USING TRANSFORMATIONS TO VERIFY PARALLEL PROGRAMS

Ernst-Rüdiger Olderog* and Krzysztof R. Apt

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188

TR-89-36 November 1989

* Department of Computer Science, University of Oldenburg, 2900 Oldenburg, Federal Republic of Germany.

Using Transformations to Verify Parallel Programs

Ernst-Rüdiger Olderog
Department of Computer Science
University of Oldenburg
2900 Oldenburg
Federal Republic of Germany

Krzysztof R. Apt
Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam
The Netherlands
and
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712–1188
U.S.A.

Abstract We argue that the verification of parallel programs can be considerably simplified by using program transformations. We illustrate this approach by proving correctness of two parallel programs under the assumption of fairness: asynchronous fixed point computation and parallel zero search.

1 Introduction

The aim of this paper is to show how program transformations can simplify the task of proving parallel programs with shared variables correct. To this end, we present four transformations all of which preserve partial and total correctness and fairness, and which consequently can be used in proofs of these correctness properties.

The first transformation links parallel programs to nondeterministic sequential ones. This is as in the work of Ashcroft and Manna [1971], Flon and Suzuki [1981] and, more recently, Back [1989] and Chandy and Misra [1988]. However, to avoid the introduction of auxiliary variables that would destroy the program structure, we present this transformation only for a restricted class of parallel programs.

To enhance the usefulness of this transformation, we combine it with two transformations on parallel programs which introduce more points of interference. These transformations are inspired by Lipton [1975]. Whereas Lipton considered only ordinary termination proofs, we deal here also with fairness.

Fair termination is proved on the level of nondeterministic programs by reducing it to ordinary termination with the help of a fourth transformation due to Apt and Olderog [1983] which makes use of random assignments.

Considered in isolation these transformations look very simple but when combined they can substantially reduce the task of verification. This reduction is achieved by delaying the assertional correctness proof as much as possible, viz. after a stepwise transformation of the original parallel program into a well-structured nondeterministic program. The proposed transformations can also be used to construct parallel programs from nondeterministic ones.

We illustrate our approach by proving total correctness of two parallel programs under the assumption of fairness: asynchronous fixed point computation and parallel zero search.

There are two alternatives to these correctness proofs. The first one is to use the transformational approach to fairness in parallel programs presented in Olderog and Apt [1988]. It calls for proving ordinary total correctness of a transformed parallel program simulating the fair computations of the original program. Another possibility is to first translate the original program directly into a nondeterministic program as in Flon and Suzuki [1981] and then use one of the available methods for proving correctness of a nondeterministic program under the assumption of fairness (see Francez [1986] for their overview).

In both cases the verification becomes extremely tedious and complicated because the transformations of Olderog and Apt [1988] and Flon and Suzuki [1981] introduce auxiliary variables that destroy the structure of the original program.

Besides the two parallel programs we also prove correctness of the program transformations themselves (except of the one taken from Apt and Olderog [1983]). These proofs appear in the appendix to our paper and are based on a simple operational program semantics due to Hennessy and Plotkin [1979].

2 Preliminaries

Throughout this paper we mean by a *parallel program* a program of the form

$$S_0; [S_1 \parallel \ldots \parallel S_n]$$

where each S_i is a **while**-program. We call S_0 an *initialization statement* and each S_i for $i > 0$ a *component program*. Within the component programs we additionally allow *atomic regions*. Syntactically, these are loop free **while**-programs enclosed in angle brackets $\langle$ and $\rangle$. Sometimes we write $[\|_{i=1}^{n} S_i]$ instead of $[S_1 \parallel \ldots \parallel S_n]$. Note that $S_1, \ldots, S_n$ *may share* variables.

Intuitively, an execution of $[S_1 \| \ldots \| S_n]$ is obtained by interleaving the atomic, i.e. non-interruptible steps in the executions of the components $S_1, \ldots, S_n$. By definition, Boolean expressions, assignments, the *skip* statement and atomic regions are all evaluated or executed as atomic steps. As atomic regions are

required to be loop free, their execution is guaranteed to terminate. An interleaved execution of $[S_1||\ldots||S_n]$ terminates if and only if the individual execution of each component terminates.

For convenience, we identify

$$\langle A \rangle \equiv A$$

if A is an assignment or *skip*.

A *state* is either a *proper state*, i.e. a mapping from variables to values, or a special symbol $\perp$ denoting divergence.

We consider here three semantics of parallel programs, all referring to an interleaving model of execution. Given a parallel program S we distinguish:

- partial correctness semantics $\mathcal{M}[\![S]\!]$,
- total correctness semantics $\mathcal{M}_{tot}[\![S]\!]$,
- fair parallelism semantics $\mathcal{M}_{fair}[\![S]\!]$.

In the partial correctness semantics, given an initial proper state, only the final proper states are recorded. In the total correctness semantics additionally a possibility of divergence is recorded as $\perp$. Finally, the fair parallelism semantics is like the total correctness semantics but only the fair computations are taken into account. A computation of a parallel program is called *fair* if each component that has not yet terminated is eventually activated again. In particular, every finite computation is fair.

For details concerning the semantics we refer to the appendix. Each of these three semantics induces a corresponding notion of program correctness. We thus distinguish between

- partial correctness $\models$,
- total correctness $\models_{tot}$,
- fair total correctness $\models_{fair}$.

Each of these correctness notions refers to a *correctness formula*, i.e. a construct of the form $\{p\}\ S\ \{q\}$ where p and q are assertions and S a program. We assume from the reader some knowledge of the basic concepts on program verification.

3 Transformations

We now present four program transformations. The first of them transforms a nondeterministic program in the sense of Dijkstra [1975] into a parallel program. We study here only *one level* nondeterministic programs, i.e. programs of the form

$$S \equiv S_0;\ \mathbf{do}\ \Box_{i=1}^{n}\ B_i \rightarrow S_i\ \mathbf{od}$$

where the subprograms S_i are loop free **while**-programs.

For these programs we refer to the same three semantics and program correctness notions as those introduced above. The notion of a fair computation is obtained here by considering enabled branches of a **do**-loop instead of nonterminated components of a parallel program.

Theorem 1 (Parallelization) Consider a one level nondeterministic program

$$S \equiv S_0;\ \mathbf{do}\ \Box_{i=1}^{n}\ B \rightarrow S_i\ \mathbf{od},$$

the parallel program

$$T \equiv S_0;\ [\|_{i=1}^{n}\ \mathbf{while}\ B\ \mathbf{do}\ \langle S_i \rangle\ \mathbf{od}]$$

and two assertions p and q. Suppose that for every $i \in \{1, \ldots, n\}$

$$\models_{tot} \{q \wedge \neg B\}\ S_i\ \{q \wedge \neg B\}.$$

Then

$$\models \{p\}\ S\ \{q\} \text{ iff } \models \{p\}\ T\ \{q\}$$

and analogously for $\models_{tot}$ and $\models_{fair}$.

Proof. See the appendix. □

The Parallelization Theorem transforms **do**-loops with identical guards into parallel programs of a very restricted format. In particular, components that are **while**-loops consisting only of a single atomic region are rare in practice. To enhance the usefulness of the Parallelization Theorem we shall combine its application with two additional transformations of parallel programs which introduce more points of interference. These transformations are inspired by Lipton [1975].

We say that two programs are *disjoint* if none of the variables which can be changed by one of them appears in the other. We say that a Boolean expression B is *disjoint from a program* S if none of the variables which can be changed by S appears in B.

