PROOF RULES FOR DYNAMIC PROCESS CREATION AND DESTRUCTION

by

Limor Fix and Nissim Francez

Technical Report #547
September 1990
PROOF RULES for DYNAMIC PROCESS CREATION and DESTRUCTION

by

Limor Pli and Nissim Francez

Computer Science Department, The Technion

Haifa 32000, Israel

E-address: francez@techsel (BITNET), francez@sel (CSNET)

Abstract: The paper presents proof-rules for partial-correctness of dynamic concurrency, by which processes are dynamically created and destroyed during the execution of a concurrent program. Both message-passing concurrency and shared-variables concurrency are considered. For the latter, the paper presents also a proof method for deadlock freedom. The proofs-rules are canonical and present several proof-theoretical issues absent in previous static proof-systems. In particular, both interference-freedom and cooperation are needed in both models. Several examples of applications of the rule are presented.
1. INTRODUCTION

The purpose of this paper is to present proof rules for proving partial-correctness and deadlock-freedom of concurrent programs with dynamic process creation and destruction. Both shared-variables concurrency and message-passing concurrency are considered. The dynamic structure of concurrency is achieved by explicit instructions for the creation and destruction of processes and thereby it transcends the power of recursion over concurrency, e.g. as in CCS [Mi80] or TCSP [Ho85]. Such a dynamic concurrency is found more and more frequently in new programming languages like Ada [ADA83], NIL [SY83], POOL [Am85], and its rigorous understanding is a prerequisite for its useful utilization.

The only attempt in this direction we are aware of is [dB86], which deals only with process creation (no destruction), only with partial correctness (no deadlock-freedom) and only for message-passing concurrency (no shared variables). Although our study began independently, that paper strongly influenced our work. An informal study of some semantic problems in the context of process creation and destruction, which goes beyond partial-correctness properties, is found in [FS86]. A formal study of the semantics of process-creation (without process destruction) with no accompanying proof-theory appears in [AB88].

Our study is carried out in the context of two-staged proof systems for concurrency, where one has local proofs for each process in the first stage, and a global (usually pairwise) consistency relation among these proofs in the second stage: noninterference in the case of shared variables [OG76] and cooperation [AFR80], or satisfaction [LG81], for message passing. This dichotomy seems to break down in the context of dynamic concurrency, as suggested by this paper. We assume that the reader is familiar with two-leveled, assertional proof systems for partial correctness for both models considered here (basically, with [OG76] and [AFR80]).

There are two major design goals in designing a proof system for partial correctness, beyond the (by now well known) soundness and semantic (or relative) completeness. While the latter are meaningful requirements for any proof system for program correctness, the new goals only apply to dynamic concurrency. The systems suggested in this paper meet both additional goals.

1) Canonical proofs

At the first stage, a separate proof (-outline) for each process is constructed. However, in the dynamic case, the actual number of processes and their exact identities are only determined during execution, and depend also on the initial state. It is still required to have only a fixed number of such local proofs. Hence, one needs a system where only the process types are individually verified (their number is fixed and statically determined), implying the local correctness of all their surviving (i.e. created and not destroyed thereafter) dynamic instances. Still, each instance may behave differently and interact differently with its environment. Such
2) Self interaction

The problem of self interaction arises at the second stage, in which local proofs are interrelated. In the case of interference-freedom proofs, the standard definition in [OG76] considers only interference of proofs from separate processes. By the same token as above we have to consider here the interference of instances with each other. Here we encounter a new problem: two instances of the same process-type (with the same program text) may interfere with each other! Furthermore, in principle a statement in one instance (of some process-type) may interfere with the same statement in another instance of the same type. As these statements have the same precondition associated with them, their conjunction will never yield a contradiction, as required in case this interference does not happen because the two identical statements are so synchronized as to never actually be executed in parallel. By means of an appropriate usage of indexed variables, this problem is also solved. As for the analogous problem occurring in message-passing concurrency, again two instances of the same process-type may communicate, and a cooperation proof of a process-type with itself is needed. However, here the situation is somewhat easier, as a global invariant is available to contradict the (identical) preconditions. Again, this is achieved by keeping track in this invariant about possible relationships among these indices.

The dynamic nature of the programs causes also a certain inadequacy of the method for proving deadlock-freedom proposed in [OG76] for shared-variables concurrency. This method is based on the notion of a situation, which is a representation of the local program-counters of the processes by assertions which hold in those locations. According to this method, given a partial-correctness proof-outline, the assertions corresponding to every blocked-situation (a candidate for deadlock) are shown to be inconsistent. Thus, no such situation can arise during execution, and hence no deadlock is possible. Trying to apply this method to shared-variables dynamic concurrency raises a problem. It is impossible to check all the blocked-situations of the program since the number of instances of each process-type is not bounded. As a result, there is an unbounded number of possible blocked-situations. The extension of the method for proving deadlock-freedom presented here overcomes this problem by grouping all possible blocked-situations into a bounded number of groups, again using a kind of canonicity. Another difference
from [OG76] in this respect is the insufficiency of characterizing blocked-situations only by representative assertions, one for each process-type. What is needed is yet another exclusion assertion, implying that no active process has its control at a point not represented in the blocked-situation at hand. This information was present "free" in the [OG76] static system.

While for static concurrency there exist proof systems employing either interference-freedom (for shared variables), or cooperation (for message passing), it turns out that both are needed in the dynamic case. This is because of the structure of the precondition and the postcondition of a concurrent composition. In the static case (in both models) the precondition of a concurrent composition is the conjunction of the local preconditions of all the processes. In the dynamic case, we take the precondition of the only preexisting process as the precondition of the concurrent composition. Thus, a special provision is needed for establishing preconditions upon creation of an instance. Formally, this turns out to appear as a special cooperation test. The new form of the postcondition of a concurrent composition is a conjunction over all the local postconditions of the surviving processes, in contrast to a conjunction over postcondition of all processes where only process creation is considered [dB86].

For our proof systems we need more logical variables to "surface" from the semantics, on top of the incarnation counters used in [dB86]. Actually, such counters were used already in [DB85] (and see there for earlier references) in a canonical proof-system for DP [Bh78], a credit missing in [dB86]. These variables denote the current collection of active (i.e. created and not destroyed) processes. In some sense, these are shared variables, in principle enforcing noninterference tests, even for the message passing concurrency. As it happens, these become simple due to the presence of a global invariant.

The rest of the paper is organized as follows: Section 2 contains the description of the two programming languages used, together with a formal operational semantics. Section 3 discusses the common parts of the two proof-systems, mainly the axioms for create and cancel, and their rationale. Section 4 contains the proof-system of the message-passing language with its extended cooperation test due to process destruction and presents examples. Section 5 contains the proof-system for the shared-variables concurrency with the extended interference freedom rule and the additional cooperation rule. It also presents the (new) extension towards proofs of deadlock-freedom, with examples. Section 6 ends with conclusions, relation to other work (mainly [dB86]), a consideration of the soundness and completeness of the proposed proof-system, and future work.

2. THE PROGRAMMING LANGUAGES

In this section, we describe the programming languages in terms of which our proof-systems are formulated. We try to present the whole discussion as uniformly as possible, and in particular, we discuss only once features
common to both languages.

We consider two simple programming languages, one similar to that of [OG76], augmented with primitives for process creation and destruction and guarded commands [Di75]. The second is a dynamic version of CSP [Ho78] with synchronous message-passing communication, similar to the one considered in [dB86] but augmented with a cancel primitive. These two languages were chosen since they represent two models, the message-passing model and the shared-memory one. Moreover, they both have well known proof-systems for the static case, so it is possible to focus on the dynamic aspects.

A typical program in both languages has the form $P:: (T_1 = S_1, \ldots, T_n = S_n)$, having some statically determined finite number $n$ of process-types $T_i$, $1 \leq i \leq n$. Each process-type $T_i$ has a corresponding body $S_i$, which is a program in the appropriate language. Instances of a process-type $T_i$, $1 \leq i \leq n$, are created by a command of the form $x := \text{create}(T_i)$, assigning to the variable $x$ a pair $<i, i>$ which is a unique process identifier: $x[i]$ (i.e. $i$) is the process-type number and $x[i+2]$ (i.e. $i$) is the incarnation number, called also the instance number. Initially, one instance of $T_1$ exists by convention, and no other instances of type $T_i$, $1 \leq i \leq n$ exist. All other processes to be executed in the program are to be explicitly created by executing create commands. A process, whose identifier is in a variable $x$, may be destroyed by executing a command cancel $(x)$. When a process is cancelled its execution stops "immediately", no further communications addressing it are possible. Its local variables (if it has any) are set to an undefined value, to avoid a situation where these variable satisfy "irrelevant" properties, which are unprovable and destroy the completeness of the proof-system. Note that dynamic-concurrency languages do not have an explicit concurrent-composition operator. A process is an implicit concurrent-composition of all its active processes. Since this paper is not concerned with liveness properties (e.g., termination), we omit the discussion of the exact meaning of "immediately". Similarly, the issue of when exactly is a newly created process starting to execute is not further elaborated here either. In [FS86], several possible interpretation of these issues are informally presented.

In the static case, the postcondition of the whole program, called the global postcondition, is the conjunction of the local postconditions of all processes. In the dynamic case a process which was cancelled does not contribute to the global postcondition. The global postcondition is determined in a dynamic manner according to the creations and cancellations taking place during the execution of the program. A process created but not destroyed is called active. A terminated active process is called a surviving process and can still be cancelled. The program is said to be terminated when all active processes have terminated.
2.1 The message-passing language

As mentioned before the language is a dynamic version of CSP [Ho78] with synchronous message-passing communication, similar to the one considered in [dB86]. The body of a process-type, ranged over by $S$, is given by:

$$
S ::= \text{skip} \mid x := E \mid x ? y \mid ? y \mid \text{lt} \mid \text{lt} \mid S_1 \mid S_2 \mid [\cdot] B_1 \rightarrow S_1 \mid \ast \{\cdot\} B_1 \rightarrow S_1 \mid \text{cancel} (x)
$$

$$
E ::= t \mid \text{self} \mid \text{create}(T_1).
$$

We expand the language and the verification method suggested in [dB86] with a cancel ($x$) primitive, where $x$ is a variable holding a process identifier. The generated process identifiers are also used to denote communication partners of interprocess communication primitives: $x ? y$, where $? y$ stands for input and $i$ stands for output. The language also allows for communication with an unspecified partner [Fr83], but at least one partner has to be specified. Thus, the meaning of $x ? y$ is waiting for an input from the process the identity of which is the current value of the variable $x$, while $? y$ means waiting for an input from any process; in both cases, the input value is assigned to the variable $y$. Similarly, $\text{lt}$ means waiting to output the value of $t$ to the process identified by $x$, while $\text{lt}$ means waiting to output the value of $t$ to any process. The communication is delaying, meaning that the process is delayed until a matching partner waits for the dual part of the communication, at which time the communication is enabled and can be executed. Evaluation of an expression $\text{create}(T_1)$ results in a reference to a new process which starts executing $S_1$. Evaluation of cancel ($x$) results in cancelling the process identified by $x$. Issuing cancel to a non-existing process is undefined (causing a run-time error). Free manipulation of variables holding process identifiers is not allowed. A variable can get such a value as a result of executing the command $x := \text{create}(T_1)$ or via communication. Therefore, issuing cancel to non-existing process can only be as a result of a pre-cancellation of that process. The postcondition of the cancelled process does not take part in the global postcondition. Even a process which was cancelled after having terminated does not contribute to the global postcondition. This is the chosen semantic for the cancel primitive but other choices are also possible. The motivation for such an interpretation is that a process is cancelled since presumably there is no further interest in it and its results. Evaluation of the expression self yields the evaluating process' unique identifier $\langle l, i \rangle$. All other constructs have their usual meaning, carried over from the static case.

