PROOF RULES FOR COMMUNICATION ABSTRACTS

by

G. Taubenfeld and N. Francez

Technical Report #332

August 1984.
Proof Rules for Communication Abstractions

by

Gadi Taubenfeld and Nissim Francez

Computer Science dept.
Technion, Haifa

Abstract

A modular proof system is presented for proving partial correctness and freedom from deadlock of concurrent programs using scripts (including recursive scripts). Its applications to augmentations of CSP and a subset of ADA are discussed. The proof rules are a generalization of both the procedure rules and the concurrency rules. Correctness proofs for examples are presented.

C.R. Categories:


[F.3.1] Logics and meaning of programs: specifying and verifying and reasoning about programs.

[F.3.3] Logics and meaning of programs: studies of program constructs.

Key Words: proof rule, verification, concurrent programming, deadlock, script, invariants.
1. INTRODUCTION

1.1 Background and overview

In [FH83], a programming language construct named script was introduced, to serve as a communication abstraction mechanism, to be added to any programming language in which concurrent and communication are expressible. The exposition there was informal and concentrated on the concurrency and communication related issues:

The purpose of this paper is to present a more formal definition of the concept by means of proof rules for proving partial correctness and freedom from deadlock assertions about concurrent programs using scripts. There are two main aspects of the script that dictate an approach towards the formulation of the required rules.

(i) The script, viewed as an abstraction, is a multi-party communication and synchronization construct, generalizing the primitives found in most languages for concurrent computation, which involve binary communication and synchronization.

(ii) The (joint) script-enrollment of processes to roles in a script can be viewed as a generalization of the procedure-call mechanism, whereby a "distributed call" consists of each process-calling "its piece" of a procedure, namely a role in the script. The overall effect of a script is reached by means of parameter passing.

Thus, the task is to find a proper amalgamation of proof rules dealing with concurrency and communication with these related to procedures, to form a uniform proof system defining the script construct.

As far as concurrency and communication are involved, our system is a natural extension of what became known as "cooperation proofs". We had to generalize both the sequential proof rules for a process/role to deal with enrollment, and the notion of cooperation, dealing with the concurrent composition. A major
design goal is to introduce into the proof system the same degree of modularity
induced by the script construct on the program.

Thus, we adopted the idea, derived from the proof-theory of procedures, to
prove a "parametric assertion" about a script, which is then adapted to the enrollment
environment by means of a generalization of the adaptation rule and the recursive rule for procedures calls.

To preserve this kind of modularity in proofs of deadlock freedom, we had
also to modify the concept of a "blocked situation" [AFR80,OG76]. Thus, an
enrollment to a script that has a danger of potential deadlock is itself considered
a blocked situation.

The presentation consists of two parts: The first part attempts to present the
verification ideas in a host-language independent way. In the second part we
assume CSP [H078] as a host language, and consider an augmentation of the
proof system presented in [AFR80] to our needs. CSP has been chosen because of
its natural suitability for our context, the availability of established proof sys-
tems for it and the familiarity of the authors with both. We devote also a small
section to the discussion of adopting the ideas to the framework of an ADA subset
dealing with concurrency, for which a version of cooperating proofs also exists.
Nowhere is the dependency on the host language essential.

The results of the paper can be best understood on the basis of previous
knowledge of proof systems for concurrency and procedures. For partial self-
containment a brief review of the functional structure of a script is presented in
the next section. The rest of the paper is organized as follows: In section 3 and 4
we introduce the partial correctness proof system. Section 5 contains a case
study in full details. Section 6 generalizes the proof system to freedom from
deadlock. Finally, in section 7 we extend the proof system for partial correctness
and freedom from deadlock to apply also to recursive scripts.
2. THE STRUCTURE OF A SCRIPT

The main purpose of a script is to serve as an abstraction mechanism, hiding the implementation details of various communication patterns among communicating processes. It is viewed also as a programming language construct that can be added to any host programming language for expressing concurrent programs. It is best conceived in analogy with the procedure construct in languages for sequential programming.

More details about the script (including examples) can be found in [PH83]. We briefly repeat here the functional structure of a script, for (partial) self-containment of the paper:

Basically a script is a parametrized concurrent program section, to which processes enroll in order to participate. It consists of the following components:

- **body** - this is a concurrent composition of disjoint formal processes (i.e. no shared variables), each of which is called a role. Communication among the various roles of a script is expressed using the inter-process communication primitives of the host language.

- **roles** - these are formal processes, to which (actual) processes enroll. Parameter passing is used for interface with a script.

- **data parameters** - these are formal data parameters (as in the case of ordinary procedures) associated with the roles.

In this paper we assume, for simplicity, that the actual parameters, transferred by an actual process to a role, are expressions referring to distinct identifiers. This assumption is motivated similarly to the analogous assumption regarding procedures, avoiding aliasing.

There are two methods of partners-policy enrollment:

- **partners-unnamed enrollment**: upon enrollment a process specifies only its own role (and, of course, the script name).
partners - named enrollment: a process not only specifies the role to which it enrolls, but also names the identities of (some or all of) the other processes it wants to communicate with in the script and their intended roles. In such cases, the processes will jointly enroll in the script only when their enrollment specifications match.