The next transformation reduces the size of atomic regions.

Theorem 2 (Atomicity) Consider a parallel program $S \equiv S_0;\ [S_1 \| \ldots \| S_n]$. Let T result from S by replacing in one of its components, say S_i with $i > 0$, either

- an atomic region $\langle R_1;\ R_2\rangle$ where one of the R_l's ($l \in \{1,2\}$) is disjoint from all components S_j with $j \neq i$ by

$$\langle R_1\rangle;\ \langle R_2\rangle$$

or

- an atomic region $\langle \textbf{if } B \textbf{ then } R_1 \textbf{ else } R_2 \textbf{ fi}\rangle$ where B is disjoint from all components S_j with $j \neq i$ by

$$\textbf{if } B \textbf{ then } \langle R_1\rangle \textbf{ else } \langle R_2\rangle \textbf{ fi}.$$

Then the programs S and T have the same semantics, i.e.,

$$\mathcal{M}[\![S]\!] = \mathcal{M}[\![T]\!],$$

and analogously for $\mathcal{M}_{tot}$ and $\mathcal{M}_{fair}$.

Proof. See the appendix. □

Corollary 3 (Atomicity) Under the assumptions of the Atomicity Theorem, for all assertions p and q

$$\models \{p\}\ S\ \{q\} \text{ iff } \models \{p\}\ T\ \{q\}$$

and analogously for $\models_{tot}$ and $\models_{fair}$. □

The Atomicity Theorem describes a simple but very useful transformation on parallel programs. The given program S has a coarser grain of atomicity than T — it has less points for possible interference among its components and thus admits fewer computations. Therefore S is easier to prove correct than T, either directly by using a proof systems for proving correctness of parallel programs or, if possible, by using the Parallelization Theorem. On the other hand, the resulting program T has a finer grain of atomicity and is thus more realistic than S.

The third transformation moves initializations inside the parallel composition.

Theorem 4 (Initialization) Consider a parallel program of the form

$$S \equiv S_0;\ R_0;\ [S_1 \| \ldots \| S_n].$$

Suppose that for some index $i \in \{1,\ldots,n\}$ the initialization part R_0 is disjoint from all component programs S_j with $j \neq i$. Then the program

$$T \equiv S_0;\ [S_1 \| \ldots \| R_0;\ S_i \| \ldots \| S_n]$$

has the same semantics as S, i.e.

$$\mathcal{M}[\![S]\!] = \mathcal{M}[\![T]\!],$$

and analogously for $\mathcal{M}_{tot}$ and $\mathcal{M}_{fair}$.

Proof. See the appendix. □

Corollary 5 (Initialization) Under the assumptions of the Initialization Theorem, for all assertions p and q

$$\models \{p\}\ S\ \{q\} \text{ iff } \models \{p\}\ T\ \{q\}$$

and analogously for $\models_{tot}$ and $\models_{fair}$. □

Again, the given program S admits fewer computations and is easier to prove correct whereas the transformed program T has more points for possible interference.

To reason about fair total correctness of nondeterministic programs, we use a program transformation, originally proposed in Apt and Olderog [1983], which reduces this notion of correctness to ordinary total correctness. This transformation embeds into a given nondeterministic program an abstract scheduler that implements the fairness policy. This scheduler initializes, reads and updates private variables by using *random assignments* of the form

$$z :=?$$

which assign an arbitrary non-negative integer to an integer variable z.

Theorem 6 (Fairness) Consider a one level nondeterministic program

$$S \equiv S_0;\ \mathbf{do}\ \Box_{i=1}^{n}\ B_i \rightarrow S_i\ \mathbf{od}.$$

Let T be obtained from S as follows:

$$\begin{aligned} T \equiv\ & INIT;\ S_0; \\ & \mathbf{do}\ \Box_{i=1}^{n}\ B_i \wedge SCH_i \rightarrow UPDATE_i;\ S_i\ \mathbf{od} \end{aligned}$$

where for variables $z_1, \ldots, z_n$ not occurring in S

$INIT \equiv z_1 :=?;\ \ldots;\ z_n :=?,$

$SCH_i \equiv z_i = min\{z_k \mid k \in \{1,\ldots,n\} \text{ and } B_k\},$

$UPDATE_i \equiv z_i :=?;$
for all $j \in \{1,\ldots,n\} - \{i\}$ **do**
if B_j **then** $z_j := z_j - 1$ **fi**
od.

Then

$$\mathcal{M}_{fair}[\![S]\!] = \mathcal{M}_{tot}[\![T]\!] \textbf{ mod } \{z_1,\ldots,z_n\},$$

where the **mod**-notation means that the final states agree *modulo* $\{z_1,\ldots,z_n\}$, i.e. on all variables except $z_1,\ldots,z_n$.

Proof. See Apt and Olderog [1983]. □

Corollary 7 (Fairness) Under the assumptions of the Fairness Theorem, for all assertions p and q which do not contain the variables $z_1,\ldots,z_n$

$$\models_{fair} \{p\}\ S\ \{q\} \text{ iff } \models_{tot} \{p\}\ T\ \{q\}$$

□

4 Asynchronous fixed point computation

As a first application of the Parallelization Theorem let us consider the problem of asynchronous fixed point computation studied in Apt and Olderog [1983]. We considered there a monotonic operator $F : L^n \to L^n$ on the n-fold product of a complete lattice L with the *finite chain property* (no infinite strictly growing sequence exists). We proved that under the assumption of fairness the nondeterministic program

$$S \equiv \textbf{do } \Box_{i=1}^n\ \bar{x} \neq F(\bar{x}) \to x_i := F_i(\bar{x}) \textbf{ od}$$

computes the least fixed point of F:

$$\models_{fair} \{\bar{x} = \bot\}\ S\ \{\bar{x} = \mu F\}.$$

F_i stands for the i-th component function $F_i : L^n \to L$ of F defined by

$$F_i(x_1,\ldots,x_n) = y_i \text{ if } F(x_1,\ldots,x_n) = (y_1,\ldots,y_n),$$

$\bar{x}$ abbreviates $(x_1,\ldots,x_n)$ and $\bot$ denotes the least element in L^n.

Now we wish to parallelize S. To this end, we check the condition of the Parallelization Theorem, i.e. whether

$$\models_{tot} \{\bar{x} = \mu F \wedge \bar{x} = F(\bar{x})\}\ x_i := F_i(\bar{x})\ \{\bar{x} = \mu F \wedge \bar{x} = F(\bar{x})\} \qquad (1)$$

for all $i \in \{1, \ldots, n\}$. By the definition of F_i, the precondition $\bar{x} = F(\bar{x})$ implies that for all $i \in \{1, \ldots, n\}$

$$x_i = F_i(\bar{x}).$$

Hence for all $i \in \{1, \ldots, n\}$ the value of x_i remains unchanged under the assignment $x_i := F_i(\bar{x})$. Thus (1) holds and the Parallelization Theorem yields that under the assumption of fairness the parallel program

$$T \equiv [\|_{i=1}^{n}\ \textbf{while}\ \bar{x} \neq F(\bar{x})\ \textbf{do}\ x_i := F_i(\bar{x})\ \textbf{od}]$$

also computes the least fixed point of F:

$$\models_{fair} \{\bar{x} = \bot\}\ T\ \{\bar{x} = \mu F\}.$$

5 Parallel zero search

The next example illustrates how all four transformations can be combined to verify a parallel program. We prove that under the assumption of fairness the parallel program

$$S \equiv found := \textbf{false};\ [S_1 \| S_2]$$

with

$S_1 \equiv$ $x := 0$;
while $\neg found$ **do**
$x := x + 1$;
if $f(x) = 0$ **then** $found := \textbf{true}$ **fi**
od

and

$S_2 \equiv$ $y := 1$;
while $\neg found$ **do**
$y := y - 1$;
if $f(y) = 0$ **then** $found := \textbf{true}$ **fi**
od

finds a zero of the function f provided such a zero exists:

$$\models_{fair} \{\exists u : f(u) = 0\}\ S\ \{f(x) = 0 \vee f(y) = 0\}. \qquad (2)$$

We proceed in 5 steps.