2.1.1 The operational semantics

We use Plotkin's axiomatic transitions to define an interleaving operational semantics. A state assigns values from some domain $D$ to the program variables which are indexed by a free logical variable $i$, ranging over natural numbers. For example, let $x$ be a local variable to processes of type $T_1$, then each process created by the command $\text{create}(T_1)$ has a private copy of $x$. The variable $x$ can be thought of as an array in which element $x[i]$ is local to process $\langle l, i \rangle$. The free logical variable $i$ is needed to support canonical proofs. The state also assigns values to $C_i$.
and to $A_i$, $1\leq i \leq n$, which are special semantic variables, keeping track of the dynamic aspects of creation and destruction of processes:

1) $C_i$, $1\leq i \leq n$ - records the number of created instances of type $T_i$ (as in [dB86]).

2) $A_i$, $1\leq i \leq n$ - records the set of instance numbers of all active instances of type $T_i$.

A process identifier is a pair $\alpha=<i,j>$, where $1\leq i \leq n$, $1\leq j \leq C_i$. These latter state components are reflected also in the assertion language used in the proof system.

When the programming language contains only a primitive for creation of processes (without cancellation of processes), it is possible to trace the creation of processes by incarnation counters only [dB86]. Those counters $C_i$, $1\leq i \leq n$, count the number of times a creation command for each process-type was executed and are used to generate the unique process identifiers. Adding a primitive for cancellation of a process adds complications since counting creations is not enough. One must keep track of the exact identifiers of the processes which were created and were not cancelled. Those identifiers are kept in the variables $A_i$, $1\leq i \leq n$. The actual role of the $C_i$ counters is serving as a means for generating unique process identifiers.

We use $N$ to denote the natural numbers and $N^*$ for positive natural numbers. We assume an undefined value $\bot$, which is the value of variables of either newly created or of destroyed processes.

A configuration is a pair $CON=<X, \sigma>$, where $\sigma$ is the current state and $X$ is a set of pairs $(\alpha, S(, \sigma \in N^\times N$ being a process identifier and $S$ being the syntactic continuation, the program section still to be executed by the process denoted by $\alpha$. This structure of a configuration is induced by the lack of an explicit concurrent-composition. Every configuration $(X, \sigma)$ satisfies the following well-formedness conditions:

1) $\forall 1\leq i \leq n: (\alpha(2;1) \cdot X, \alpha(1;1) = \sigma(A_i)$

2) $<\alpha, S(, \sigma \in X, \alpha(2;1) = C_i$

3) $<\alpha, S(, <\beta, S(, \sigma \in X \Rightarrow \alpha(2;1) = \sigma(\beta)$,

meaning that in state $\sigma$, $X$ contains exactly one element for each active process and no other elements.

We define a transition relation among configurations, $(X, \sigma) \quad \frac{A}{\text{in}_N \cdot A} \quad (X', \sigma')$. Here $\text{in}_N$ denotes the number of atomic computation steps in the derivation of the configuration to the right of the arrow from the one to the left, and $A$ denotes the set of identifiers of the processes participating in these steps. When $A$ is omitted, it is existentially quantified. The semantic variable $A$ ranges over histories, possibly empty sequences of triples of one of the following forms, where $\alpha$ and $\beta$ are processes identifiers:
... deleted from the variable \( a \), with the intended meaning that \( a \) creates \( p \),
\( \_a, p \), with the intended meaning that \( a \) cancels \( p \),
\( d, a, p \), with the intended meaning that the value \( d \) is sent by \( a \) to \( \beta \).

The recording of histories in the semantics is done to facilitate completeness proofs of the suggested proof rules. The full definition of the semantics is presented in Appendix 1 (where rules common to both languages are marked with '_' ). Only the two axioms for creation and destruction of a process are presented here in more detail.

Several notations are first introduced. Let \( \text{var}(T_i) = \{ y_1, \ldots, y_m \} \) denote the local variables of process-type \( T_i \). Then \( \{ y_1[i], \ldots, y_m[i] \} \) denote the local variables of process \( <i, i> \). The notation \( u/y[i] \) stands for assigning the undefined value \( u \) to the variable \( y[i] \). We use the notation \( \overline{u/\text{var}(T_i)} \) as an abbreviation for \( u/y_1[\sigma(C_i)+1], \ldots, u/y_m[\sigma(C_i)+1] \). Let \( FV(t) \) be all free variables in \( t \). The notation \( t(i) \) is an abbreviation for \( t[x_1[i], \ldots, x_m[i]/y_m] \) where \( FV(t) = \{ x_1, \ldots, x_m \} \). For the description, we stipulate an empty syntactic continuation \( E \) (not in the language itself), satisfying \( E; S = S; E = S \) for every \( S \).

[1] The create-transition axiom:

\[
(X \cup \{<a,x:=\text{create}(T_i)>, \sigma\}, \overline{\alpha, a, \beta}) \xrightarrow{1, \alpha} (X \cup \{<\alpha, E>, <\beta, S_i>\}, \sigma'),
\]

where \( \beta = <i, \sigma(C_i)+1> \) and \( \sigma' = \sigma[\beta(2)/C_i], \sigma(A_i) \cup \{\beta(2)/A_i, \beta x[\alpha(2)/\text{var}(T_i)]\} \).

This axiom expresses the meaning of creating a new instance of process-type \( T_i \) by increasing the creation-counter \( C_i \) by one, and by adding the incarnation number to the variable \( A_i \), which contains all the instance numbers of active processes of type \( T_i \). When a new process is created a new set of variables is created with it, having an undefined initial value, as expressed by the assignment \( \overline{u/\text{var}(T_i)} \). The resulting configuration contains one more element \( <\beta, S_i> \), representing the process just created.

[2] The cancel-transition axiom:

\[
(X \cup \{<a,cancel(x)>, \sigma\}, \overline{\alpha, a, \beta}) \xrightarrow{1, \alpha} (X \cup \{<\beta, ?>\} \cup \{<\alpha, E>, \sigma'\},
\]

if \( \sigma(x) = \beta \), and \( \beta, ?> \in X \), where \( \sigma' = \sigma[\sigma(A_{\beta(1)} \cup \beta(2))/A_{\beta(1)}; \overline{u/\text{var}(T_{\beta(1)})}] \).

Furthermore, if \( <\beta, ?> \in X \) then some error transition (not further specified here) takes place.

Here '?' stands for the (irrelevant) program section about to be executed by the cancelled process \( \beta \).

This axiom expresses the meaning of the cancel command of process \( \beta \). The cancellation is possible only if the process exists, i.e., if \( <\beta, S> \in X \) for some \( S \). The element \( \beta(2) \) (the instance number of the cancelled process) is deleted from the variable \( A_{\beta(1)} \). The counter \( C_{\beta(1)} \) is not affected. The variables of the cancelled process are assigned the value \( u \). If the chosen semantics would not allow a cancellation of a process which has already ter-
minated, then the condition $<\beta, ?> \in X$ should have been changed to $<\beta, S > \in X$ and $S \in E$.

For the formal definition of termination and partial-correctness we need the following definitions:

Definition: A configuration $(X, \sigma)$ is terminal if $\forall <\alpha, S > \in X: S = E$. []

Thus, in a terminal configuration, all surviving processes have an empty continuation.

Definition: A computation of a program $P$ over an initial state $\sigma$ is a maximal (finite or infinite) sequence of configurations $CON_i$, $i \geq 0$, such that:

1) $CON_0 = \langle (<1,1>, S_1), \sigma \rangle$, where $\sigma(C_1) = 1$, $\sigma(A_1) = \{1\}$, and for $2 \leq i \leq n$, $\sigma(C_i) = 0$, $\sigma(A_i) = \emptyset$.

2) If the sequence is infinite then for every $0 \leq i < n$, $CON_i \xrightarrow{h}{1,A} CON_{i+1}$ holds for some $h$ and $A$.

3) If the sequence is finite with last element $CON_m$, then for every $0 \leq i < m$, $CON_i \xrightarrow{h}{1,A} CON_{i+1}$ holds for some $h$ and $A$. []

Definition: A computation is terminating iff it is finite and its last configuration is terminal. A computation is deadlocked iff it is finite and its last configuration is not terminal. []

Program termination is defined as the termination of all processes which were created and were not cancelled during the execution of the program.

A program $P$ is partially-correct w.r.t. precondition $p$ and postcondition $q$ iff for all terminating computations of $P$ starting in an initial state satisfying $p$, the assertion $q$ holds in the final state.

2.2 The shared-variables language

As mentioned before, the language considered here is similar to that of [Og76], but with additional primitives for process creation and destruction and with guarded commands. In particular, we assume that assignments conform to Reynolds' rule of a single reference to a shared variable and are taken to be atomic. A body $S$ of a process-type is given by:

$S ::= D | \text{await } B \text{ do } D \text{ od}

D ::= \text{skip} | x := E | S_1 ; S_2 | [i \in \text{[1..m]} B_i \rightarrow S_i] | [i \in \text{[1..m]} B_i \rightarrow S_i]; \text{cancel}(z)

E ::= t | \text{self} \downarrow \text{create}(T)

The await statement is enabled only in states where $B$ is true, and then $D$ is executed atomically. All other statements have their expected meanings.
2.2.1 The operational semantics

As before, we present here an interleaving operational semantics. The general definitions are very much the same as for the synchronous message-passing language. The following modifications are required:

1) The variables here are not indexed, since they are shared both among instances of different process-types, as well as among instances of the same process-type.

2) The \textit{create} and \textit{cancel} transition axioms do not assign the value \( u \) to the process variables.