There are also two methods of script initiation and termination:

- delayed initiation: processes must first enroll in the roles of a given script; only then may the execution of that script start.
- immediate initiation: the script is activated upon the enrollment of its first participating process. Other processes may enroll while the script is in progress.

- delayed termination: will free (together) the processes enrolled in a script after the roles have terminated.
- immediate termination: will free each process as soon as it completes its own role.

The case where both initiation and termination are immediate is not treated by this paper. Note that in this case a given process may enroll in several roles of the same script, provided these roles do not communicate with each other within the script's body.

For simplicity, we assume that there is only one instance of a script; however, the proof system can also allow the case of multiple instances of a script.

The collective activation of all the roles of a script is called a performance. The minimum semantic requirement assumed is that all of the roles of a given performance must terminate before a subsequent performance of the same script can begin. Note that a delayed-initiation or delayed-termination policy would automatically guarantee that the successive activations rule is met.

In the examples presented, a mixture of CSP [HO78] and Pascal notations is used as the host language.
CSP's convention for distributed termination of loops is not assumed in this paper.

Example 1: Broadcast:

The first example is of a star-like broadcast script in which a transmitter \((R_1)\) communicates non-deterministically with each of two recipients \((R_2,R_3)\).

**SCRIPT broadcast:**

**INITIATION:**

**TERMINATION:**

\[
\begin{align*}
\text{[ ROLE } R_1 \text{ (VALUE } z_1 \text{: item)} : & \\
\text{VAR } send : \text{ ARRAY } [2..3] \text{ OF boolean; } & \\
\text{send}[2..3] & := \text{false; } & \\
\text{[* } [\{k=2,3\} \text{: send}[k]; R[k]!z_1 \text{: send}[k] := \text{true}] & \\
\end{align*}
\]

\[
\begin{align*}
\text{ROLE } (i=2,3) \text{: } R_i \text{ (RESULT } z_i \text{: item)} : & \\
\end{align*}
\]

The stars "****" stand for \texttt{DELAYED} or \texttt{IMMEDIATE} which determine the method of initiation and termination (which is no concern in this example).

A process may enroll as the transmitter by:

\[
\text{ENROLL-IN broadcast AS } R_1 \text{ (} u \text{, } v \text{, } -1 \text{);}
\]

and as the first recipient by:

\[
\text{ENROLL IN broadcast AS } R_2 \text{ (} w \text{);} 
\]

In this paper we further restrict the Script in two ways:

(1) A role can directly communicate only with other roles of the same script.

(2) The processes enrolling to the same performance of some script are all roles in some other script. The external (main) program is also considered as a
script regarding this restriction.

These restrictions ensure that apart from the actual processes that enroll to some script, no other process can influence the result of a performance of the script. That is so because a process (different from the script roles) can not communicate with the script's roles, neither directly (1) nor indirectly, via another script in which it and the role(s) would enroll (2). These restrictions simplify the design of an inference rule for the script. Without them we would not be able, because of the interaction between scripts, to handle each script separately, as we do later. It also avoids some scoping problems.

A nested enrollment, where a role in one script can enroll in some other script, is allowed. Recursive scripts, where a role can enroll in its own script, and mutual recursion among scripts are allowed only in section 7, where the issue of recursive scripts is treated separately.

Finally, to avoid cumbersome presentation, we consider only scripts that use exclusively either inter-role communication or enroll commands (not both in the same script). External processes can only communicate via enroll commands. The extension to any mixture of primitive inter-process communication and script enrollment is possible but rather technical. The possibility of having nested enroll commands within the body of an accept in the extension to arbitrary mixtures when using ADA is briefly discussed at the end of section four.

3. PROVING PROPERTIES OF SCRIPT BODIES

The way we intend to prove partial correctness of programs that use scripts is closely related to the way procedures are treated [HO71]. First for each body of a script some assertion, relating pre- and post-conditions, is proved; then, using these proofs, an assertion about the main program is proved.

In case of nested enrollments a script regards another script that enrolls in it as a main program, while it is regarded itself as a main program by a script it enrolls in. Hence, to avoid the artificial distinction, from now on we only use the
term script. Everything we say about it relates also to the main program.

With each script we associate an invariant $S_I$ called the **script invariant**, (i.e. each script has its own invariant). Each $S_I$ expresses global information about a script. It may refer to the formal parameters and local variables of all the roles in the script.

When a script uses only primitive inter-role communication, the pre- and post-assertions associated with its body are proved using any proof system of the host language. In case it uses enroll commands (i.e. there are nested enrollments) the system described in the sequel is used.

As in the case of the procedure inference rule [H071], which is used as interface between the procedure call and its body, we present a new proof rule which is a generalization of the procedure rule.

### 3.1 Script enrollment

The definition, $\text{ROLE } R_j (\text{VALUE } x^*_j; \text{VALUE-RESULT } y^*_j, \text{RESULT } z^*_j); B_j$ defines a role $R_j$ with value parameters $x^*_j$, value-result parameters $y^*_j$, result parameters $z^*_j$ and body $B_j$.