Step 1. Simplifying the program

We first use the Atomicity Corollary and Initialization Corollary and reduce the original problem (2) to the following claim

$$\models_{fair} \{\exists u : f(u) = 0\}\ T\ \{f(x) = 0 \vee f(y) = 0\} \tag{3}$$

where

$$T \equiv found := \textbf{false};\ x := 0;\ y := 1;\ [T_1 \| T_2]$$

with

$$T_1 \equiv \textbf{while}\ \neg found\ \textbf{do}\ \langle\ x := x + 1;\ \textbf{if}\ f(x) = 0\ \textbf{then}\ found := \textbf{true}\ \textbf{fi}\rangle\ \textbf{od}$$

and

$$T_2 \equiv \textbf{while}\ \neg found\ \textbf{do}\ \langle\ y := y - 1;\ \textbf{if}\ f(y) = 0\ \textbf{then}\ found := \textbf{true}\ \textbf{fi}\rangle.\ \textbf{od}$$

Both corollaries are applicable here by virtue of the fact that x does not appear in S_2 and y does not appear in S_1. Recall that by assumption assignments and the *skip* statement are considered to be atomic regions.

Step 2. Decomposing fair total correctness

To prove (3) we use the fact that fair total correctness can be decomposed into fair termination and partial correctness. More precisely we use the following observation.

Lemma 8 For all nondeterministic or parallel programs R and all assertions p and q

$$\models_{fair} \{p\}\ R\ \{q\} \text{ iff } \models_{fair} \{p\}\ R\ \{\textbf{true}\} \text{ and } \models \{p\}\ R\ \{q\}.$$

Proof By the definition of fair total correctness and partial correctness. □

Thus to prove (3) it suffices to prove

$$\models_{fair} \{\exists u : f(u) = 0\}\ T\ \{\textbf{true}\} \tag{4}$$

and

$$\models \{\exists u : f(u) = 0\}\ T\ \{f(x) = 0 \vee f(y) = 0\}. \tag{5}$$

Step 3. Reduction to nondeterminism

To prove (4) we use the Parallelization Theorem. Consider the following nondeterministic program

$$
\begin{array}{l}
T' \equiv \mathit{found} := \textbf{false};\ x := 0;\ y := 1; \\
\quad \textbf{do}\ \neg \mathit{found} \rightarrow\ x := x + 1; \\
\qquad\qquad\qquad\ \textbf{if}\ f(x) = 0\ \textbf{then}\ \mathit{found} := \textbf{true}\ \textbf{fi} \\
\quad \Box\ \neg \mathit{found} \rightarrow\ y := y - 1; \\
\qquad\qquad\qquad\ \textbf{if}\ f(y) = 0\ \textbf{then}\ \mathit{found} := \textbf{true}\ \textbf{fi} \\
\quad \textbf{od}.
\end{array}
$$

Clearly

$$
\begin{array}{l}
\models_{tot}\ \{\textbf{true} \wedge \mathit{found}\} \\
\qquad\quad x := x + 1; \\
\qquad\quad \textbf{if}\ f(x) = 0\ \textbf{then}\ \mathit{found} := \textbf{true}\ \textbf{fi} \\
\qquad \{\textbf{true} \wedge \mathit{found}\}
\end{array}
$$

and

$$
\begin{array}{l}
\models_{tot}\ \{\textbf{true} \wedge \mathit{found}\} \\
\qquad\quad y := y - 1; \\
\qquad\quad \textbf{if}\ f(y) = 0\ \textbf{then}\ \mathit{found} := \textbf{true}\ \textbf{fi} \\
\qquad \{\textbf{true} \wedge \mathit{found}\}.
\end{array}
$$

Thus by the Parallelization Theorem, to prove (4) it suffices to prove

$$\models_{fair}\ \{\exists u : f(u) = 0\}\ T'\ \{\textbf{true}\}. \tag{6}$$

Step 4. Proving fair termination

To prove (6) we use a proof rule for fair total correctness of one level nondeterministic programs, introduced in Apt and Olderog [1983]. This rule is obtained from the Fairness Corollary 7 by absorbing, as it were, the scheduler parts $\mathit{INIT}, \mathit{SCH}_i$ and UPDATE_i referring to the scheduling variables $z_1, \ldots, z_n$ of the transformed program into the pre- and postconditions.

For the case of the identical loop guards this proof rule reads as follows:

FAIR LOOP RULE

$$
\begin{array}{ll}
(i) & \{p \wedge B\}\ S_i\ \{p\}, i \in \{1, \ldots, n\}, \\
(ii) & \{p \wedge B \wedge \bar{z} \geq 0 \wedge \exists z_i \geq 0 :\ t[z_j + 1/z_j]_{j \neq i} = \alpha\} \\
 & S_i \\
 & \{t < \alpha\}, i \in \{1, \ldots, n\}, \\
(iii) & p \wedge \bar{z} \geq 0 \rightarrow t \in W \\
\hline
 & \{p\}\ \textbf{do}\ \Box_{i=1}^{n}\ B \rightarrow S_i\ \textbf{od}\ \{p \wedge \neg B\}
\end{array}
$$

where

- t is an expression which takes values in a partial order $(P, <)$ that is well-founded on the subset $W \subseteq P$,
- $z_1, \ldots, z_n$ are integer variables that may occur freely in t, but not in p, B_i or S_i, for $i \in \{1, \ldots, n\}$,
- $t[z_j + 1/z_j]_{j \neq i}$ denotes the expression that results from t by substituting for every occurrence of z_j in t the expression $z_j + 1$; here j ranges over the set $\{1, \ldots, n\} - \{i\}$,
- $\bar{z} \geq 0$ abbreviates $z_1 \geq 0 \wedge \ldots \wedge z_n \geq 0$,
- α is a simple variable ranging over P and not occurring in p, t, B_i or S_i, for $i \in \{1, \ldots, n\}$; its purpose is to freeze the value of $t[z_j + 1/z_j]_{j \neq i}$ before the execution of S_i.

Note that with the precondition of premise (ii) simplified to

$$p \wedge B \wedge t = \alpha$$

and premise (iii) simplified to

$$p \rightarrow t \in W,$$

we obtain the usual rule for total correctness of nondeterministic **do**-loops. The above usage of the variables $z_1, \ldots, z_n$ in the premises allows us to establish fair total correctness.

We call p the *invariant* of the loop and t the *bound function* of the loop. In the proof outlines we denote them by **inv:** p and **bd:** t, respectively.