3) To express atomicity of the \textit{await} statement, we use the notation \((\text{ln}(S), \alpha)\) to mean an execution of \(S\) of length \(ln\), in which only the process with identifier \( \alpha \) has participated.

One additional rule for the transition relation concerns the \textit{await} statement:

\[
\begin{align*}
& (X \cup \{<\alpha, \text{await } B \text{ do } S \text{ od}>\}, \sigma) \quad h(S) \quad (Y \cup \{<\alpha, E>\}, \sigma'), \sigma = B \rightarrow \\
& (X \cup \{<\alpha, \text{await } B \text{ do } S \text{ od}>\}, \sigma) \quad h(S) \quad (Y \cup \{<\alpha, E>\}, \sigma')
\end{align*}
\]

All other axioms and rules regarding transitions appear in Appendix 1 (marked with "*").

The notions of a computation, termination, deadlock and partial-correctness are defined as for the message-passing case.

3. THE CREATION AND DESTRUCTION AXIOMS

In this section we present the parts of the partial-correctness proof-system that deal with process creation and destruction and are similar for both models. Two axioms are to be used in the local part of the proof-system for the message-passing model. In this model, both assertions \( p \) and \( q \) refer only to variables local to some process-type.

\textit{Process-creation local axiom}: For arbitrary local assertions \( p, q \): \( \{p\} \langle x := \text{create}(T_I) \rangle^{\alpha_0} \{q\} \).

This axiom expresses a "guess" about the value of \( x \), i.e., about the identifier of the newly created process. It is similar in nature to the I/O axioms in [AFR80] and gives rise to a need of a respective cooperation-test in the second, global stage of a proof.

\textit{Process-destruction local axiom}: \( \{p\} \langle \text{cancel}(x) \rangle^{\alpha_0} \{p\} \).

This axiom captures the semantic fact that process cancellation has no side effects on any variables local to the cancelling process. Thus, any precondition is preserved as a postcondition.

The superscript \( i \) around the commands in the message-passing concurrency means that all local variables are indexed by their instance number. In the shared-variables concurrency only one copy of each shared variable exists and \( i \) influences only the value of the term \( \text{self} \).
Two additional axioms, almost identical for both models of concurrency, capture the more global aspects of process creation and destruction. They are used in cooperation proofs in the message-passing model, where the global invariant (discussed below) has access to global information, including auxiliary variables. For their formulation, we make use of the two special variables, \( C \) and \( A_i \), \( 1 \leq i \leq n \), reflecting the values of the corresponding state components (with the same names). These special variables are accessible only to the global invariant. We stress once again, these variables, which formally are shared, cannot be used in assertions belonging to local proofs for the message-passing language, otherwise a non-trivial interference-freedom test would be needed.

**Process-creation global axiom:**

\[
\{ p \left[ A_i \cup \{ C_i + 1 \} / A_i, C_i + 1 / C_i, \{ x / \{ i \}, \bar{u} / \bar{v} / \bar{a} (T) \} \right] (x := \text{create}(T))^{0} (p). \]

This axiom reflects the creation of a new instance of type \( T \) by updating the \( i \)th incarnation counter \( C_i \) and \( A_i \). It also reflects the assignment of the identity of the new process to \( x \). In the message-passing language \( x \) is a local variable of process \( x \{ i \} \), so it is denoted by \( x[ i ] \). In the shared-memory language \( x \) is a shared variable so the indexing is immaterial. The assignment \( \bar{u} / \bar{v} / \bar{a} (T) \) is relevant only in the message-passing language.

**Process-destruction global axiom:**

\[
\{ p [ A_{i+1} \mid \{ x \{ i+1 \} / A_{i+1} \}, \bar{u} / \bar{v} / \bar{a} (T) \} (cancel(x))^{0} (p). \]

This axiom reflects cancellation by updating the variable \( A_{i+1} \), recording that the postcondition of the cancelled process will not be part of the global postcondition. Note that the incarnation counter, \( C_{i+1} \), is left unchanged. The assignment \( \bar{u} / \bar{v} / \bar{a} (T) \) is relevant only in the message-passing model.

In the next sections, we discuss briefly the rest of the local parts of the proof-system and go deeper into the global parts of the proof systems, dealing with interference-freedom and cooperation. These facilitate a concurrent-composition rule, by means of which global facts about the whole program are proved.

### 4. THE MESSAGE-PASSING PROOF SYSTEM

#### 4.1 The local part

In the local part of the proof-system a separate proof-outline is constructed for each process-type. Each process-type might have during the program execution several active instances, each behaving differently according to its interaction with the environment. Still, only one proof-outline suffices for each process-type, a canonical proof-outline. The canonical proofs of [DB86] are adopted here. The canonicity is obtained by using logical variable \( i \) not appearing in the program. Around each program section appearing in an axiom or in a proof-rule we place
a superscript, for example \( p_i S_j^{P_i} T_i \). The meaning of such superscript is that all variable references in the program section \( S_j \) are replaced by the same variables but with an index \( i \), i.e., \( S_j^{P_i} = S_j[x_1, y_1, \ldots, x_m, y_m] \) where \( \{x_1, \ldots, x_m\} \) is the set of all variables free in \( S_j \). In the local assertions all the variables are also indexed by \( i \). The locality of the assertions is ensured by all the local proofs obeying the following restriction.

For every local proof \( \{p_i \} R \{ q_i \} \) from \( S_j \), \( FV(p, q, p_i, q_i) \cap (\text{var}(S_j) \cup \{A_1, \ldots, A_n, C_1, \ldots, C_d\}) = \emptyset \) and every (local) variable \( x \) in \( FV(p, q, p_i, q_i) \cap \text{var}(S_j) \) occurs only subscripted by a free occurrence of \( i \) in \( p, q, p_i, q_i \). Here \( FV \) denotes the collection of all free variables (of the corresponding argument).

A full list of the local axioms and rules is presented in Appendix 2. An example is presented after the description of the global part of the proof-system.

### 4.2 The global part

Next we consider the global stage of partial-correctness proofs for dynamic synchronous message-passing concurrency. In the static proof system [AFR80] we find in the global part a cooperation test. All occurrences of communication commands are put into bracketed sections, and between each two matching bracketed sections a cooperation test is performed. The definition of the bracketed section of [AFR80] is extended here, and such sections are built also around \( \text{create}(T_i) \) commands and \( \text{cancel}(s) \) commands. In addition, a new kind of bracketed sections, called the empty sections, is introduced, to match the create-sections.

**Definition:** A bracketed section, denoted also as BS, is of the form \(<S>\), where \( S:: R_1; R_2; R_3; \ldots; R_k \), and \( R_i \) is a communication, cancel, create or is empty. We refer to \( R_i \) as the core of the BS. Furthermore, in \( R_1, R_2 \) there are no occurrences of communication, create, or cancel statements, and if \( R^{'i}: x::=\text{create}(T_i) \), then \( x \) is not changed in \( R_2 \).

**Definition:** The bracketed sections \(<R_1>, <R_2> \) (syntactically) match iff \( R_1:: R_3; R_1; R_4 \), \( R_2:: R_3; R_2; R_4; R_6 \), and one of the following is the case:

1) \( R:: x::=x, \) and either \( R^{'i}: y::=y \) or \( R^{'i}: x::=x \)
2) \( R:: x::=x \) and \( R^{'i}: y::=y \)
3) \( S:: \text{create}(T_i), \) and \( R^{'i} \) is an empty BS in front of \( S_j \).

**Definition:** A program is properly bracketed iff all occurrences of communication, create and cancel are within BSs, and is from of the body of each process-type a BS with an empty core is embedded.

**Definition:** Let \( P:: <T_1::=S_1, \ldots, T_n::=S_n> \) be properly bracketed. Let \( \{p_i S_j^{P_i} \} T_j \), \( 1 \leq i \leq n \), be \( n \) local proofs (one
for every process-type in \( P \). These proofs cooperate w.r.t. the global invariant \( I \) if for some logical variables \( i, k \) (different from \( j \)):

a. \( \text{FV}(I) \cap \{ i, j, k \} = \emptyset \), and there are no free occurrences in \( I \) of variables which can be changed outside a BS.

b. Let \( \langle \text{pre}(R_1) \rangle < R_1 > \langle \text{post}(R_1) \rangle \), \( \langle \text{pre}(R_2) \rangle < R_2 > \langle \text{post}(R_2) \rangle \) be two matching bracketed-sections (possibly belonging to the same \( S_j \)), then:

\[
(R_1^0 \equiv R_2^0) \\
(\neg (i \land \text{pre}(R_1)[i \land i \text{self}[2] \land \text{pre}(R_2)[k \land k \text{self}[2]])
\]

If the two bracketed-sections are taken from the same process-type, then \( j \neq k \) can be added to the precondition. If one of the bracketed-section is empty then its precondition is omitted.

c. Let \( \langle \text{pre}(R) \rangle < R > \langle \text{post}(R) \rangle \), where \( R = < R_1 ; \text{cancel}(x) ; R_2 > \), then:

\[
i \land \text{pre}(R)[i \land i \text{self}[2] \land \text{post}(R)[i \land i \text{self}[2] \land \text{post}(R)] \equiv \text{true}.
\]

comments: In clause (a) \( i, j \) and \( k \) are forbidden to appear free in \( I \) since \( I \) may contain only global information. Clauses (b) and (c) ensure the validity of \( I \) upon leaving any bracketed-section. Clause (c) actually verifies any assumption made in \( I \) about \( A_j \), i.e., about the identity of the canceled process.

The above definition of the cooperation-test is canonical. The test is carried out between the canonical proof-outlines of the process-types. The cooperation test should also confront each process-type with itself, so that the term \( j \neq k \) can be added to the precondition, since an instance cannot communicate with itself, only with other instances of the same type. The rest of the axioms and rules to be used to accomplish the cooperation test are presented in Appendix 2. We present here only the meta-rule for concurrent composition.