For a script $S$ with roles as defined above, the notation $\text{SCRIPT } S (x, y, z); B_s$ is used. Here $x, y, z$ denote the formal parameters of the roles $x^*_1, \ldots, x^*_n_r$: $y^*_1, \ldots, y^*_n_S$, $z^*_1, \ldots, z^*_n_S$, respectively, where $n_S = |S|$ denote the number of roles in the script $S$. Also, $B_s$ denotes the script body $(\frac{n_s}{j} B_j)$

As mentioned above, with any given script $S$, an assertion $\{\text{pre}(S)\} B_s \{\text{post}(S)\}$ can be associated. Both $\text{pre}(S)$ and $\text{post}(S)$ are constructed by conjoining, respectively, the preconditions and postconditions of the various roles with the script invariant.

The formal data parameters referred to by the predicates $\text{pre}(S), \text{post}(S)$ may only be $x, y$ and $y, z$, respectively. They may also refer to constants and free variables to describe initial and final values (called 'logical variables' in [G6]). Note that $z$ must be initialized inside $B_s$, which explains why $\text{pre}(S)$ may
not refer to the result parameters. Also since the value parameters \((\#)\) have no

\(\text{effect on the enrolling processes upon termination of a performance of a script.}\)

\(\text{post}(S)\) may not refer to the value parameters. Again, these restrictions are

- motivated similarly to the analogous restrictions regarding procedures and do

- not restrict generality.

When applying the proof system presented in [AFR80] to a script \(S\) which

- uses CSP's primitive communication commands, the script roles and the predicate

- \(\text{pre}(S)\) correspond, respectively, to processes and a precondition over the initial

- state in CSP programs.

**Example 1.1:** Consider again the broadcast example. Using the proof system

- for CSP described in [AFR80], we may prove:

\[\{x_1 = C\} \text{Broadcast} \{x_2 = x_3 = C\}\]

The proof outline for the script:

\[
\begin{align*}
\text{R1: } & \{x_1 = C\} \text{ send}[2..3] := \text{false}; \\
\text{Li: } & \{x_1 = C\} \\
& [\{ k \in [2,3) \} \rightarrow \text{send}[k]; \text{R}[k]!x_1 \rightarrow \text{send}[k] := \text{true} \{L1\}] \\
& \{L1\} \\
& \| \\
\text{R}(t=2,3) & \{\text{true } R_1?x_1 \{x_1 = C\}\} \\
\end{align*}
\]

In this case, \(S_{1,1}\) is true.

For establishing cooperation we have to prove:

\[\{x_1 = C\} \text{ R}[k]!x_1 || R_1?x_1 \{x_1 = C\} \land z_i = C\} \quad (\text{for } k = i)\]

which is done by applying the communication and preservation axioms and con-

- junction rule. By the parallel composition and consequence rules the proof is

- finished.

\(C\) is free variable "freezing" the initial value of the transmitter and final

- values of all the roles. Because \(\{x_1 = C\} \text{Broadcast} \{x_2 = x_3 = C\}\) is universally true, \(C\)

- may be replaced by any term to yield another universally true statement.
A process $P_i$ can enroll as role $R_j$ in script $S$ using the command $E_j^i(a_i^*, b_i^*, c_i^*)$, where the variables $a_i^*$, $b_i^*$ and $c_i^*$ are the arguments corresponding to the parameters $x_j^i$, $y_j^i$ and $z_j^i$, respectively. The value arguments $a_i^*$ can be expressions. $E_j^i$ is a shorthand notation for ENROLL IN $S$ AS $R_j$.

Definition: $E_1^i, \ldots, E_{n_2}^i$ are matching enrollments if they may enroll to different roles in the same performance of $S$.

By the assumption that initiation termination are not both immediate, no two $E_i$, $E_j^i \neq j$ belong to the same process.

This notion is a natural generalization of that of matching communication commands that is used in verifying CSP programs [AFR80].

Note that from restriction (2) in the script definition above, matching enrollments consist only of enroll commands which are all made by roles from the same script.

We now introduce a new inference rule used as an interface between the enrolling processes and the script. Again, this rule naturally generalizes the 'rule of adaptation' used for procedures.

enrollment rule: for a script $S$ and matching enrollments $E_1^i, \ldots, E_{n_2}^i$,

$$
\{ \text{pre}(S) \} E \{ \text{post}(S) \}
$$

$$
\{ \text{pre}(S)[d; b, z; y] \} \left[ \prod_{j=1}^{n_2} E_j^i(a_j^i, b_j^i, c_j^i) \right] \{ \text{post}(S)[b; c, y; z] \}
$$

where $d, b, c$ denote $a_1^*, \ldots, a_{n_2}^*$; $b_1^*, \ldots, b_{n_2}^*$; $c_1^*, \ldots, c_{n_2}^*$, respectively. By definition all the processes $P_{k_j}$ ($k_j=1..n$) and the roles $R_j$ ($j=1..n_2$) are disjoint.