We use the above rule to first prove a weaker fair termination result than (6), viz. where f has a zero $u > 0$:

$$\models_{fair} \{f(u) = 0 \wedge u > 0\}\ T'\ \{\mathbf{true}\}. \tag{7}$$

A proof outline for (7) has the following structure:

$\{f(u) = 0 \wedge u > 0\}$
$found := \mathbf{false};$
$x := 0;$
$y := 1;$
$\{f(u) = 0 \wedge u > 0 \wedge \neg found \wedge x = 0 \wedge y = 1\}$
$\{\mathbf{inv} : p\}\{\mathbf{bd} : t\}$
$\mathbf{do}\ \neg found \rightarrow\ \{p \wedge \neg found\}$
$x := x + 1;$
$\mathbf{if}\ f(x) = 0\ \mathbf{then}\ found := \mathbf{true}\ \mathbf{fi}$

$\{p\}$
$\square\ \neg found \rightarrow\ \{p \wedge \neg found\}$
$y := y - 1$
if $f(y) = 0$ **then** $found := \mathbf{true}$ **fi**
$\{p\}$
od
$\{p \wedge found\}$
$\{\mathbf{true}\}$.

It remains to find a loop invariant p and a bound function t that will complete this outline.

Since the variable u is left unchanged by the program S, certainly

$$f(u) = 0 \wedge u > 0$$

is an invariant. But for the completion of the proof outline we need a stronger invariant relating u with the program variables x and $found$. We take as an overall invariant

$$p \equiv f(u) = 0 \wedge u > 0 \wedge x \leq u \wedge \mathbf{if}\ \neg found\ \mathbf{then}\ x < u\ \mathbf{fi}.$$

Notice that the implications

$$f(u) = 0 \wedge u > 0 \wedge \neg found \wedge x = 0 \wedge y = 1 \rightarrow p$$

and

$$p \wedge found \rightarrow \mathbf{true}$$

are obviously true and thus confirm the proof outline as given outside the **do**-loop.

To check the proof outline inside the loop, we take as partial order the set

$$P = Z \times Z,$$

ordered lexicographically by $<_{lex}$ and well-founded on the subset

$$W = N_0 \times N_0,$$

where Z denotes the set of integers and N_0 the set of natural numbers.

As a bound function we take

$$t \equiv < u - x, z_1 > .$$

In t the scheduling variable z_1 counts the number of executions of the second loop component before the next switch to the first one, and $u - x$, the distance

between the current test value x and the zero u, counts the remaining number of executions of the first loop component.

We show now that our choices of p and t complete the overall proof outline as given inside the **do**-loop. To this end, we have to prove the premises of the Fair Loop Rule.

We do this for the second premise. For the first loop component we have the proof outline:

$$
\begin{array}{l}
\{ \quad \neg found \wedge f(u) = 0 \wedge u > 0 \wedge x < u \\
\quad \wedge \ z_1 \geq 0 \wedge z_2 \geq 0 \wedge \exists z_1 \geq 0 : < u - x, z_1 >= \alpha\} \\
\{\exists z_1 \geq 0 : < u - x, z_1 >= \alpha\} \\
\{< u - x - 1, z_1 > <_{lex} \alpha\} \\
x := x + 1; \\
\{< u - x, z_1 > <_{lex} \alpha\} \\
found := f(x) = 0 \\
\{< u - x, z_1 > <_{lex} \alpha\} \\
\{t <_{lex} \alpha\}.
\end{array}
$$

Thus the bound function t drops below α because the program variable x is incremented into the direction of the zero u.

For the second loop component we have the proof outline:

$$
\begin{array}{l}
\{ \quad \neg found \wedge f(u) = 0 \wedge u > 0 \wedge x < u \\
\quad \wedge \ z_1 \geq 0 \wedge z_2 \geq 0 \wedge \ < u - x, z_1 + 1 >= \alpha\} \\
\{< u - x, z_1 + 1 >= \alpha\} \\
\{< u - x, z_1 > <_{lex} \alpha\} \\
y := y - 1; \\
found := f(y) = 0 \\
\{< u - x, z_1 > <_{lex} \alpha\} \\
\{t <_{lex} \alpha\}.
\end{array}
$$

Notice that only with the help of the scheduling variable z_1 we can prove that the bound function t drops here below α; the assignments to the program variables y and *found* do not affect t at all.

The remaining two premises can be easily established. This completes the proof of (7).

Symmetrically we can deal with the case when f has a zero $u \leq 0$:

$$\models_{fair} \{f(u) = 0 \wedge u \leq 0\}\ T'\ \{\mathbf{true}\}.$$

Combining this with (7) by standard rules of Hoare's logic yields (6).

Step 5. Proving partial correctness

It remains to prove (5). To this end, we use the approach of Owicki and Gries [1976] and Lamport [1977]. First we need to construct interference free proof outlines for partial correctness of the component programs T_1 and T_2 of T.

For T_1 we use the invariant

$$\begin{aligned} p_1 \equiv \quad & x \geq 0 && (8)\\ \wedge \ & (found \rightarrow (x > 0 \wedge f(x) = 0) \vee (y \leq 0 \wedge f(y) = 0)) && (9)\\ \wedge \ & (\neg found \wedge x > 0 \rightarrow f(x) \neq 0) && (10) \end{aligned}$$

to construct the proof outline

$$\begin{aligned} & \{\mathbf{inv} : p_1\}\\ & \mathbf{while}\ \neg found\ \mathbf{do}\\ & \quad \{x \geq 0 \ \wedge (found \rightarrow y \leq 0 \wedge f(y) = 0) \qquad (11)\\ & \qquad\qquad \wedge (x > 0 \rightarrow f(x) \neq 0)\}\\ & \quad \langle\ x := x + 1;\\ & \quad\ \ \mathbf{if}\ f(x) = 0\ \mathbf{then}\ found := \mathbf{true}\ \mathbf{fi}\rangle\\ & \mathbf{od}\\ & \{p_1 \wedge found\}. \end{aligned}$$

Similarly, for T_2 we use the invariant

$$\begin{aligned} p_2 \equiv \quad & y \leq 1 && (12)\\ \wedge \ & (found \rightarrow (x > 0 \wedge f(x) = 0) \vee (y \leq 0 \wedge f(y) = 0)) && (13)\\ \wedge \ & (\neg found \wedge y \leq 0 \rightarrow f(y) \neq 0) && (14) \end{aligned}$$

to construct the proof outline

$$\begin{aligned} & \{\mathbf{inv} : p_2\}\\ & \mathbf{while}\ \neg found\ \mathbf{do}\\ & \quad \{y \leq 1 \ \wedge (found \rightarrow x > 0 \wedge f(x) = 0)\\ & \qquad\qquad \wedge (y \leq 0 \rightarrow f(y) \neq 0)\}\\ & \quad \langle\ y := y - 1;\\ & \quad\ \ \mathbf{if}\ f(y) = 0\ \mathbf{then}\ found := \mathbf{true}\ \mathbf{fi}\rangle\\ & \mathbf{od}\\ & \{p_2 \wedge found\}. \end{aligned}$$

The intuition behind the invariants p_1 and p_2 is as follows. Conjuncts (8) and (12) state the range of values that the variables x and y may assume during the execution of the loops T_1 and T_2.

Thanks to the initialization of x with 0 and y with 1 in T, the condition $x > 0$ expresses the fact that the loop T_1 has been traversed as least once, and similarly the condition $y \leq 0$ expresses the fact that the loop T_2 has been traversed at least once. Thus the conjuncts (9) and (13) in the invariants p_1 and p_2 state that if the variable $found$ is true, then the loop T_1 has been traversed at least once and a zero x of f has been found, or that the loop T_2 has been traversed at least once and a zero y of f has been found.