**Concurrent composition rule:**

\[
\frac{\langle p_j \rangle \langle q_i \rangle, 1 \leq i \leq n, \text{cooperate w.r.t. the global invariant } I}{\langle (U \land \text{pre}(U)[i \land i \text{self}[2] \land \text{post}(R)[q_j] \land q_i \land i \text{self}[2] \land \text{post}(R)]) \equiv \text{true} \rangle}
\]

This rule formulates the second stage of a correctness proof, the cooperation between the local proofs. Actually, we should require that the proof-outlines of the separate process-types do not interfere with each other. This test is necessary since we have added the variables \( C_j \) and \( A_k, 1 \leq k \leq n \), which are shared variables among the instances of all types. However, this test degenerates because of the very specific and special use of these variables. These variables are needed only in the global stage of the proof and can appear only in \( I \) and not in the local assertions. The global postcondition is the conjunction of the local postconditions of all processes which were created and were not cancelled. The identities of those processes are in the variables \( A_j, 1 \leq j \leq n \).
Example 1: Consider the program $P::=T_1::=S_1, T_2::=S_2, T_3::=S_3$, together with its proof-outline, presented in Figure 1. A process-type $T_1$ has two potential candidate types $T_2, T_3$ for solving some problem. The (only) process of type $T_1$ creates one instance of each of the two candidate types and communicates with both of them. It receives execution parameters from both processes, then weighs them, leaves only one process and cancels the other, giving the surviving process the only "key" to access some resource exclusively. The global postcondition of the program expresses the fact that exactly one instance of either of the types $T_2, T_3$ survived. The auxiliary variable token keeps track of the number of existing instances, serving as an abstraction of the $A_i$ variables. The auxiliary variable flag eventually holds the identity of the chosen candidate. The global invariant is:

$$I = (\text{token} [1]=0 \Rightarrow C_2=0) \land (\text{token} [1]=1 \land \text{flag} [1]=0 \Rightarrow C_3=0) \land$$

$$(\text{token} [1]=1 \land \text{flag} [1]=1 \land A_2 \leq 1 \land A_3 \geq 1 \land$$

$$(\text{flag} [1]=2 \Rightarrow A_2=1 \land A_3 \leq 0) \land$$

$$(\text{flag} [1]=3 \Rightarrow A_3=1 \land A_2 = 0).$$

We have to check the cooperation between all pairs of BSs which match syntactically. Two such tests are presented below: one regarding communication and the second regarding process creation.

---

$S_1:: (\text{true})<\text{token}:=0; \text{flag}:=0>; (\text{token} [i]=0 \land \text{flag} [i]=0)$

key $i:=1$;

$<s::\text{create} (T_3); \text{token} := \text{token} + 1>; (x [i]=<2, 1> \land \text{token} [i]=1 \land \text{key} [i]=1 \land \text{flag} [i]=0)$

$<y::\text{create} (T_3); \text{token} := \text{token} + 1>; (x [i]=<2, 1> \land \text{token} [i]=1 \land \text{key} [i]=1)$

$<z::\text{create} (T_3); \text{token} := \text{token} + 1>; (x [i]=<2, 1> \land \text{key} [i]=1)$

$<y::\text{create} (T_3); \text{token} := \text{token} + 1>; (x [i]=<2, 1> \land \text{token} [i]=1 \land \text{key} [i]=1)$

$<\text{cancel} (y); \text{token} := \text{token} - 1; \text{flag} := 2>; (x [i]=<2, 1> \land \text{token} [i]=1 \land \text{key} [i]=1 \land \text{flag} [i]=2)$

$<\text{cancel} (y); \text{token} := \text{token} - 1; \text{flag} := 2>; (x [i]=<2, 1> \land \text{token} [i]=1 \land \text{key} [i]=1 \land \text{flag} [i]=2)$

$<\text{cancel} (x); \text{token} := \text{token} - 1; \text{flag} := 3>; (y [i]=<2, 1> \land \text{token} [i]=1 \land \text{key} [i]=1 \land \text{flag} [i]=3)$

$<\text{cancel} (x); \text{token} := \text{token} - 1; \text{flag} := 3>; (y [i]=<2, 1> \land \text{token} [i]=1 \land \text{key} [i]=1 \land \text{flag} [i]=3)$

$S_2:: (true)<> (true)

\text{pri} 2::=\text{calculated value};$

$<\text{key} 2 >; (\text{true})$

$<\text{key} 2 >; (\text{key} 2 [i]=1)$

$S_3:: \text{similar, with pri} 3 \text{ and key} 3 \text{ replacing pri} 2 \text{ and key} 2$, respectively.

---

Figure 1: destruction example
According to clause (b) in the cooperation test, it should be proved that:

$$(x[j]=<2, 1> \land y[j]=<3, 1> \land token[j]=2 \land key[j]=1 \land flag[j]=0 \land I)$$

$$(<x?>priority2j) \parallel (<\text{pr}2k>)$$

$$\{I \land token[j]=0 \land flag[j]=0 \land key[j]=1 \land true\}$$

Using rule (17) (in Appendix 2) and the definition of the superscript:

$$(x[j]=<2, 1> \land y[j]=<3, 1> \land token[j]=2 \land key[j]=1 \land flag[j]=0 \land I)$$

$$\{x[j]?priority2j\parallel <\text{pr}2k>\}$$

Since this cooperation relates to communication, we skip the rest of its details, and pass to the cooperation related to a create statement. By clause (b), we have to show (after index renaming):

$$\{I \land token[j]=0 \land \text{flag}[j]=0 \land \text{key}[j]=1 \land \text{true}\}$$

$$(<x:=create(T_{2})\parallel \text{token}:=\text{token}+1>)\parallel <\text{pr}2k>$$

$$\{I \land x[j]=<2, 1> \land token[j]=0 \land flag[j]=0 \land key[j]=1\}$$

(2)

According to (18) in Appendix 2, this reduces to:

$$\{I \land token[j]=0 \land flag[j]=0 \land key[j]=1 \land true\}$$

$$(<x:=create(T_{2})\parallel \text{token}:=\text{token}+1>)$$

$$\{I \land x[j]=<2, 1> \land token[j]=0 \land flag[j]=0 \land key[j]=1\}$$

(1)

and to

$$\{I \land x[j]=<2, 1> \land token[j]=0 \land flag[j]=0 \land key[j]=1\}$$

$$(\text{token}:=\text{token}+1)^{0}$$

(2)

$$\{I \land x[j]=<2, 1> \land token[j]=1 \land flag[j]=0 \land key[j]=1\}$$.

Showing (2) is trivial. Assertion (1) is proved using [19] (in Appendix 2). We have to show that:

$$\{I \land I \land token[j]=0 \land flag[j]=0 \land key[j]=1 \land true\}$$

$$(<x:=create(T_{3})\parallel <\text{pr}2k>$$

$$\}$$

$$\{I \land x[j]=<2, 1> \land token[j]=0 \land flag[j]=0 \land key[j]=1 \land true\}$$

By using (14-b) (in Appendix 2), we have to show that:

$$\{I \land I \land token[j]=0 \land flag[j]=0 \land key[j]=1 \land true\}$$

$$x[j]:=create(T_{2})$$

$$\{I \land x[j]=<2, 1> \land token[j]=0 \land flag[j]=0 \land key[j]=1 \land true\}$$

However, $I \land I \land I = i = 1$, therefore the precondition contains token(1)=0. According to I, this implies $C_{i}=0$, and by using the global create axiom we get $x[j]=<2, 1>$ in the postcondition.

The global postcondition obtained by the concurrent-composition rule is:
From this we get that $IA_2 + IA_3 = I$, implying that either $IA_2 = I$ and $IA_3 = 0$, or $IA_3 = 1$ and $IA_2 = 0$, which in turn implies that one process only, either of type $T_2$ or of type $T_3$, survived and got the key.

Note that a stronger postcondition for the whole program is also provable (with a somewhat stronger global invariant), relating the identity of the surviving process to the values of the priorities. Here we concentrated on the unique-survivor property.

Example 2: Consider the program $P:: <T_1 = S_1, T_2 = S_2>$, presented in Figures 2.1, 2.2. A network with a ring configuration is dynamically created by the process of type $T_1$. The ring consists of $n$ processes of type $T_2$ and one process of type $T_1$. The distributed program $P$ is a superposition [K87, BF88, CM88] of two known algorithms. The basic algorithm computes the greatest common divisor following the algorithm presented in [AFR80]. The superposed algorithm is a deadlock detection algorithm which is a simplified version of (CMHR83).

Each of the processes reads one external value $\sigma_i$, $1 \leq i \leq n+1$, which is kept in variable $x$. The basic algorithm has the property that when all processes reach a final state and have computed the gcd the program is blocked in a deadlock state since no process knows that all other processes are in final states. The superposed algorithm detects the deadlock state. Then the process of type $T_1$ cancels all other processes of the ring and its local variable $x$ is equal to $gcd(\sigma_1, \ldots, \sigma_n)$. Note that, the ability to cancel processes is used as solution for terminating a blocked program. Methods for terminating a blocked program [Fr80, FR80] are of interest since it may be easier to write such a program than a corresponding program that will terminate.

The variables of the program are:

- $x$- keeps an external value received by the procedure GET ($x$).
- $s_{\text{tok}}$- an array which keeps the identification of the dynamically created processes.
- $\text{right} \quad \text{(left)}$- keeps the identification of the right (left) neighbour in the ring.
- $\text{rtr} \quad \text{(rfl)}$- true if the process is ready to send its value to the right (left) neighbour.
- $\text{status}$- if idle then the process does not wish to send any basic messages to its neighbors.
- $\text{latest}$- an array, where $\text{latest}[i]$ is the largest sequence number $m$ which was send or received by a query ($i$, $m$) communication.
- $\text{wait}$- an array, where $\text{wait}[i]$ is true if and only if the process has been idle continuously since $\text{latest}[i]$ was last updated.
count is an auxiliary variable added to the program which counts the number of copies of process type $T_2$.