Here $p[d; y]$ denotes the assertion obtained from $p$ by substituting (simultaneously) $d$ for all free occurrences of $y$.

Explanation: The script $S$ operates on the actual parameters $d; b; c$ in exactly the same way as the body $E_i$ would do with the formal parameters $d; y; z$. Thus it is expected that $\text{post}(S)[b; c, y; z]$ is true after execution of the script provided that $\text{pre}(S)[d; b, z; y]$ was true beforehand.
Furthermore, let $SI$ be the script invariant for $B_s$ referring to the formal parameters. Then, after passing the actual parameters, $SI$ remains invariant (i.e., parameter passing does not affect the invariance of $SI$).

**Example 1.2:** Consider a program $P :: [P_1 || P_2 || P_3]$ using the broadcast script specified above, where:

- $P_1 :: E_1(5)$
- $P_2 :: E_2(c_2)$
- $P_3 :: E_3(c_3)$

($E$ abbreviates here $E^{\text{broadcast}}$)

We prove: $\{\text{true}\} [P_1 || P_2 || P_3] \{c_2 = c_3 = 5\}$

Using the proof that $\{x_1 = C\} B^{\text{broadcast}} \{x_2 = x_3 = C\}$ which was given before, we take $C$ to be 5 and get: $\{x_1 = 5\} B^{\text{broadcast}} \{x_2 = x_3 = 5\}$

By the enrollment rule we get:

$$\frac{\{x_1 = 5\} B^{\text{broadcast}} \{x_2 = x_3 = 5\}}{\{z_1 = 5 / z_1, z_2 = z_2 / z_2, z_3 = z_3 / z_3\}}$$

After substitution we obtain:

$$\{5 = 5\} [E_1(5) || E_2(c_2) || E_3(c_3)] \{c_2 = c_3 = 5\} \quad \Box$$

Note that, as in case of the procedure-call rule (see [GL80]), the enrollment rule is independent of the script body; it depends only on the specification of the body, namely the pre- and post-conditions of the script body. This is a strong argument supporting the use of scripts as an abstraction mechanism.

Before continuing, we would like to contemplate on the meaning of the enrollment rule as a semantic definition of enrollments. As the rule uses substitutions into global states, one may falsely conclude that both delayed initiation and delayed termination are implied.

Enrolling processes need to be synchronized in order for such a global state to be an actual state in the computation satisfying, in particular, the script invariant (after substitution), so that the usual inductive argument can be
applied to deduce the invariant upon total termination.

This, however, is not so. It suffices that at least one event, either initiation or termination be delayed, the other one possibly being immediate. The argument for showing this is a variant on the one used in [EF82], as each performance of a script under such conditions satisfies similar properties to these of communication-closed layers; the only difference is that these layers do not form a cross-section of the whole program, only of the participating processes. We refer the reader to [EF82] for further discussions.

We would like to note also, that the kind of execution induced by these rules is such that processes do local activities until all face enrollments. Then, a whole group, forming a matching enrollment, is advanced one "big step". This generalizes the execution of CSP programs induced by the [AFR80] system, where processes are advanced one pair at the time. For a proof that an arbitrary execution is equivalent to such a serialized one, see [AP83].
Finally, we introduce two new proof rules which are also a natural generalization of those for procedures. The names chosen for the rules are the same as those used for procedures [AP81]. Both of them refer to script S and matching enrolments $E_1, \ldots, E_n$.

**Parameter substitution rule**

$\begin{align*}
\frac{\{p\} \left[ \prod_{j=1}^{n_2} E_j^x(a_{kj}, b_{kj}, c_{kj}) \right] \{q\}}{\{p \ [d; \beta / \delta; \delta]\} \left[ \prod_{j=1}^{n_2} E_j^x(a_{kj}, \delta_{kj}, f_{kj}) \right] \{q \ [\beta; f / \delta; \delta]\}}
\end{align*}$

where $\text{var}(d; \beta; \delta) \cap \text{free}(p, q) \subseteq \text{var}(\delta; \beta; \delta)$.

$d, \delta$ denote a sequence of expressions,

$\beta, \delta, \beta, \delta$ denote a sequence of variables,

$p [d; \beta / \delta; \delta]$ stands for simultaneous substitution of the expressions and variables from $d$ and $\beta$ for those from $\delta$ and $\delta$.

$\text{var}(d; \beta; \delta)$ denotes the set of all variables appearing in $d; \beta$ and $\delta$.

$\text{free}(p, q)$ denotes the set of all free variables of $p$ and $q$.

A similar restriction appears and is explained in [AP81, p. 464].

**Variable substitution rule**

$\begin{align*}
\frac{\{p\} \left[ \prod_{j=1}^{n_2} E_j^x(a_{kj}, b_{kj}, c_{kj}) \right] \{q\}}{\{p \ [\delta; \beta / \delta; \delta]\} \left[ \prod_{j=1}^{n_2} E_j^x(a_{kj}, b_{kj}, c_{kj}) \right] \{q \ [\beta; \delta / \beta; \delta]\}}
\end{align*}$

where $\text{var}(\delta; \beta) \cap \text{var}(d; \beta; \delta) = \emptyset$.