The conjunct (10) in p_1 states that if the variable *found* is false and the loop T_1 has been traversed at least once, then x is not a zero of f. Analogously for the conjunct (14) in p_2.

Let us discuss now the proof outlines. In the first proof outline the most complicated assertion is (11). Note that

$$p_1 \wedge \neg found \rightarrow (11)$$

as required by the definition of a proof outline.

Given (11) as a precondition, the loop body in T_1 establishes p_1 as a post-condition, as required. Notice that the conjunct

$$found \rightarrow y \leq 0 \wedge f(y) = 0$$

in the precondition (11) is necessary to establish the conjunct (9) in the invariant p_1.

Next we deal with the interference freedom of the above proof outlines. In total 6 correctness formulas have to be proved, 3 for each component, pairwise symmetric.

The most difficult case is the interference freedom of the assertion (11) in the proof outline for T_1 with the loop body in T_2. It is proved by the following proof outline:

$\{ \quad x \geq 0 \wedge (found \rightarrow y \leq 0 \wedge f(y) = 0) \wedge (x > 0 \rightarrow f(x) \neq 0)$
$\quad \wedge \; y \leq 1 \wedge (found \rightarrow x > 0 \wedge f(x) = 0) \wedge (y \leq 0 \rightarrow f(y) \neq 0)\}$
$\{x \geq 0 \wedge y \leq 1 \wedge \neg found \wedge (x > 0 \rightarrow f(x) \neq 0)\}$
$\langle \; y := y - 1;$
$\quad \textbf{if } f(y) = 0 \textbf{ then } found := \textbf{true fi}\rangle$
$\{x \geq 0 \wedge (found \rightarrow y \leq 0 \wedge f(y) = 0) \wedge (x > 0 \rightarrow f(x) \neq 0)\}.$

Note that the first assertion in the above proof outline indeed implies $\neg found$:

$$(found \rightarrow (x > 0 \wedge f(x) = 0)) \wedge (x > 0 \rightarrow f(x) \neq 0)$$

implies

$$found \rightarrow (f(x) \neq 0 \wedge f(x) = 0)$$

implies

$$\neg found.$$

This information is recorded in the second assertion of the proof outline and used to establish the last assertion.

The remaining cases in the interference freedom proof are straightforward and left to the reader.

We now apply the rule of parallel composition and get

$$\{p_1 \wedge p_2\}\ [T_1 \| T_2]\ \{p_1 \wedge p_2 \wedge found\}.$$

From this correctness formula it is straightforward to prove the desired partial correctness result (5).

This concludes the proof of (2).

Discussion

(i) In the above proof we first simplified (2) to (3) and then decomposed (3) into (4) and (5). Clearly we could have decomposed in an analogous way (2). But this would lead to a much more complicated proof of partial correctness because S contains more interference points than T. In particular, to deal with the initialization $x := 0$ and $y := 1$ within the parallel composition in S requires the use of auxiliary variables.

This shows that the Atomicity and Initialization Theorems simplify the task of proving parallel programs correct.

(ii) To prove (4) we used the Parallelization Theorem. It is useful to note that we cannot use it to prove (3) directly. Indeed, to apply it we would have to prove

$$\begin{array}{ll} \models_{tot} & \{(f(x) = 0 \vee f(y) = 0) \wedge found\} \\ & x := x + 1; \\ & \textbf{if } f(x) = 0 \textbf{ then } found := \textbf{true fi} \\ & \{(f(x) = 0 \vee f(y) = 0) \wedge found\} \end{array}$$

and a similar claim for the second component. However, the above claim does not hold as the assignment $x := x + 1$ can invalidate the assertion $f(x) = 0$.

This shows that the Parallelization Theorem is of limited applicability and has to be used in conjunction with other methods.

(iii) To prove fair termination of T' in (6) or (7) we could have applied the Fairness Corollary 7 to T' and proved ordinary termination of the transformed version of T'. However, we preferred to use the Fair Loop Rule presented in Step 3 because it allowed us to reason directly about the original program T'. In this way certain parts of the transformation are handled uniformly and a generation of several intermediate assertions (for example dealing with random assignments) is avoided.

Appendix

In this appendix we prove Theorems 1 and 2. As a preparation we define rigorously the program semantics. We use here the operational approach due to Hennessy and Plotkin [1979]. Its basic concept is a *configuration* which is simply a pair $< S, \sigma >$ consisting of a program S and a proper state σ. The semantics is then defined in terms of *transitions*. Intuitively, a transition

$$< S, \sigma > \rightarrow < R, \tau >$$

means: executing S one step in a proper state σ can lead to state τ with R being the remainder of S still to be executed. To express termination we allow the *empty program* E inside configurations: $R \equiv E$ in in the above transition means that S terminates in τ. We stipulate that $E;\ S$ and $S;\ E$ abbreviate to S. Also, we identify

$$[E \| \ldots \| E] \equiv E.$$

This expresses the fact that a parallel program terminates iff all its components terminate.

In the following σ, τ stand for proper states, i.e. mappings from variables to values. We write $\sigma(t)$ to denote the value of an expression t in σ and $\sigma \models B$ to express that the Boolean expression B evaluates to true in σ. Further on, $\sigma[\sigma(t)/u]$ is a proper state that agrees with σ except for the variable u where its value is $\sigma(t)$. The transition relation $\rightarrow$ is defined by induction on the structure of programs. We use the following transition *axioms* and *rules*:

(i) $< skip, \sigma > \rightarrow < E, \sigma >$,

(ii) $< u := t, \sigma > \rightarrow < E, \sigma[\sigma(t)/u] >$,

(iii)

$$\frac{< S_1, \sigma > \rightarrow < S_2, \tau >}{< S_1;\ S, \sigma > \rightarrow < S_2;\ S, \tau >}$$

(iv) $<$ **if** B **then** S_1 **else** S_2 **fi**, $\sigma > \rightarrow < S_1, \sigma >$ where $\sigma \models B$,

(v) $<$ **if** B **then** S_1 **else** S_2 **fi**, $\sigma > \rightarrow < S_2, \sigma >$ where $\sigma \models \neg B$,

(vi) $<$ **do** $\Box_{i=1}^{n}\ B_i \rightarrow S_i$ **od** , $\sigma > \rightarrow < S_i;$ **do** $\Box_{i=1}^{n}\ B_i \rightarrow S_i$ **od** , $\sigma >$
where $\sigma \models B_i$ and $i \in \{1, \ldots, n\}$,

(vii) $<$ **do** $\Box_{i=1}^{n}\ B_i \rightarrow S_i$ **od** , $\sigma > \rightarrow < E, \sigma >$ where $\sigma \models \wedge_{i=1}^{n}\ \neg B_i$.

(viii)

$$\frac{< S, \sigma > \rightarrow^* < E, \tau >}{< (S) > \rightarrow < E, \tau >}$$

(ix)

$$\frac{< S_i, \sigma > \rightarrow < T_i, \tau >}{< [S_1 || \ldots || S_i || \ldots || S_n], \sigma > \rightarrow < [S_1 || \ldots || T_i || \ldots || S_n], \tau >}$$

where $i \in \{1, \ldots, n\}$.