It can be proved that when the basic algorithm is blocked the following assertion holds:

$$\forall i \in \mathbb{Z}_+ \quad i = \gcd(c_i, \ldots, c_{n+1})$$

Thus, we use the following global invariant:

$$\exists x \in \mathbb{N} : i = x \cdot \gcd(c_i, \ldots, c_{n+1})$$

$S_1$:

status := active; latest := 0; wait := false; rsl := true; rsr := true; deadflag := true;

j = 1; *(j < n -> <sons [j] = create ($T_2$); count := count + 1; sons [j] := self; j := j + 1;)

j := 2; *(j < n -> sons [j] := sons [j - 1]; j := j + 1; sons [j] := self;)

j := 1; *(j < n - 1 -> sons [j] := sons [j + 1]; j := j + 1; sons [n] := self;)

Comment: A network with a RING CONFIGURATION has been created

GET (x);

right := sons [n]; left := sons [1]; my_id := self [2];

*(deadflag ^ rsl); left := rsl := false

*(deadflag ^ rsl); right := rsl := false

deadflag; right ? query (i, m) -> [status = active -> skip] status := idle

l := [true ^ m = latest [i] ^ l = latest [i] ^ i = my_id] -> right ! reply (i, m)

deadflag; left ? query (i, m) -> [status = active -> skip] status := idle

l := [true ^ m = latest [i] ^ l = latest [i] ^ i = my_id] -> right ! reply (i, m)

Comment: A network with a RING CONFIGURATION has been created

Figure 2.1: destruction of a blocked ring - Process type $T_2$
$S_2$:

status := active; latest := 0; wait := false; rsr := true; rsr := true
?center; ?right; ?left;
GET(x); my_id := self[2];
* [rsr]; left !x \rightarrow rsr := false

rsr; right !x \rightarrow rsr := false

right ?y \rightarrow [y \geq x \rightarrow \text{skip} \; \square \; y < x \rightarrow [y \mid x \rightarrow x := y \; \square \; y \mid x \rightarrow x := x \mod y]]

rsr := true; rsr := true; status := active; wait := false

left ?z \rightarrow [z \leq x \rightarrow \text{skip} \; \square \; z > x \rightarrow [z \mid x \rightarrow x := x \mod z]]

rsr := true; rsr := true; status := active; wait := false

(\neg rsr \land \neg rsr) \rightarrow status := idle; latest [my_id] := latest [my_id] + 1; left !query (my_id, latest [my_id])

right ?query (i, m) \rightarrow [status := active \rightarrow \text{skip} \; \square \; status := idle \rightarrow

[m > latest [i] \rightarrow latest [i] := m; wait [i] := true; left !query (i, m);

\square \text{m} \leq latest [i] \rightarrow [(wait [i] := true \land m = latest [i] \land i = my_id) \rightarrow right !reply (i, m)]

\square (m = latest [i] \land wait [i] := true) \rightarrow

left !reply (i, m) \rightarrow [status := active \rightarrow \text{skip} \; \square \; status := idle \rightarrow

[(m = latest [i] \land wait [i] := true) \rightarrow

[i = my_id \rightarrow right !reply (i, m) \; \square \; i = my_id \rightarrow center !deadlock ()]

\square (m = latest [i] \land wait [i] := true) \rightarrow \text{skip}]]

Figure 2.2: destruction of a blocked ring. Process type $T_2$

GI = \neg deadflag [1] \Rightarrow (L_1 \land x [1] = gcd (\sigma_1, \ldots , \sigma_{n+1})) \land (x [1] = gcd (\sigma_1, \ldots , \sigma_{n+1})) \land A_1 = \{1\} \land count [1] = A_2 \mid i_{id_1}

The loop invariant of $T_1$ is:

$L_1 := (\neg deadflag [i] \Rightarrow count [i] = \neg (\neg deadflag [i] \Rightarrow count [i] = 0))$

The postcondition of $T_1$ is:

$q_1 := \neg deadflag [i] \land count [i] = 0$

According to the concurrent composition rule the postcondition of the program is:

iA_2 := 0 \land x [1] = gcd (\sigma_1, \ldots , \sigma_{n+1})$. 
5. THE SHARED- VARIABLES PROOF SYSTEM

5.1 The local part

In the local part, just as in the message-passing model, each process-type is verified separately in a canonical manner. Since the variables of the program are shared among the processes they are not indexed by the logical variable \(i\). Still \(i\) is used to formulate the local assertions, for example in an assertion of the form: \(f(i=1 \Rightarrow x=i) \wedge (i=2 \Rightarrow x=3)\).

Auxiliary variables can be added to the program and are used for proving interference-freedom among the process-types (as in [OG76]). A problem not occurring in the static case arises here, namely how to initialize the auxiliary variables, in case every process-type may have several instances during execution. Placing the initialization in one of the process-types would cause reinitialization each time a new instance of that process-type is created.

To solve the problem, an auxiliary process-type \(T_{aux}\) is added to the program. Formally, this process-type will have the initially preexisting instance. Its body contains only initialization assignments to the auxiliary variables and exactly one \(create(T_i)\) command.

To handle the auxiliary type formally, we introduce another rule.

**Auxiliary initialization rule:**

\[
(p \land C_{aux}=1 \land \bigwedge_{i=1}^{n} C_i=0 \land A_{aux}(1) \land \bigwedge_{i=1}^{n} A_i=p) \rightarrow p(T_q)
\]

The other axioms and rules needed for the local part are standard and are presented in Appendix 3.

5.2 The global part

Two new features are presented here: the extension of the interference-freedom test and the additional cooperation-test. In the static proof-system [OG76] the global stage of the proof calls for an interference-freedom test. The dynamic proof system presented here contains two tests in the global stage, the cooperation-test and the interference-freedom test. The cooperation test is needed to ensure the assumed precondition of newly created processes. The interference-freedom test is an extension of the interference-freedom test in [OG76]. This extension has two aspects.

The first aspect concerns the self-interference-freedom test. Two instances of the same process type may interfere with each other. In particular, a specific statement in one instance may interfere the same statement in the other instance. If (during execution) this interference does not happen because the two identical statements are synchronized as to never actually run in parallel, the conjunction of their precondition should yield a contradiction:
Concurrent composition rule:

the static concurrent composition rule).

use the new definition of interference in the following rule, combining the local proofs into a global one (replacing

the term self or i properly. For example: if \( \text{pre}(S) = \{(\text{key}=\text{set}(\{2\})\}\), then \((j \neq k) \wedge j = k \Rightarrow \text{false}\).

The second aspect of the extension of the interference-freedom test is the need for bracketed-sections around the create and cancel commands. The interference-freedom should be checked only outside such sections. The candidate commands for interfering are the assignment, await, create and cancel.

Definition: A bracketed section, BS, is of the form <S>, where \( S:: F; R_1; R_2 \), \( R \) is a create or a cancel statement and \( R_1, R_2 \) contain only assignments to auxiliary variables, updating of the auxiliary variable of both the creating and created process (destroying and destroyed, resp.) simultaneously with the creation (destruction, resp.). []

Definition: Let \( P:: <T_1; S_1, \ldots, T_n; S_n> \) be properly bracketed (i.e., all occurrences of create and cancel statements are in BSs). Proofs \( \{p_1\}S_1(q_1), 1 \leq i \leq n \) cooperate if for every BS \( R'=<R_1; x:=\text{create}(T_i); R_2>:\{\text{pre}(R')\}[j \neq k \wedge [i \neq j \wedge \text{key}=\text{set}(\{2\})]\} \). []

Definition: Given a proof \( \{p\}T^0(q) \) and a statement \( S \) with precondition \( \text{pre}(S) \), \( S \) does not interfere with \( \{p\}T^0(q) \) if the following conditions hold:

a. \( \{q[j \neq k \wedge \text{pre}(S)[j \neq k, j \neq i, \text{set}(\{2\})]] \} \), and if \( S \) appears in the same process-type as \( T \) then \( i=j \) can be added to the precondition. The reason is, that self-interference can occur only between two different instances of the same process-type, never between some instance and itself.

b. Let \( S' \) be any statement within \( T \) but not within an await or a bracketed section. Then \( \{\text{pre}(S')\}[j \neq k \wedge \text{pre}(S)[j \neq i, j \neq i, \text{set}(\{2\})]] \} \). If \( S \) appears in the same process-type as \( T \) then \( i=j \) can be added to the precondition. []

Definition: \( \{p_1, S_1, S_2(q_1), \ldots, p_n, S_n(q_n)\} \) are interference free if the following holds. Let \( S \) be an await or assignment (which do not appear in an await or BS) or a BS (which does not appear in an await) of process-type \( T_i \), then for all \( i, 1 \leq i \leq n, S \) does not interfere with \( \{p_i\}S^0(q) \). []

Note that this definition extends the one in [CO76] in that freedom of self-interference is also checked. We use the new definition of interference in the following rule, combining the local proofs into a global one (replacing the static concurrent composition rule).

Concurrent composition rule:
Example: Following (in Figures 3 and 4) is an example of a proof-outline, using only creates (and no cancels, exemplified in the message-passing section). In the example, in order to choose an arbitrary \( x \), \( 15 \leq x < 7 \), a chain of processes of type \( T_2 \) is created, each increasing a variable \( x \) by 1. However, each ceases to affect the value of \( x \) upon the creation of its successor. The noninterference is due to the absence of coexisting modifications of \( x \) by different instances of \( T_2 \). The shared indexed variable \( I[i] \) equals 1 in case instance \( i \) of \( T_2 \) is still able to modify the value of \( x \). Note the propagation of preconditions, where each instance of \( T_2 \) sets the value of variable \( I \) of its successor to 1 upon creating that successor. Furthermore, note that though proper variables are fully shared, the indexed auxiliary variables are associated with specific processes (type instances).

The noninterference test: We check that assignments, await, or BS do not interfere with the proofs.

1. Semantically, there is no need to check the (degenerated) self-interference of statements from \( T_1 \) with the proof of \( T_1 \), because \( C_1 = 1 \) constantly holds, i.e. there are no coexisting different instances of this type. Formally, by using the rule allowing the introduction of the assumption \( i \in A_1 \), we get as the combined precondition \( j \in A_1 \land k \in A_1 \land j \neq k \land 1A_1 = 1 \) which is clearly contradictory.

\[
\begin{align*}
S_1:: (N>1) \\
\{N>1 \land \left( \sum_{i=1}^{C_1} I[i] \right) \leq 1 \land 1 \land C_2 = 0 \}
\end{align*}
\]

\[
x:=1; \{15x < N \land N > 1 \land \left( \sum_{i=1}^{C_1} I[i] \right) \leq 1 \land 1 \land C_2 = 0 \}
\]

\[
[x = e \rightarrow \langle x := \text{create}(T_2), i \{2\} = i > \\
(15x < N \land N > 1 \land \left( \sum_{i=1}^{C_1} I[i] \right) \leq 1 \land 1 \land C_2 = 0) \\
\]

\[
\text{true} \rightarrow \text{skip} \{15x < N \land N > 1 \land \left( \sum_{i=1}^{C_1} I[i] \right) \leq 1 \land 1 \land C_2 = 0 \}
\]

\[
\{15x < N\}
\]

Figure 3: example - first process-type proof-outline
2. Statements from $T_1$ do not interfere with the proof of $T_2$:

2.1 The precondition of $x := x + 1$ contains $C_2 = 0$, contradicting every assertion from $T_2$.

2.2 The precondition of $<z := \text{create} ...>$ contains $C_2 = 0$, as before.

3. Statements from $T_2$ do not interfere with the proofs:

3.1 In $T_1$ only the postcondition and the assertion after the statement $<z := \text{create} ...>$ may be interfered with, since they do not contain the term $C_2 = 0$. The conjunct $1 \leq z < N$ is not interfered with because the precondition of the command $x := x + 1$ in $T_2$ implies $1 \leq x < N$.

3.2 Checking against $T_2$’s proof, we get:

3.2.1 The statement $x := x + 1$ does not interfere with itself since

$$U([\text{self} \{2\}] = 1)^0 \land U([\text{self} \{2\}] = 1)^1 \land j \neq k \land \sum_{i=1}^{N} [i] \leq 1 = \text{false.}$$

It does not interfere with $<v := \text{create} ...>$ for the same reason.