The variable substitution rule is used to rename free variables which are not used as actual parameters. Those free variables are typically used to "freeze" the value of the parameters before enrolling command.

Both rules are useful but not necessary when recursion is not allowed. They are vital when the proof system is later extended to deal with recursion. Examples for using the rules appear in section 7.
4. PROVING PROPERTIES OF ENROLLMENTS

We now introduce the method for proving pre- and post-assertion about a script that uses enroll commands. All the scripts it uses are considered to have been already verified. This proof system is structured similarly to the one for CSP introduced in [AFR80].

We use the term process generically for both a role and an external process. That is so because when a role enrolls in some other script $S$, it can be regarded by $S$, as an external process in case of nested enrollments.

A proof of pre- and post-assertions about a script is done in two stages:

1. separate proofs are constructed in isolation for each component process.
2. the separate proofs are combined by showing that they cooperate.

To generate separate proofs for each process we need the following axiom:

**Enrollment Axiom**: Let $E$ denote any enroll command.

$$\{p\} E \{q\}$$

where $p$ and $q$ refer only to variables local to the process from which $E$ is taken.

This axiom indicates that any post-assertion $q$ can be deduced after an enroll command. Note, however, that $q$ cannot be arbitrary since at stage (2) it must pass the cooperation test. This axiom is a natural generalization of the input/output axioms introduced in [AFR80] for CSP’s communication commands. There the “arbitrariness” of $q$ is explained in more detail.

Using the enrollment axiom and the first eight rules of inference (11-18) which are listed in the appendix, we can establish separate proofs for each process. This is presented, as in [OG76], by a proof outline in which each substatement of a process is preceded and followed by a corresponding assertion.

Remark: the rules for the Alternative and Repetitive statements listed in the appendix are in a format suitable for CSP. Using another host language might cause a suitable modification.
In this proof outline a process 'guesses' the value its parameters will receive after enrollment. When the proofs are combined, these guesses have to be checked for consistency in some way. This is done by the cooperation test.

Note the role of the 'guess' in this proof rule. We may distinguish three levels of 'guessing'.

(i) "small guess" - as presented in proof system for CSP in the form of a "communication axiom" [AFR80]. The "guess" is over the effect of a single communication.

(ii) "moderate guess" - as presented in the proof system for an ADA subset (for concurrency) using the call-accept primitives [GR]. Here the "guess" is over a chain of entry calls, when an accept or call appears within the body of another accept.

(iii) "big guess" - as presented in the current system, "guessing" the effect of an enrollment, that may involve an unbounded number of primitive communications.

We now explain how, at stage (2), the separate proofs are combined.

First we need the concept of bracketing.

Definition. A process $P_i$ is bracketed if the brackets "<" and ">", are interspersed in its text so that,

(i) for each program section $<B>$, $B_i$ is of the form $B_i;E;B_i$ where $B_i$ and $B_i'$ do not contain any enroll commands, and

(ii) all enroll commands appear only within brackets as above.

The purpose of the brackets, as in [AFR80], is to delimit the script sections within which the script invariant need not necessarily hold. Again, a generalization of the situation in the script-free programs is easily recognizable.

With each proof of $\{p\} [P_i] \ldots \| P_n \} q$, we now associate a script invariant $SI$ and an appropriate bracketing. The proof rule concerning parallel composition has the following form:
Parallel Composition rule

proofs of \( \{p_i, q_i\}, i=1, \ldots, n \) cooperate
\[
\frac{\text{cooperate}}{\{p_1, \ldots, p_n, S_i[p_1] \ldots [p_n]\} [q_1, \ldots, q_n, S_i]}.
\]

provided no variable free in \( S_i \) is subject to change outside a bracketed section.

Intuitively proofs cooperate if each performance of a script validates all the post-assertions ('guesses') of the enroll-commands enrolling in this performance.

We now define precisely when proofs cooperate. Assume a given bracketing of a script \( \{p_1, \ldots, p_n\} \) and a script invariant \( S_i \) associated with it.

**Definition:** \( <B_1>, \ldots, <B_{ns}> \) are matching bracketed sections if they contain matching enrollment \( (E_1^1, \ldots, E_{ns}^n) \) to some script \( S \).

**Definition:** The proofs \( \{p_i, q_i\}, i=1, \ldots, n \) cooperate if:

(i) the assertions used in the proof of \( \{p_i, q_i\} \) have no free variables subject to change in \( p_j \) for \( i \neq j \);

(ii) \[ \{ \text{pre}(B_j) \land S_i \} \{ E_j^1, \ldots, E_{ns}^n \} \{ \text{post}(B_j) \land S_i \} \]

holds for all matching bracketed sections \( <B_1>, \ldots, <B_{ns}> \).

The following axiom and proof rule are needed to establish cooperation:

**Enrollment rule:** Parameter substitution rule and Variable substitution rule

as described in the previous section.