By definition the transitions for **while** B **do** S **od** are as for **do** $B \rightarrow S$ **od**. Rule (viii) formalizes the intuitive meaning of atomic regions by reducing each terminating computation of the "body" S of an atomic region $\langle S \rangle$ to a one step computation of the atomic region. Rule (ix) states that a parallel program $[S_1 || \ldots || S_n]$ performs a transition if one of its component performs a transition. Thus concurrency is modelled here by interleaving.

A transition $< S, \sigma > \rightarrow < R, \tau >$ is possible if and only if it can be deduced in the above transition system.

Definition 9 Let S be a parallel or nondeterministic program and σ a proper state.

(i) A *transition sequence of S starting in σ* is a finite or infinite sequence of configurations $< S_i, \sigma_i >$ $(i \geq 0)$ such that

$$< S, \sigma > = < S_0, \sigma_0 > \rightarrow < S_1, \sigma_1 > \rightarrow \ldots \rightarrow < S_i, \sigma_i > \rightarrow \ldots$$

(ii) A *computation of S starting in σ* is a transition sequence of S starting in σ which cannot be extended.

(iii) A computation of S *is terminating in τ* (or *terminates in τ*) if it is finite and its last configuration is of the form $< E, \tau >$.

(iv) A computation of S *is diverging* (or *diverges*) if it is infinite. S can *diverge from σ* if there exists an infinite computation of S starting in σ.

□

Let $\rightarrow^*$ stand for the transitive, reflexive closure of $\rightarrow$. We now define three semantics of parallel or nondeterministic programs by putting for a proper state σ

$$\mathcal{M}[\![S]\!](\sigma) = \{\tau \mid < S, \sigma > \rightarrow^* < E, \tau >\},$$

$$\mathcal{M}_{tot}[\![S]\!](\sigma) = \mathcal{M}[\![S]\!](\sigma) \cup \{\perp \mid S \text{ can diverge from } \sigma\},$$

$$\mathcal{M}_{fair}[\![S]\!](\sigma) = \mathcal{M}[\![S]\!](\sigma) \cup \{\perp \mid S \text{ can diverge from } \sigma \text{ by a fair computation.}\}$$

The corresponding notions of partial, total and fair total correctness of programs can be defined as inclusion properties of sets of states. For partial correctness we put

$$\models \{p\}\ S\ \{q\} \text{ iff } \mathcal{M}[\![S]\!]([\![p]\!]) \subseteq [\![q]\!]$$

where $[\![p]\!]$ is the set of all proper states satisfying the assertion p and analogously for q. The definitions for $\models_{tot}$ and $\models_{fair}$ refer to $\mathcal{M}_{tot}$ and $\mathcal{M}_{fair}$ instead.

Proof of Theorem 1. We proceed in 6 steps.

Step 1 We consider the case when S and T have no initialization part S_0 and introduce a subset of computations of T. To this end, observe that in an arbitrary finite or infinite transition sequence

$$\xi :< T, \sigma >=< T_1, \sigma_1 > \rightarrow \ldots \rightarrow < T_j, \sigma_j > \rightarrow \ldots$$

of T, each transition $< T_j, \sigma_j > \rightarrow < T_{j+1}, \sigma_{j+1} >$ in ξ is of one of the following three types.

It can be a *B_i-transition* passing succcessfully the loop condition B in the i-th component so that

$$\begin{aligned} T_j &\equiv [\ldots \| \textbf{while } B \textbf{ do } \langle S_i \rangle \textbf{ od} \| \ldots] \text{ and } \sigma_j \models B, \\ T_{j+1} &\equiv [\ldots \| \langle S_i \rangle;\ \textbf{while } B \textbf{ do } \langle S_i \rangle \textbf{ od} \| \ldots] \text{ and } \sigma_{j+1} = \sigma_j; \end{aligned}$$

it can be an *S_i-transition* executing the loop body S_i as an atomic action so that

$$\begin{aligned} T_j &\equiv [\ldots \| \langle S_i \rangle;\ \textbf{while } B \textbf{ do } \langle S_i \rangle \textbf{ od} \| \ldots], \\ T_{j+1} &\equiv [\ldots \| \textbf{while } B \textbf{ do } \langle S_i \rangle \textbf{ od} \| \ldots]; \end{aligned}$$

or it can be an *E_i-transition* terminating the loop of the i-th component so that

$$\begin{aligned} T_j &\equiv [\ldots \| \textbf{while } B \textbf{ do } \langle S_i \rangle \textbf{ od} \| \ldots] \text{ and } \sigma_j \models \neg B, \\ T_{j+1} &\equiv [\ldots \| E \| \ldots] \text{ and } \sigma_{j+1} = \sigma_j. \end{aligned}$$

We say that ξ is *delay free* if each B_i-transition is immediately followed by the corresponding S_i-transition.

Note that in a delay free computation of T for each S_i-transition $< T_j, \sigma_j > \rightarrow < T_{j+1}, \sigma_{j+1} >$

$$\begin{aligned} T_j \equiv [&\textbf{while } B \textbf{ do } \langle S_1 \rangle \textbf{ od} \| \ldots \\ &\| \langle S_i \rangle;\ \textbf{while } B \textbf{ do } \langle S_i \rangle \textbf{ od} \| \ldots \\ &\| \textbf{while } B \textbf{ do } \langle S_n \rangle \textbf{ od} \| \ldots] \end{aligned}$$

and

$$\begin{aligned} T_{j+1} \equiv\ & [\textbf{while } B \textbf{ do } \langle S_1 \rangle \textbf{ od}\| \ldots \\ & \|\textbf{while } B \textbf{ do } \langle S_i \rangle \textbf{ od}\| \ldots \\ & \|\textbf{while } B \textbf{ do } \langle S_n \rangle \textbf{ od}\| \ldots] \\ \equiv\ & T. \end{aligned}$$

Also, after an E_i-transition only E_j-transitions for $i \neq j$ can take place.

Step 2 To compare the computations of S and T, we use the following notion of equivalence. Two computations are called *i/o equivalent* if they start in the same state and either both diverge or both terminate in the same state.

Step 3 We prove the following two claims:

- every (fair) computation of S is i/o equivalent to a delay free (fair) computation of T,
- every delay free (fair) computation of T is i/o equivalent to a (fair) computation of S.

First consider a (fair) computation ξ of S. We construct an i/o equivalent delay free (fair) computation of T from ξ by replacing

- every loop entry transition

$$< S, \sigma > \rightarrow < S_i;\ S, \sigma >$$

  with the B_i-transition

$$\begin{aligned} < T, \sigma > \rightarrow < & [\textbf{while } B \textbf{ do } \langle S_1 \rangle \textbf{ od}\| \ldots \\ & \|\langle S_i \rangle;\ \textbf{while } B \textbf{ do } S_i \textbf{ od}\| \ldots \\ & \|\textbf{while } B \textbf{ do } S_n \textbf{ od}\| \ldots, \sigma >, \end{aligned}$$

- every transition subsequence

$$< S_i;\ S, \sigma > \rightarrow \ldots \rightarrow < S, \tau >,$$

  forming the stepwise execution of the loop body S_i, with the S_i-transition

$$\begin{aligned} < & [\textbf{while } B \textbf{ do } \langle S_1 \rangle \textbf{ od}\| \ldots \\ & \|\langle S_i \rangle;\ \textbf{while } B \textbf{ do } S_i \textbf{ od}\| \ldots \\ & \|\textbf{while } B \textbf{ do } S_n \textbf{ od}\| \ldots, \sigma > \rightarrow < T, \tau >, \end{aligned}$$

- every loop exit transition

$$< S, \sigma > \rightarrow < E, \sigma >$$

with a sequence of n final E_i-transitions, $i \in \{1, \ldots, n\}$, dealing with the state σ.