3.2.2 For $<v := \text{create} ...>$ the reasoning is the as in 3.2.1.
The cooperation test: We have to verify the following two conditions, which are easily seen to hold.

\[
\begin{align*}
C_z & \iff \{1 \leq n \land n > 1 \land \sum_{i=0}^{l-1} C_i = 1 \land C_1 = 0\} \\
<z := \text{create } (T_2); i[x\{2\}] := 1> & \iff (1 \leq n \land n > 1 \land \sum_{i=0}^{l-1} C_i = 1 \land C_2 = 1 \land i[x\{2\}] = 1)
\end{align*}
\]

(1)

This condition holds, since \(C_2 = 0\) in the precondition implies that the creation results in \(z = <2, 1>\), and therefore the assignment \(i[x\{2\}] := 1\) is actually \(i[1] := 1\).

\[
\begin{align*}
C_z & \iff \{1 \leq n \land n > 1 \land \sum_{i=0}^{l-1} C_i = 1 \land C_2 = 1 \land i[x\{2\}] = 1\} \\
<v := \text{create } (T_2); i[x\{2\}] := 1; i[\text{self}\{2\}] := 0> & \iff (1 \leq n \land n > 1 \land \sum_{i=0}^{l-1} C_i = 1 \land C_2 = 1 \land i[x\{2\}] = 1).
\end{align*}
\]

(2)

This condition holds since the creation results in \(v = <2, C_2>\) (the new \(C_2\)), and therefore the assignment to \(i\) is actually \(i[C_1] := 1\). Note that \(i[\text{self}\{2\}]\) becomes \(i[1]\) by the cooperation rule. By rule [9] in Appendix 3, \(i \in A_2\) can be added to the precondition, and by definition \(i \in A_2\) implies \(iSC_2\), which can be used in deriving the postcondition.

The final postcondition obtained by the concurrent-composition rule is \(\{1 \leq n\}\), as intended.

5.3 Blocking and deadlock

The `await` statement may delay or block the process attempting to execute it until the statement’s enabling condition \(\delta\) holds. If all active processes are blocked or have terminated and at least one of them is blocked, then the program is in a deadlock. If no execution of the program, which begins with precondition \(p\), ends in a deadlock, then the program is deadlock-free (w.r.t. \(p\)).

A method of proving deadlock freedom was proposed in [OG76] for shared-variables concurrency, and was later adapted by [AFR80] to message-passing concurrency. The method is based on the notion of a situation, which is a representation of the local program-counter of the processes by assertions which are associated with those locations by some given partial-correctness proof. By the soundness of the proof system, the conjunction of the assertions in any given situation holds whenever control of the respective processes is at the corresponding locations. We refer to this as control residing at the respective residence point. According to the proposed method, the assertions corresponding to a blocked situation are shown to be inconsistent. Central to this method is the fact that in the static concurrency case, there is a bounded number, textually determined, of blocked situations in every program. Of course, not every correct proof outline is strong enough to serve also for a deadlock freedom proof.
Trying to apply this method to shared-variable dynamic concurrency raises two problems. In contrast to the static case of [OG76], where each process has exactly one residence-point in every blocked-situation, here a process-type may contribute:

1) More than one residence-point, representing a situation where different active processes of that type simultaneously reside at these points, or

2) No residence-points, representing a situation where there are no active processes of that type, either not having been created, or having all been cancelled.

In addition, the number of instances of each process-type is unbounded, and as a result there is an unbounded number of possible blocked-situations. Hence, it is impossible to contradict each blocked-situation individually, as done by [OG76]. However, we observe that a group of blocked-situations with the same residence-points, but differ from each other by the number of instances residing in those points, called a family of blocked-situations, can be characterized by one assertion. The number of such families is bounded and textually determined for each program. So a check-out of all assertions representing the families suffices to prove deadlock freedom.

5.3.1 A special case

First, consider a simplified problem, where all boolean expressions of all the \texttt{await} statements do not refer to \texttt{self} \{2\}, and all the relevant assertions of the canonical proof-outlines do not refer to the logical indices or to the term \texttt{self} \{2\}. In such programs, two blocked-situations with the same residence-points (await-statements and terminal points), which differ from each other only by the number of instances residing at each of those residence-points, are characterized by the same conditions. For example, let \( \{q_i\} \) be postcondition of process-type \( T_i \), then the residence of some instance of that process-type at the end-point is characterized by the condition \( q_i \). The residence of several instances of that process-type at their end-points is characterized by the condition \( q_i \land \ldots \land q_i = q_i \). As a result the number of blocked-situations that have to be checked is bounded.

The extension of the method of [OG76] to dynamic shared-variables concurrency is explained by the schematic example in Figure 5. In the example, process-type \( T_1 \) has no \texttt{awaits}, \( T_2 \) has two \texttt{awaits}, labeled \( \alpha_1 \), \( \alpha_2 \), respectively, while \( T_3 \) has one \texttt{await} labeled \( \alpha_3 \).

For each process-type \( T_i \), \( 1 \leq i \leq n \), let \( ST_i \) be the collection of the following assertions: \( q_i \) and \texttt{cond} \( (\alpha_j) \) for all \texttt{await} statements \( \alpha_j \) of \( T_i \), where \texttt{cond} \( (\alpha_j) = \texttt{pre} (\alpha_j) \land \neg \beta_j \). In the schematic example:

\[
\begin{align*}
ST_1 &= \{q_1\} \\
ST_2 &= \{q_2, \text{pre} (\alpha_1) \land \neg B_1, \text{pre} (\alpha_2) \land \neg B_2\} \\
ST_3 &= \{q_3, \text{pre} (\alpha_3) \land \neg B_2\}.
\end{align*}
\]
Let $ST = \bigcup_{i \in S} ST_i$. Consider some $\emptyset \neq B \subseteq ST$ with at least one element different from $q_i$. It represents a family of possible blocked-situations.

For example, $B = \{q_1, q_2, \text{cond}(a_1), \text{cond}(a_3)\}$ represents the family of blocked-situations where all active instances of $T_1$ have terminated, some active instances of $T_2$ have terminated, some reside at $a_1$, some reside at $a_2$, and there are no active process of type $T_3$.

The conjunction of the assertions in $B$: $q_1 \land q_2 \land \text{cond}(a_1) \land \text{cond}(a_2)$ is not strong enough to uniquely represent the situations where all active processes are at those residence-points. This conjunction may hold even if other active processes reside at different points in the program. To overcome this difficulty, we introduce for each subset $\emptyset \neq B \subseteq ST$ an assertion $EX_B$, expressing the fact that no other active processes reside at points which are not residence-points in $B$. The assertion $EX_B$ is defined as the conjunction

$EX_B = \bigwedge_{b \in B} \neg EX_{b'}$,

where:

1) $EX_{b'} \land \text{pre}(S)$ is inconsistent for each statement $S$ in $T_b$ which is not a residence-point of $B$.

2) $EX_{b'} \land \text{pre}(S')$ is consistent for $S'$ a residence-point of $B$.

Returning to our schematic example, the assertion: $EX_B = q_1 \land q_2 \land \text{cond}(a_1) \land \text{cond}(a_2)$ holds only in the desired situations. It has to be shown that this assertion is a contradiction, meaning that the program never reaches those blocked-situations. If the conjunction of the assertions in $B$ is itself a contradiction, then the addition of $EX_B$ in that case is not necessary.

We now may formulate the condition for deadlock freedom (w.r.t. a precondition $p$), the natural generation of the Owicki-Gries condition: if all deadlock-candidates (nonempty subset of $ST$ not containing only $q_i$'s) are not consistent then the program is deadlock-free.
Note that the auxiliary initial process-type, \( T_{aux} \), has no effect on the program's deadlock freedom. When proving that \( P \) is deadlock free we can ignore \( T_{aux} \), meaning that no residence-point from \( T_{aux} \) is taken into account.

The generalization of the application to the schematic example to an arbitrary program should be clear. Following is a concrete example of applying the generalized method.

Example: Consider the program in Figure 6. If one of the bracketed-section is empty then its precondition is omitted. Initially, \( x = 0 \), and a chain of processes is created. Each process increments \( x \) by one, and waits until \( x \) reaches the value 20 and then ends. Three auxiliary variables \( e, m \) and \( key \) have been added to the program. The variable \( m \) counts the number of instances of type \( T_1 \) which have already reached the await statement \((a_1)\). The variable \( e \) equals zero until at least one instance of \( T_1 \) passes the await statement \((a_1)\). These two variables \( e \) and \( m \) are needed for defining \( EX_1 \). The variable \( key \) contains the instance number of the process now able to change the variable \( x \), this variable is needed for the interference-freedom test.

An assertion which holds during all instances of \( T_1 \) is:

\[
\neg e = 0 \land x < 20 \land m \leq 1 \land x \neq 20.
\]

Let us check all deadlock-candidates:

1) \( (I \land x < 20 \land m \leq 1 \land x \neq 20) \), i.e., there exist several active instances of \( T_1 \), all of which are blocked at \( a_1 \).
   - For this subset we define: \( EX_1 = m = 1 \land e = 0 \land x < 20 \).
     - The conjunction in \( EX_1 \) with assertions of \( T_1 \) other than pre \((a_1)\) or \( q_1 \) yields \( false \), since \( m < 1 \land e = 0 \land x = 20 \land m \leq 1 \land x \neq 20 \) implies \( false \). The conjunction of \( EX_1 \) with \( q_1 \) also yields \( false \) since \( e = 0 \land e > 0 = false \).
     - The conjunction in \( EX_1 \) with assertions of \( T_1 \) other than pre \((a_1)\) or \( q_1 \) yields \( false \), since \( m = 1 \land e = 0 \land x < 20 \land I \land x < 20 \land m \leq 1 \land x \neq 20 \) implies \( false \).
   - Thus, \( EX_1 \) satisfies the required exclusion conditions. We now show that the corresponding blocked-situation is contradictory. Indeed,
     - \( I \land x < 20 \land m \leq 1 \land x \neq 20 \land m = 1 \land e = 0 \land x \neq 20 \) implies \( x = 20 \), contradicting the conjunct \( x \neq 20 \).

2) \( (x = 20 \land I \land e > 0 \land m \leq 1 \land x \neq 20) \), i.e., there are several active instances of \( T_1 \), some of which have terminated, while the others are blocked at \( a_1 \). Here there is no need to define \( EX_2 \) since the conjunction itself is contradictory.
5.3.2 The general case

We now return to the original problem, where boolean expressions within the \textit{await} statements and the relevant assertions in the canonical proof-outlines (preconditions of \textit{await} statements and postconditions of the process-types) may contain references to logical indices or to the term \textit{self}\{2\}.

