (1984).

[AFR 80] K.R. Apt, N. Francez, W.P. de Roever, A proof system for communicating sequential processes, TOPLAS vol. 2(3), pp. 359–384 (1980).

[de Bakker Meertens 75] J.W. de Bakker, L.G.T. Meertens, On the completeness of the inductive assertion method, Journal of Computer and System Sci. 11, pp. 323-357 (1975).

[Barringer 85] H. Barringer, A Survey of Verification Techniques for Parallel Programs, LNCS 191, Springer-Verlag, 1985.

[Brookes 85] S.D. Brookes, An axiomatic treatment of a parallel programming language, in: Proceedings 1985 Logics of Programs Conference, Brooklyn, LNCS 193, Springer, 1985.

[Brookes 86] S.D. Brookes, A semantically based proof system for partial correctness and deadlock in CSP, in: Proceedings 1986 LICS, pp. 58-65

[Cook 78] S.A. Cook, Soundness and completeness of an axiom system for program verification, SIAM Journal of Computing 7(1), pp. 70-90 (1978).

[Cousot 81] P. Cousot, Semantic foundations of program analysis, in: Program Flow Analysis. Theory and Applications. (S.S. Muchnick, N.D. Jones, Eds.), Prentice Hall, 1981 .

[Dijkstra 82] E.W. Dijkstra, A correctness proof for communicating processes — A small exercise, in: Selected writtings on Computing: A Personal Perspective. Springer Verlag, 1982.

[Flon Suzuki 81] L. Flon, N. Suzuki, The total correctness of parallel programs, SIAM Journal of Computing 10(2), pp. 227-246 (1981).

[Floyd 67] R. W. Floyd, Assigning meanings to programs, in: Mathematical Aspects of Computer Science. (J.T. Schwartz, Ed.), pp. 19-32, Proceedings Symposium in Applied Mathematics, vol. 19, American Math. Soc., Providence, 1967.

[Harel 79] D. Harel, First-Order Dynamic Logic, Springer-Verlag, 1979.

[Lamport 80] L. Lamport, The "Hoare Logic" of concurrent programs, Acta Informatica 14, pp. 21-37 (1980).

[Levin Gries 81] G.M. Levin, D. Gries, A proof technique for communicating sequential processes, Acta Informatica 15, pp. 159–172 (1981).

[Misra Chandy 81] J. Misra, K.M. Chandy, Proofs of Networks of Processes, IEEE Transactions on Software Engineering, vol. SE-7 (4), pp. 417–426, 1981.

[Olderog 83] Ernst-Rüdiger Olderog, On the notion of expressiveness and the rule of adaptation, TCS 24, pp. 337-347 (1983).

[Owicki Gries 76] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs I, Acta Informatica vol. 6, pp. 319–340 (1976).

[Plotkin 81] G.D. Plotkin, A Structural Approach to Operational Semantics, DAIMI Report FN–19, Aarhus University, 1981.

[de Roever 85] W.P. de Roever, The quest for compositionality: a survey of assertion based proof systems for concurrent programs. Part 1, Technical Report RUU-CS-85-2, University of Utrecht, 1985.

[Schneider Andrews 86] F.B. Schneider, G.R. Andrews, Concepts for concurrent programming, in: Current Trends in Concurrency. Overviews and Tutorials. (J.W. de Bakker, W.P. de Roever, G. Rozenberg, Eds.), LNCS 224, Springer, 1986.

[Soundararajan 83] N. Soundararajan, Correctness proofs of CSP programs, Theoretical Computer Science vol. 24(2), pp. 131–141 (1983).

[ZRE 85] J. Zwiers, W.P. de Roever, P. van Emde Boas, Compositionality and concurrent networks: soundness and completeness of a proof system, LNCS 194, pp. 509–519, Springer.