Now consider a delay free (fair) computation η of T. By applying the above replacement operations in reverse direction, we construct an i/o equivalent (fair) computation of S from η.

Step 4 To compare computations of T, we introduce the following variant of i/o equivalence. Two computations are called *q-equivalent* if they both start in the same state and either both diverge or both terminate in a state satisfying assertion q.

Step 5 By a *p-computation* we mean a computation starting in a state satisfying the assertion p. Suppose that every terminating delay free p-computation of T terminates in a state satisfying the assertion q. We prove that under this assumption every (fair) p-computation of T is q-equivalent to a delay free p-computation of T.

Consider a (fair) computation

$$\xi :< T, \sigma >=< T_1, \sigma_1 > \rightarrow \ldots \rightarrow < T_j, \sigma_j > \rightarrow \ldots$$

of T with $\sigma \models p$.

Case 1 $\forall j \geq 1 : \sigma_j \models B$.

Then ξ is infinite. Let

$$S_{i_1}, S_{i_2}, S_{i_3}, \ldots$$

be the sequence of all S_i-transitions in ξ. Then there exists an infinite delay free (fair) p-computation η of T which starts in σ and has the same sequence of S_i-transitions. We can construct η by performing the corresponding B_i-transitions immediately before the S_i-transitions of this sequence. This is possible because in the present case the B_i-transitions are everywhere enabled.

Case 2 $\exists j \geq 1 : \sigma_j \models \neg B$.

Let j_0 be the smallest such j. Consider the prefix

$$\xi_0 :< T, \sigma >=< T_1, \sigma_1 > \rightarrow \ldots \rightarrow < T_{j_0}, \sigma_{j_0} >$$

of ξ. By the choice of j_0, the last transition in ξ_0 is an S_i-transition.

We first show that $\sigma_{j_0} \models q$. To this end, we argue in a similar way as above. Let

$$S_{i_1}, \ldots, S_{i_m}$$

be the sequence of all S_i-transitions in ξ_0. Then there exists a finite delay free transition sequence η_0 of T starting in σ, running through the same S_i-transitions as ξ_0, and ending in the configuration $< T, \sigma_{j_0} >$. Note that we indeed obtain here the program T thanks to the observation about S_i-transitions in delay free transition sequences stated in Step 1. Since $\sigma_{j_0} \models \neg B$, the only transitions which are possible after $< T, \sigma_{j_0} >$ are E_i-transitions, $i \in \{1, \ldots, n\}$. By adding all these transitions, we obtain a delay free p-computation η of T terminating in σ_{j_0}. By the assumption of this step, $\sigma_{j_0} \models q$.

Thus $\sigma_{j_0} \models q \wedge \neg B$. This information is sufficient to see how the original computation ξ of T continues after the prefix ξ_0. In ξ_0 there may be some B_i-transitions without a corresponding S_i-transition. Since by assumption

$$\models \{q \wedge \neg B\}\ S_i\ \{q \wedge \neg B\},$$

these remaining S_i-transitions all yield states satisfying $q \wedge \neg B$. Thus these S_i-transitions and n final E_i-transitions are the only possible transitions in the remainder of ξ. Thus also ξ terminates in a state satisfying q. Consequently, ξ and the delay free computation η are q-equivalent.

Step 6 By combining the results from Step 3 and 5, it is easy to prove the claim of the theorem for the case when S and T have no initialization part S_0. The first claim of Step 3 implies the "if"-part. The second claim of Step 3 together with the result of Step 5 imply the "only-if"-part. Indeed, suppose

$$\models \{p\}\ S\ \{q\},$$

i.e. every terminating p-computation of S terminates in a state satisfying q. Then by the second claim of Step 3, every terminating delay free p-computation of T terminates in a state satisfying q. Thus by the result of Step 5, every terminating p-computation of T terminates in a state satisfying q, i.e.

$$\models \{p\}\ T\ \{q\}.$$

Similar arguments deal with $\models_{tot}$ and $\models_{fair}$. The case when S and T have an initialization part S_0 is left to the reader. □

Proof of Theorem 2. We treat the case when S has no initialization part S_0 and T results form S by splitting $\langle R_1;\ R_2 \rangle$ into $\langle R_1 \rangle$; $\langle R_2 \rangle$. Our presentation follows the 6 steps outlined in the previous proof.

Step 1 By an R_k-transition, $k \in \{1, 2\}$, we mean a transition occurring in a computation of T which is of the form

$$< [U_1 \| \ldots \| \langle R_k \rangle;\ U_i \| \ldots \| U_n], \sigma > \ \rightarrow \ < [U_1 \| \ldots \| U_i \| \ldots \| U_n], \tau > .$$

We call a fragment ξ of a computation of T *good* if in ξ each R_1-transition is immediately followed by the corresponding R_2-transition, and we call ξ *almost good* if in ξ each R_1-transition is eventually followed by the corresponding R_2-transition.

Observe that every fair and hence every finite computation of T is almost good.

Step 2 To compare the computations of S and T, we use the i/o equivalence introduced in Step 2 of the proof of Theorem 1.

Step 3 We prove the following two claims:

- every (fair) computation of S is i/o equivalent to a good (fair) computation of T,
- every good (fair) computation of T is i/o equivalent to a (fair) computation of S.

First consider a (fair) computation ξ of S. Every program occurring in a configuration of ξ is a parallel composition of n components. Let for such a program U the program $split(U)$ result from U by replacing in the i-th component of U every occurrence of $\langle R_1;\ R_2 \rangle$ by $\langle R_1 \rangle;\ \langle R_2 \rangle$. For example, $split(S) \equiv T$.

We construct an i/o equivalent good (fair) computation of T from ξ by replacing

- every transition of the form

$$\begin{array}{l} < [U_1 \| \ldots \| \langle R_1;\ R_2 \rangle;\ U_i \| \ldots \| U_n], \sigma > \\ \rightarrow \ < [U_1 \| \ldots \| U_i \| \ldots \| U_n], \tau > \end{array}$$

  with two consecutive transitions

$$\begin{array}{l} < split([U_1 \| \ldots \| \langle R_1;\ R_2 \rangle;\ U_i \| \ldots \| U_n]), \sigma > \\ \rightarrow \ < split([U_1 \| \ldots \| \langle R_2 \rangle;\ U_i \| \ldots \| U_n]), \sigma_1 > \\ \rightarrow \ < split([U_1 \| \ldots \| U_i \| \ldots \| U_n]), \tau > \end{array}$$

  where the intermediate state σ_1 is defined by

$$< \langle R_1 \rangle, \sigma > \ \rightarrow \ < E, \sigma_1 >,$$

- every other transition

$$< U, \sigma > \rightarrow < V, \tau >$$

with

$$< split(U), \sigma > \rightarrow < split(V), \tau > .$$

Now consider a good (fair) computation η of T. By applying the above replacement operations in reverse direction we construct an i/o equivalent (fair) computation of S from η.

Step 4 For the comparison of computations of T we use i/o equivalence, but to reason about it we also introduce a more discriminating variant of it called "permutation equivalence".