An assertion which refers to the term \textit{self} or is indexed by \textit{i} can have different meanings, depending on the instance identification of the process which resides at the point characterized by such an assertion. Therefore the claim suitable for the special case, that all blocked-situations of the same family are characterized by the same \textit{assertion}, does not hold any more.

As in the special case, blocked-situations of the same family are characterized by the same residence-points. However, due to the dependency on the indexing mentioned, the consistency/inconsistency of the blocked-situations is not necessarily the same for all members of one family. For example, in a program which may deadlock, a family can consist of both consistent and inconsistent blocked-situations. When proving deadlock-freedom, it has to be shown for each family that all blocked-situations of that family are inconsistent.

\[
\begin{align*}
S_{\text{AUX}}: \{ x=0 \} \\
& m:=0; e:=0; \\
& \{ x=0 \land m=0 \land e=0 \land C_1=0 \} \\
& \langle z := \text{create}(T_1); \text{key}:=\{2\} > \\
& \{ \text{true} \} \\
S_1: \{ x<20 \land \text{key}=\text{sel}f\{2\} \land m<1A_1 \} \\
& x:=x+1; \\
& \{ x<20 \land \text{key}=\text{sel}f\{2\} \land m<1A_1 \} \\
& \langle c := \text{create}(T_1); m:=m+1; \text{key}:=\{2\} >; \\
& \{ x<20 \land m<1A_1 \}
\end{align*}
\]

\[
\begin{align*}
\square \\
x:=20 \rightarrow \{ x=20 \land \text{key}=\text{sel}f\{2\} \land m<1A_1 \} \\
& m:=m+1 \\
& \{ x=20 \land m<1A_1 \land \text{key}=\text{sel}f\{2\} \land I \} \\
& \langle I \land x<20 \land m<1A_1 \}
\end{align*}
\]

\[
\begin{align*}
a_1: \text{await } x=20 \ 	ext{do} \{ I \land x<20 \land m<1A_1 \} e:=e+1 \ 	ext{od} \\
& \{ x=20 \land I \land e>0 \land m<1A_1 \}
\end{align*}
\]

Figure 6: An example proof-outline for deadlock-freedom
As in the special case, with each process-type $T_i$, $1 \leq i \leq n$, is associated a set $ST_i$ containing the following assertions: $q_i$ and cond$(a_i)$, for each statement $a_i$: await ..., of $T_i$, where cond$(a_i) \equiv pre(a_i) \land \neg B_i$. Recall that those assertions can now depend on $i$ (or on self[2]), whose meaning is the same, therefore from this point on only $i$ will be mentioned.

Let $ST = \bigcup ST_i$. It has to be shown that all deadlock-candidates are inconsistent. Let $\emptyset \neq F \subseteq ST$ be a family with two residence-points, $a_1$ in $T_i$ and $a_2$ in $T_m$ (possibly with $m = i$). All situations in this family contain at least one process residing at $a_1$ and at least one process residing at $a_2$ and none of the processes resides at points other then $a_1$ or $a_2$.

First, consider a blocked-situation where exactly one instance of $T_i$, with instance number $j$, resides at $a_1$ and exactly one instance of $T_m$, with instance number $k$, resides in $a_2$. This situation is characterized by: $\Phi(j) \land \Psi(k) \land EX_B$, where $\Psi(k) \equiv (pre(a_2) \land \neg B_2)(k/i)$, $\Phi(j) \equiv (pre(a_1) \land \neg B_1)(j/i)$ and $EX_B$ expresses that no other processes are at points different from $a_1$ and $a_2$. The notations $\Phi(j)$ and $\Psi(k)$ are chosen to emphasize the dependency of the assertions on the instance number. In the general case, the exclusion assertion $EX_B$ is defined as the conjunction $\bigwedge_{i = 1}^{m} EX_{B,i}$, where:

1) $\forall i: (i \in A_j \land EX_{B,j} \land pre(S) \Rightarrow false)$ for each statement $S$ in $T_i$ which is not a residence-point of $B$.

Explanation: if $pre(S)$ contains restrictions on possible values for $A_i$ then the conjunction with $i \in A_i$ will restrict the check to those values only.

2) $\forall i: (i \in A_i \land pre(S') \land EX_{B,i} \Rightarrow false)$, where $S'$ is a residence-point of $B$. [1]

Going back to our example: To prove that all blocked-situations with exactly one instance at $a_1$ and exactly one instance at $a_2$ are not consistent, it has to be shown that:

\[ (*) \forall j \forall k : (j \in A_j \land \Phi(j) \land k \in A_k \land \Psi(k) \land EX_B) \Rightarrow false. \]

In order to prove that all blocked-situations with exactly two instances at $a_1$ and exactly one instance in $a_2$ are not consistent, it should be proved that:

\[ (**) \forall j \forall k \forall r : (j \in A_j \land \Phi(j) \land k \in A_k \land \Psi(k) \land r \in A_i \land \Phi(r) \land EX_B) \Rightarrow false. \]

It can be seen $(*) \Rightarrow (**)$, and so on... We can conclude that it suffices to prove $(*)$ for deducing that all blocked-situations of that family are not consistent.

Example: Consider the program in Figure 7. Initially, $x=0$ and $i_0=1$. A chain of processes is created, where each process increases the value of variable $x$ by 1 and waits until $x$ reaches the value 20 and then terminates. The creation of the processes chain might be delayed according to the behavior of instances of process-type $T_2$. Process-type $T_1$ has only one instance. It has one residence-point at the end, and an auxiliary variable $e$ is attached to this
point. Process-type \( T_2 \) may have several instances and it has three residence-points: an auxiliary variable \( l \) (array) is attached to the point \( a_1 \), an auxiliary variable \( r \) (array) is attached to the point \( a_2 \), and an auxiliary variable \( m \) (array) is attached to the end point of \( T_2 \). Process-type \( T_3 \) has only one instance. It has one residence point at the end, and an auxiliary variable \( d \) is attached to this point. An auxiliary variable \( key \) holds the instance number of \( T_2 \) which is able to change \( x \). The program is deadlock-free since while \( x < 20 \) the value of variable \( tic \) keeps changing from 1 to 0 and vice versa. Eventually, \( x \) reaches the value 20.

Define:
\[
I = (\text{tic}=0 \lor \text{tic}=1) \land (\text{tic}=0 \Rightarrow l[A_3] \geq 1) \land (l[j]=1 \Rightarrow x=20) \land \forall j \in A_3 (l'[j]=0 \land (r'[j]=0 \lor r'[j]=1) \land (m'[j]=0 \lor m'[j]=1)),
\]
\[
I_0 = m[\text{self}[2]]=0, \quad I_1 = l[\text{self}[2]]=1, \quad r_0 = r[\text{self}[2]]=0, \quad r_1 = r[\text{self}[2]]=1, \quad m_0 = m[\text{self}[2]]=0, \quad m_1 = m[\text{self}[2]]=1.
\]

Define:
\[
ST_1 = \{ \text{a}=1 \land C_1=1 \}\]
\[
ST_2 = \{ l[i]=1 \land r[i]=0 \land m[i]=0 \land x<20 \land l \land \text{key} = i \land \text{tic}=1, \quad l'[i]=0 \land r'[i]=1 \land m'[i]=0 \land x \leq 20 \land x \neq 20, \quad l'[i]=0 \land r'[i]=0 \land m'[i]=1 \land x \leq 20 \}
\]
\[
ST_3 = \{ d=1 \land A_3=1 \land x \geq 20 \land \text{tic}=1 \}
\]

Checking all deadlock-candidates:

(1) \( l[i]=1 \land r[i]=0 \land m[i]=0 \land x<20 \land l \land \text{key} = i \land \text{tic}=1 \)

Define:
\[
EX_{1,1} = C_1=0 \]
\[
EX_{1,2} = A_3=0 \land \text{ tic } = 1
\]
\[
EX_{1,3} = A_3=0
\]

For each \( i \) the above conjunction with the \( EX \)'s yields false since \( \text{ tic }=1 \Rightarrow \text{ tic }=0 \) and \( \text{ tic }=0 \Rightarrow \lnot A_3 \geq 1 \), and this contradict \( EX_{1,3} \).

(2) \( l[i]=1 \land r[i]=0 \land m[i]=0 \land x<20 \land l \land \text{key} = i \land \text{tic}=1, \quad l'[i]=0 \land r'[i]=1 \land m'[i]=0 \land x \leq 20 \land x \neq 20 \)

Define:
\[
EX_{2,1} = C_1=0 \]
\[
EX_{2,2} = A_3=0 \land \text{ tic } = 1
\]
\[
EX_{2,3} = A_3=0
\]
Figure 7: Another example of deadlock-freedom

For all $i$ and for all $j$ we get a contradiction as in (1).

(3) $\forall [i] \land [i] = 1 \land m([i] = 0 \land i \times l \land x \neq 20$

There is no need to define $EX$ since it yields $false$ for each $i$ and $j$ as $x=20 \land x \neq 20$.
There are still several more cases to be checked, but we omit their details.

6. CONCLUSIONS

We have presented proof-rules for partial-correctness of dynamic concurrency with process creation and destruction, both for a shared-variable model and a message-passing model. This is the first time that proof-rules for process destruction are presented. Also, we show that in the proof theory of dynamic concurrency both interference-freedom and cooperation arise in both models, in contrast to the static case, having one of each of these two possibilities per model.

More work needs to be done for providing proof-rules for liveness properties, in particular in view of the more intricate destruction primitives considered in [FS86]. Also, as indicated there, partial (and even total) correctness is only an initial step towards more complicated properties associated with dynamic concurrent programs. Such properties should be identified and rules for their verification should be provided in order to facilitate the proper use of these constructs.

An issue not treated here is that of the soundness and relative completeness of the proposed proof-rules (w.r.t. the operational semantics presented). We believe that these results can be obtained by properly modifying the corresponding results of [dB86] for process creation. The latter themselves were obtained by a modification of Apt's original proof [Ap83] for the [AFR80] system, and the details of this sequence of modifications becomes quite cumbersome and therefore omitted.

ACKNOWLEDGMENTS

We thank Frank deBoer for some clarifications regarding his work, and Shmuel Katz and Amir Pnueli for useful discussions. The part of the second author was partially supported by the Foundation for Research in Electronics, Computers and Communications administered by the Israeli Academy of Sciences and Humanities and by the Fund for the Promotion of Research in the Technion.

REFERENCES


N. Francez, R. Strom: "On alternative semantics of dynamic process creation and destruction" (extended abstract), IBM Watson research center, July 1986.