**Rearrangement rule:**

\[
\frac{\{p_i, B_i, \ldots, B_{ns}, \{p_1, \ldots, p_{ns}\} \{E_j^1, \ldots, E_{ns}^n\} \}}{\{p_i, \{B_i, E_j^1, \ldots, E_{ns}^n\} \}}
\]

provided \( B_1, B_\ldots, B_{ns}, B_{ns} \) do not contain any enroll commands and \( E_1^1, \ldots, E_{ns}^n \) above are matching enrollments.

The rearrangement rule reduces the proof of cooperation to sequential reasoning, except for an appeal to the enrollment rule. Note that the rearrangement of \( B_1, \ldots, B_{ns} \) and \( B'_1, \ldots, B'_{ns} \) is arbitrary, since they are disjoint in variables. This is a generalization of the binary rearrangement used for CSP, called the 'for-
mation rule' in \([AFR80]\).

For proving cooperation we also need the preservation rule (19, in the appendix). Finally to complete the proof system the substitution rule (110) and the auxiliary variable rule (111) are needed.

Example 1.3 Consider the program \(P_1 \parallel P_2 \parallel P_3\), where:

\[\begin{align*}
P_1 &::= E_2(a_1) \\
P_2 &::= a_2 := 5; E_1(a_2 + 1) \\
P_3 &::= E_3(a_3)
\end{align*}\]

for the rest of the section \(E = E^{\text{broadcast}}\).

Note that \(P_2\) enrolls as the transmitter and \(P_1, P_3\) enroll as recipients.

Using the system above we can prove \(\{\text{true}\} [P_1 \parallel P_2 \parallel P_3] \{a_1 = a_3 = 6 \land a_2 = 5\}\).

The proof outline is:

\[\begin{align*}
P_1 &::= \{\text{true}\} E_2(a_1) \{a_1 = 6\} \\
P_2 &::= \{\text{true}\} a_2 := 5; E_1(a_2 + 1) \{a_2 = 5\} \\
P_3 &::= \{\text{true}\} E_3(a_3) \{a_3 = 6\}
\end{align*}\]

and we may choose \(S_{1,3} = \text{true}\).

There is only one matching enrollment, so for cooperation we must prove:

\(\{a_2 = 5\} [E_1(a_2 + 1) \parallel E_2(a_1) \parallel E_3(a_3)] \{a_1 = a_3 = 6 \land a_2 = 5\}\)

Using the proof that \(\{x_1 = \overline{C}\} B_{\text{broadcast}} \{z_2 = z_3 = \overline{C}\}\) which was given before, we take \(\overline{C}\) to be 6 and get:

\(\{x_1 = 6\} B_{\text{broadcast}} \{z_2 = z_3 = 6\}\)

By the enrollment rule we get:

\[\{x_1 = 6\} B_{\text{broadcast}} \{z_2 = z_3 = 6\}\]

\[\{x_1 = 6; a_2 + 1 / x_1\} [E_1(a_2 + 1) \parallel E_2(a_1) \parallel E_3(a_3)] \{z_2 = z_3 = 6; a_1, a_3 / x_2, x_3\}\]

and after substitution \(\{a_2 + 1 = 6\} [E_1(a_2 + 1) \parallel E_2(a_1) \parallel E_3(a_3)] \{a_1 = a_3 = 6\}\).

By preservation axiom: \(\{a_2 = 5\} [E_1(a_2 + 1) \parallel E_2(a_1) \parallel E_3(a_3)] \{a_2 = 5\}\).

Using the conjunction rule the required cooperation is obtained.
Finally, by applying the parallel composition rule, the proof is finished.

The cooperation test between proofs requires comparisons of all syntactically matching enrollments, even though some of them will never take place during any performance of the script considered.

In this context, the main role of the script invariant $SI$ is to carry global information helping to determine which of the syntactic matches also match semantically. This information is expressed using Auxiliary Variables (different from the program variables) [OG76].

Consider example 1.4:

$$
P_1: \quad E_1(5); \quad E_2(a_2); \quad E_3(a_3);
\quad E_2(a_1); \quad E_1(a_2+1); \quad E_3(a_1)
$$

In this example there are four syntactically matching enrollments (denoted: 1,2,3,4). Two of them, namely (3,4), are not semantically matching enrollment (i.e. will never take place). The other two, namely (1,2), are semantically matching.

We use this example to demonstrate the concept of bracketing and script invariant.

To verify the program, three auxiliary variables $i,j,k$ are used.

**proof outline** (for the bracketed program)

$$\begin{align*}
P_1: & \quad \{i=0\} \quad E_1(5); \{\text{true}\}; \quad \{j=0\} \quad E_2(a_2); \{a_2=5\}; \quad \{k=0\} \quad E_3(a_3); \{\text{true}\}; \quad E_2(a_1); \quad E_1(a_2+1); \quad E_3(a_1)
\end{align*}$$

We choose $SI_{1.4}=i=j=k$.