First consider an arbitrary computation ξ of T. Every program occurring in a configuration of ξ is the parallel composition of n components. To distinguish between different kinds of transitions in ξ, we attach labels to the transition arrow $\rightarrow$. We write

$$< U, \sigma > \stackrel{R_k}{\rightarrow} < V, \tau >$$

if $k \in \{1, 2\}$ and $< U, \sigma > \rightarrow < V, \tau >$ is an R_k-transition of the i-th component of U,

$$< U, \sigma > \stackrel{i}{\rightarrow} < V, \tau >$$

if $< U, \sigma > \rightarrow < V, \tau >$ is any other transition caused by the activation of the i-th component of U, and

$$< U, \sigma > \stackrel{j}{\rightarrow} < V, \tau >$$

if $j \neq i$ and $< U, \sigma > \rightarrow < V, \tau >$ is a transition caused by the activation of the j-th component of U.

Hence with each transition arrows in a computation of T there is a unique label associated. This enables us to define:

Two computations η and ξ of T are *permutation equivalent* if

- η and ξ start in the same state,
- for all states σ, η terminates in σ iff ξ terminates in σ,
- the possibly infinite sequence of labels attached to the transition arrows in η and ξ are permutations of each other.

Clearly, permutation equivalence of computations of T implies their i/o equivalence.

Step 5 We prove the following claim: every (fair) computation of T is i/o equivalent to a good (fair) computation of T.

To this end, we establish two simpler claims.

Claim 1 Every (fair) computation of T is i/o equivalent to an almost good (fair) computation of T.

Proof of Claim 1. Consider a computation ξ of T which is not almost good. Then by the observation stated in Step 1, ξ is not fair and hence diverging. More precisely, there exists a suffix ξ_1 of ξ which starts in a configuration $< U, \sigma >$ with an R_1-transition and then continues with infinitely many transitions not involving the i-th component any more, say

$$\xi_1 :< U, \sigma > \stackrel{R_1}{\rightarrow} < U_0, \sigma_0 > \stackrel{j_1}{\rightarrow} < U_1, \sigma_1 > \stackrel{j_2}{\rightarrow} \ldots$$

where $j_k \neq i$ for $k \geq 1$. By the definition of semantics of **while**-programs we conclude the following: if R_1 is disjoint from S_j with $j \neq i$, then there is also an infinite transition sequence of the form

$$\xi_2 :< U, \sigma > \stackrel{j_1}{\rightarrow} < V_1, \tau_1 > \stackrel{j_2}{\rightarrow} \ldots,$$

and if R_2 is disjoint from S_j with $j \neq i$, then there is also an infinite transition sequence of the form

$$\xi_3 :< U, \sigma > \stackrel{R_1}{\rightarrow} < U_0, \sigma_0 > \stackrel{R_2}{\rightarrow} < V_0, \tau_0 > \stackrel{j_1}{\rightarrow} < V_1, \tau_1 > \stackrel{j_2}{\rightarrow} \ldots$$

We say that ξ_2 is obtained from ξ_1 by *deletion* of the initial R_1-transition and ξ_3 is obtained from ξ_1 by *insertion* of an R_2-transition. Replacing the suffix ξ_1 of ξ by ξ_2 or ξ_3 yields an almost good computation of T which is i/o equivalent to ξ. □

Claim 2 Every almost good (fair) computation of T is permutation equivalent to a good (fair) computation of T.

Proof of Claim 2. By the definition of semantics of **while**-programs the following: if R_k with $k \in \{1, 2\}$ is disjoint from S_j with $j \neq i$, then the relations $\stackrel{R_k}{\rightarrow}$ and $\stackrel{j}{\rightarrow}$ *commute*, i.e.

$$\stackrel{R_k}{\rightarrow} \circ \stackrel{j}{\rightarrow} = \stackrel{j}{\rightarrow} \circ \stackrel{R_k}{\rightarrow}$$

where $\circ$ denotes relational composition. Repeated application of this commutativity allows us to permute the transitions of every almost good fragment ξ_1 of a computation of T of the form

$$\xi_1 :< U, \sigma > \xrightarrow{R_1} \circ \xrightarrow{j_1} \circ \ldots \circ \xrightarrow{j_m} \circ \xrightarrow{R_2} < V, \tau >$$

with $j_k \neq i$ for $k \in \{1, \ldots, m\}$ into a good order, i.e. into

$$\xi_2 :< U, \sigma > \xrightarrow{j_1} \circ \ldots \circ \xrightarrow{j_m} \circ \xrightarrow{R_1} \circ \xrightarrow{R_2} < V, \tau >$$

or

$$\xi_3 :< U, \sigma > \xrightarrow{R_1} \circ \xrightarrow{R_2} \circ \xrightarrow{j_1} \circ \ldots \circ \xrightarrow{j_m} < V, \tau >$$

depending on whether R_1 or R_2 is disjoint from S_j with $j \neq i$.

Consider now an almost good computation ξ of T. We construct from ξ a permutation equivalent good computation ξ^* of T by successively replacing every almost good fragment of ξ of the form ξ_1 by a good fragment of the form ξ_2 or ξ_3.

Note that a computation η of T is fair iff there exists a configuration $< U, \sigma >$ such that every sequential component of U has either terminated or is activated infinitely often in the suffix of η starting in $< U, \sigma >$. Since this property is preserved by the above construction of a permutation equivalent computation ξ^* from ξ, we conclude: if ξ is fair, also ξ^* is fair. □

Claims 1 and 2 together imply the claim of Step 5.

Step 6 By combining the results of Step 3 and 5, we get the claim of the theorem for the case when S has no initialization part S_0 and T results from S by splitting $\langle R_1;\ R_2 \rangle$ into $\langle R_1 \rangle$; $\langle R_2 \rangle$. The cases when S has an initialization part S_0 and where T results from S by splitting the atomic region $\langle$**if** B **then** R_1 **else** R_2 **fi**$\rangle$ are left to the reader. □

The proof of the Initialization Theorem follows the same lines as the proof of the Atomicity Theorem and is therefore omitted.

References

[1] E. Ashcroft and Z. Manna, Formalization of properties of parallel programs, *Machine Intelligence 6*, pp. 17–41, 1971.

[2] K.R. Apt and E.-R. Olderog, Proof rules and transformations dealing with fairness, *Science of Computer Programming 3*, pp. 65–100, 1983.

[3] R.J.R. Back, A method for refining atomicity in parallel algorithms, *Lecture Notes in Computer Science 366*, Springer-Verlag, 1989.

[4] M. Chandy and J. Misra, *A Foundation of Parallel Program Design*, Addison-Wesley, 1988.

[5] E. W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, *Communications of the ACM 18*, pp. 453–457, 1975.

[6] L. Flon and N. Suzuki, The total correctness of parallel programs, *SIAM Journal of Computing*, pp. 227-246, 1978.

[7] N. Francez, *Fairness*, Springer-Verlag, 1986.

[8] M.C.B.Hennessy and G.D. Plotkin, Full abstraction for a simple programming language, *Lecture Notes in Computer Science 74*, Springer-Verlag, 1979.

[9] L. Lamport, Proving the correctness of multiprocess programs, *IEEE Transactions on Software Engineering SE-3:2*, pp.125–143, 1977.

[10] R. Lipton, Reduction: a method of proving properties of parallel programs, *Communications of the ACM 18*, pp. 717–721, 1975.

[11] E. R. Olderog and K. R. Apt, Fairness in parallel programs, the transformational approach, *ACM TOPLAS 10*, pp. 420–455, 1988.

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