APPENDIX 1 - rules and axioms of the message-passing semantics

The rules marked with "*" are common to both models.

The transition-axioms

[2]* (X U (X \(<a, create(T))>, \sigma) \xrightarrow{\leq a} (X U (X \(<E, \langle Beta, ?>), \sigma)\)

where \(\sigma=\sigma(I) \cup \langle Beta, \gamma \rangle\), \(\sigma(\alpha(I)) = \langle Beta, \gamma \rangle\), and \(\sigma=\sigma(\alpha(I)) \cup \langle Beta, \gamma \rangle\).

[2]* (X U (X \(<a, cancel(B))>, \sigma) \xrightarrow{\leq a} (X U (X \(<\beta, ?>), \sigma)\)

where \(\sigma = \sigma(A(M(I)) \cup \langle Beta, \gamma \rangle\), \(\sigma(\alpha(I)) = \langle Beta, \gamma \rangle\), and \(\sigma=\sigma(\alpha(I)) \cup \langle Beta, \gamma \rangle\).

The rules marked with '.' are common to both models.

[3] (X U (X \(<a, skip), \sigma) \xrightarrow{\leq a} (X U (X \(<E, \sigma)\).

[4] (X U (X \(<a, x := \alpha), \sigma) \xrightarrow{\leq a} (X U (X \(<E, \sigma)\).

[5] (X U (X \(<a, x := \alpha), \sigma) \xrightarrow{\leq a} (X U (X \(<E, \sigma)\).

[6] (X U (X \(<a, x \neq y), \beta, \gamma), \sigma) \xrightarrow{\leq a} (X U (X \(<E, \gamma), \sigma)\).

[7] (X U (X \(<a, x \neq y), \beta, \gamma), \sigma) \xrightarrow{\leq a} (X U (X \(<E, \gamma), \sigma)\).

[8] (X U (X \(<a, x \neq y), \beta, \gamma), \sigma) \xrightarrow{\leq a} (X U (X \(<E, \gamma), \sigma)\).

[9] (X U (X \(<a, \langle beta, \gamma), \sigma) \xrightarrow{\leq a} (X U (X \(<E, \sigma)\).

If \(\sigma=\sigma(\alpha(\gamma))\), for some \(i \leq m \leq n\).
The rules:

\[ \frac{\langle X \cup \{<a, S_1>\}, \sigma \rangle \xrightarrow{\alpha} \langle X \cup \{<a, S_2>\}, \sigma \rangle}{\langle X \cup \{<a, S_1; S>\}, \sigma \rangle \xrightarrow{\alpha} \langle X \cup \{<a, S_2; S>\}, \sigma \rangle} \]

\[ \frac{\langle X, \sigma \rangle \xrightarrow{\alpha} \langle Y, \sigma \rangle, \langle Y, \sigma \rangle \xrightarrow{\alpha} \langle Z, \sigma \rangle}{\langle X, \sigma \rangle \xrightarrow{\alpha} \langle Z, \sigma \rangle} \]

Here \( \alpha \) is the concatenation of \( h_2 \) to \( h_1 \).

For the appropriate modifications to fit the shared-variables case, see 1) and 2) in section 2.2.1.
APPENDIX 2 - rules and axioms of message-passing proof system

Local part

[1] {p} skip (p).

[2] {p} \{t(i) \{x := t\}(i)\} (x := t)(i) \{p\}

where \( t = \{ x_1, \ldots, x_n \} \).

[3] {p} \{ <i, j > \{ x := self \}(i) \{p\}

where \( x := \text{self} \) is in \( S \).

[4] {p} \{ x := create (T, \{ i \}) \{q\} \}

[5] (p, (x ? y) \{q\},

(\{ t \} \{q\},

(\{ t \} \{q\},

(\{ t \} \{q\},

(\{ t \} \{q\},

[6] (p) \{R_1 \} \{r\}, (p) \{R_2 \} \{q\},

[7] (p \land \bigwedge_{i=1}^m B_i) \{R\} \{q\},

where \( R \{i\} = B_i \{x_1[i]/x_1, \ldots, x_n[i]/x_n\} \).

[8] (p \land \bigwedge_{i=1}^m B_i) \{R\} \{p\},

where \( R \{i\} = B_i \{x_1[i]/x_1, \ldots, x_n[i]/x_n\} \).

[9] {p} \{cancel (x)\} \{p\}.

[10] (p) R_1 \{q\}, (q) R_2 \{q\}

[11] \{p \Rightarrow q\}, (p) \{q\} \Rightarrow q,

\{p\} \{q\}
All local assertions must satisfy the following restriction:

\[ \forall 1 \leq s_n, FV(p) \cap (\text{var}(S_i) \cup \{A_1, \ldots, A_n, C_1, \ldots, C_j\}) = \emptyset, \]

for \( p \) local to the proof-outline of \( S_i \).

**Global part**

[12] \( \{p \in \{A \cup \{C_{i+1}/A_i, C_{i+1}/C_i, \ldots, A_1, C_1\}\} \cap \{x := \text{create}(T)\}\} \)

[13] \( \{p \in \{A \cup \{x \in \{2\} \cup \{A_{n[1]} \cup \ldots \cup \{A_{n[2]} \}\} \} \cap \{x := \text{create}(T)\}\} \)

[14] a. \( R \Rightarrow x <1> \Rightarrow \text{create}(T) \Rightarrow R \)

b. \( R \Rightarrow x <1> \Rightarrow \text{create}(T) \Rightarrow R \)

[15] \( R = x <1> \Rightarrow \text{create}(T) \Rightarrow R \)

[16] \( R = x <1> \Rightarrow \text{create}(T) \Rightarrow R \)

[17] \( R = x <1> \Rightarrow \text{create}(T) \Rightarrow R \)

[18] \( \{p \in \{R \cup \{p_1, p_2\} \} \cap \{R \cup \{p_3, p_4\} \} \} \)

[19] \( \{p \in \{R \cup \{p_1\} \} \cap \{R \cup \{p_3, p_4\} \} \} \)
where $R$ is in $S$. 

$[20] \frac{[p]P'[q]}{[p]P[q]}$

if $FV(q) \cap AUX = \emptyset$, where $P$ is derived from $P'$ by removing all assignments to $x \in AUX$. Here $AUX$ is the set of the auxiliary variables. For each statement $x := t$ in $P'$ if $FV(t) \cap AUX = \emptyset$ then $x \in AUX$.

$[21]$ initialization rule:

\[
\frac{(p \land C_1 = 1 \land \bigwedge_{i \leq 2} C_i = 0 \land A_1 = \emptyset) \land \bigwedge_{i \leq 3} A_i = \emptyset)P(q)}{[p]P[q]}
\]

$[22]$ concurrent composition rule:

proofs $(p) \cup_i (q_i)$, $1 \leq i \leq n$, cooperate w.r.t. the global invariant $I$

\[
[I \land P_1 \{1i, \text{if self}(2)\}] \cup \bigwedge_{i \leq 3} q_i \{j, j' \text{if self}(2)\})
\]
APPENDIX 3 - rules and axioms of the shared-variables proof system.

1. \( (p) \text{skip}(0)(p) \).
2. \( (p \{v^0/Ix/(x:=t)^0\}p \)
   where \( t^0 \in \{[self'\ 2] \} \).
3. \( (p \{[<l, l>\times x]/(x:=self)^0\}p \)
   where \( x:=\text{self} \) is in \( S_i \).
4. \( (p \{A_i \cup \{C_i+1\}/A_i, C_i+1/C_i, \langle l, C_i+1\times x\rangle/(x:=\text{create}(T_i))^0\}p \)
5. \( (p \land B^{(0)}_i)R^{(0)}(q), 1 \leq i \leq m \)
   \( \frac{(p, (R_{i_j} \rightarrow B^{(0)}_i))^0(q)}{(p, B^{(0)} \rightarrow B^{(0)}_i))^0(q) \}
   where \( B^{(0)}_i = B_i \{	ext{self'}\ 2} \) \)
6. \( (p \land B^{(0)}_i)R^{(0)}(p), 1 \leq i \leq m \)
   \( \frac{(p, (R_{i_j} \rightarrow B^{(0)}_i))^0(p \land \neg B^{(0)}_i)}{(p, (R_{i_j} \rightarrow B^{(0)}_i))^0(p \land \neg B^{(0)}_i)} \)
   where \( B^{(0)}_i = B_i \{\text{self'}\ 2} \) \)
7. \( (p \{A_{x+1} \rightarrow (x\ 2)/A_{x+1} \cup (\text{cancel}(x))^0\}p \)
8. \( (p \land B^0)S^{(0)}(q) \)
   \( \frac{(p, \text{await } B \text{ do } S \text{ od})^{(0)}(q)}{(p, \text{await } B \text{ do } S \text{ od})^{(0)}(q)} \)
9. \( (p \land (i \in A_i)) \Rightarrow p_1, (p_1 \{q_1 \Rightarrow (i \in A_i)) \Rightarrow q \)
   \( (p, R^{(0)}(q) \}
   \) where \( R \) is in \( S_i \).
10. \( (p, R^{(0)}(q), (q_1 \land (i \in A_i)) \Rightarrow q \)
   \( (p, R^{(0)}(q) \}
   \)
11. \( (p_2 \Rightarrow p_1, (p_1 \{q_1), q_1 \Rightarrow q \)
   \( (p, P^{(0)}(q) \}
   \)
12. \( (p, P^{(0)}(q) \}
   \( (p, P^{(0)}(q) \}
   \)
(like [20] in Appendix 2).
[13] initialization rule:

\[ (p \land C_1 = 1 \land \bigwedge_{i=2}^n C_i = 0 \land A_{1i} = 1) \land \bigwedge_{i=2}^n A_{i} = \emptyset \] \[ \implies (p \land C_{AUX} = 1 \land \bigwedge_{i=1}^n C_i = 0 \land A_{AUX} = 1) \land \bigwedge_{i=1}^n A_i = \emptyset \] \[ \implies \quad (p \land \{q\}) \]

if \( T_{AUX} \) is needed then the initialization rule is:

\[ (p \land C_{AUX} = 1 \land \bigwedge_{i=1}^n C_i = 0 \land A_{AUX} = 1) \land \bigwedge_{i=1}^n A_i = \emptyset \] \[ \implies \quad (p \land \{q\}) \]

[14] concurrent composition rule:

\[ (p_1 S^p \{q_1\}, \ldots, (p_n S^p \{q_n\}) \text{ are interference free and cooperate} \]

\[ \implies (p_1 \{\ell_1, 1 \in \ell_1 \\}, \ldots, (p_n \{\ell_n, 1 \in \ell_n \\}, 1 \in \{2\})) \]