We now show that the two semantically matching enrollments (1,2) pass the cooperation test. In the other syntactic matching enrollment (3,4), the conjunction of the preconditions contradicts the invariant, so it trivially passes the cooperation test.
(1) We must prove
\[ \{S_1, \Lambda i=j=k=0\} \parallel \{E_1(5), i:=1\} \parallel \{E_2(a_2), j:=1\} \parallel \{E_3(a_3), k:=1\} \{S_1, \Lambda a_2=5 \Lambda a_3=5 \}
\]
Taking \(E_1\) to be 5, we get by the enrollment rule:
\[ \{\text{true}\} \parallel \{E_1(5) \parallel E_2(a_2) \parallel E_3(a_3)\} \{a_2=a_3=5\} \]
By the assignment and preservation axioms:
\[ \{a_2=5\} \parallel i:=1\parallel j:=1\parallel k:=1 \parallel \{i=j=k=1 \Lambda a_2=5\} \]
By applying the consequence and rearrangement rules the proof of (1) is finished.

(2) We must prove
\[ \{S_1, \Lambda a_2=5 \Lambda a_3=5 \} \parallel \{E_1(a_2+1), E_2(a_3) \parallel E_3(a_3)\} \{S_1, \Lambda a_1=6 \Lambda a_2=5\}
\]
from example 1.3 we know that
\[ \{a_2=5\} \parallel \{E_1(a_2+1), E_2(a_3) \parallel E_3(a_3)\} \{a_1=a_3=6 \Lambda a_2=5\}
\]
applying the preservation axiom and the conjunction rule the proof of (2) is finished.

Hence, by the parallel composition, consequence, and auxiliary variables rules:
\[ \{i=0 \Lambda j=0 \Lambda k=0\} \parallel \{P_1 \parallel P_2 \parallel P_3\} \{a_1=a_3=6 \Lambda a_2=5\}
\]
Finally by applying the substitution rule we obtain
\[ \{\text{true}\} \parallel \{P_1 \parallel P_2 \parallel P_3\} \{a_1=a_3=6 \Lambda a_2=5\} \]

Before ending this section we want to clarify a point concerning the extension of the proof system for ADA (presented in [GR]), to any mixture of primitive call-accept communications and script enrollments.

Such extension enables the possibility of having occurrences of enroll commands within the body of an accept, such a phenomenon is not possible in extending the rule to mixtures in CSP.

A similar problem, of having occurrences of calls or accepts, within the body of another accept was resolved in [GR, sec 3] by restricted the notation of brack-
eting in such way that the invariant also holds when such inner calls or accepts are reached.

Applying that method in exactly the same way to enroll commands nested within accept gives an easy and smooth solution. We present below a modified definition for bracketed task; the rest of the details in the extension, as we said before, are rather technical.

Definition: A task is called bracketed if the brackets '〈' and '〉' are interspersed in its text, so that:

(1) for each bracketed section, 〈B〉; B is of the form

(a) \( B_1; \text{CALL } T_a(\text{arguments}); B_2, \)

(b) \( B_1; \text{ENROLL IN's } \text{AS } R_1(\text{arguments}); B_2, \)

(c) \( \text{ACCEPT } b(\text{parameters}) \text{ DO } B_1, \)

(d) \( B_2; \text{ENDACCEPT; } \)

where \( B_1 \) and \( B_2 \) do not contain any entry call or accept or enroll, and may be null statement.

(2) each call, accept and enroll is bracketed as above.
5. EXAMPLES

In this section we present a somewhat larger case study in full detail. We present a script and two different patterns of enrollment to this script, yielding two different effects in the enrolling program.

First the script ROTATE is introduced. It consists of \( m \) roles arranged as a ring configuration. Each role \( R_i \) has a formal parameter \( x_i \) with an initial value denoted by the free variable \( C_i \). Each role \( R_i \) non-deterministically sends its own initial value to its right neighbor \( R_{i+1} \) and receives the initial value of its left neighbor \( R_{i-1} \). (In this section, + and - are interpreted cyclically in \( \{1, \ldots, m\} \).) The effect where each role transfers its initial value to its right neighbor is called rotate right. The indices are used in order to clarify the presentation.

The script declaration:

```
SCRIPT rotate ::

[ ROLE (i=1,m) R_i (VALUE RESULT x_i: integer) ]
```

```
VAR send_i: receive_i: boolean; temp_i: integer;

send_i := false; receive_i := false;
```

```
[ -send_i: R_{i+1}[x_i \Rightarrow send_i:=true

-receive_i: R_{i-1}[?temp_i \Rightarrow receive_i:=true

]: x_i:=temp_i

]
```

We prove: \( \{ \bigwedge_{i=1}^{m} (x_i=C_i) \} \) \( B_{\text{rotate}} \) \( \{ \bigwedge_{i=1}^{m} (x_i=C_{i-1}) \} \)

To verify the script two auxiliary variables \( s_i \) and \( r_i \) are introduced for each role \( R_i \).

Following is the proof outline for the script:

\( R_i : \{ x_i=C_i \land s_i=r_i=false \} \)
send\_i := false; receive\_i := false;

\[ L_i \{ x_i = C_i \land send\_i = s_i \land receive\_i = r_i \} \]

\[
\text{\textit{send\_i} : } < R_{i+1} \{ x_i \rightarrow s_i := \text{true}; send\_i := \text{true} \} \{ L_i \}
\]

\[
\text{\textit{receive\_i} : } < R_{i+1} \{ x_i \rightarrow r_i := \text{true}; receive\_i := \text{true} \} \{ L_i \}
\]

\[
\{ L_i \land receive\_i \land send\_i \} x_i := \text{temp}_i \{ s_i \land r_i \land x_i = \text{temp}_i \}
\]

We choose the script invariant \( SI = \prod_{i=1}^{m} \left( s_i \land r_i \land x_i = \text{temp}_i \right) \)

(note that \( SI \) refers also to local variables).

Matching bracketed sections consist of the first alternative of some \( R_i \) and the second alternative of \( R_{i+1} \), so for establishing cooperation we have to prove

\[
\{ \text{-send\_i} \land \text{-receive\_i+1} \land L_i \land L_{i+1} \land SI \}
\]

\[
\left[ < R_{i+1} \{ x_i := \text{true}; send\_i := \text{true} \} \left\| < R_{i+1} \{ x_i := \text{true}; receive\_i+1 := \text{true} \} \right. \right]
\]

\[
\{ L_i \land L_{i+1} \land SI \}
\]

By the arrow rule it remains to be proved

\[
\{ \text{-send\_i} \land \text{-receive\_i+1} \land L_i \land L_{i+1} \land \prod_{j=1}^{m} \left( (s_j \land r_{j+1}) \rightarrow \text{temp}_{j+1} = C_j \right) \land \text{temp}_{i+1} = x_i \}
\]

\[
\text{\textit{s\_i} := \text{true}; send\_i := \text{true}; r_{i+1} := \text{true}; receive\_i+1 := \text{true} \} \{ L_i \land L_{i+1} \land SI \}
\]

The above precondition is postcondition of: \( R_{i+1} \{ x_i \left\| R_i \right\} \text{temp}_{i+1} \) are inferred by the axiom of communication and preservation.

Using the assignment axiom and consequence rule the required cooperation is obtained.

By the parallel composition rule:

\[
\{ SI \land \prod_{i=1}^{m} \left[ x_i = C_i \land s_i = r_i = \text{false} \right] \} \uplus B_{\text{parallel}} \{ SI \land \prod_{i=1}^{m} \left[ x_i = \text{true}; s_i = \text{true}: x_i = \text{temp}_i \right] \}
\]

The post-assertion \( \{ SI \land \prod_{i=1}^{m} \left[ x_i = \text{true}; s_i = \text{true}; x_i = \text{temp}_i \right] \) implies \( \prod_{i=1}^{m} \left[ x_i = C_{i-1} \right] \)

So, finally, by the consequence, auxiliary variables and substitution rules the required result is obtained. \[ \]

Technion - Computer Science Department - Technical Report CS0332 - 1984
In the next two examples we have again \( m \) processes arranged as a ring configuration. In the first program, using the rotate-script, the effect of "rotate right" is achieved. In the second example, using a different pattern of enrollment to the rotate-script, the effect of "rotate left" is achieved.

**Example 2.1 (rotate right)**

Let

\[ P ::= [ \{ P_i \}_{i=1}^m ] \]

\[ P_i ::= a_i = i; E_i(a_i) \]

For the rest of the section, \( E = \text{rotate} \).

We prove:

\[ \{ \text{true} \} P \{ \Lambda(a_i=i-1) \} \]

**Proof outline:**

\[ P_i ::= \{ \text{true} \} a_i := i \{ a_i = i \} E_i(a_i) \{ a_i = i-1 \} \]

and we may choose \( S_{i,1} = \text{true} \).

For cooperation we must prove:

\[ \{ \Lambda(x_i=i) \} [ \{ \Lambda(x_i=i-1) \} B_{\text{rotate}} \{ \Lambda(x_i=i-1) \} ] \]

We take \( C \) to be \( i \) and get:

\[ \{ \Lambda(x_i=i) \} B_{\text{rotate}} \{ \Lambda(x_i=i-1) \} \]

By the enrollment rule:

\[
\begin{align*}
&\{ \Lambda(x_i=i) \} B_{\text{rotate}} \{ \Lambda(x_i=i-1) \} \\
&\{ \Lambda(x_i=i) \} [ E_i(a_i) \} \{ \Lambda(a_i=x_i) \} ]
\end{align*}
\]

which after substitution yields the required result.

By the parallel composition rule the proof is finished.

**Example 2.2 (rotate left)**

Let

\[ P ::= [ \{ P_i \}_{i=1}^m ] \]

\[ P_i ::= a_i := i; E_{m-i}(a_i) \]

For simplicity, we denote \( m \rightarrow l + 1 \) by \( k_i \). \{\( k_1, \ldots, k_m \)\} is permutation of \{1, \ldots, m\}, so \( P \) has exactly one matching enrollment.
We prove: \( \{ \text{true} \} P \{ \Lambda (a_i = i + 1) \} \).

**Proof outline:** \( P : \{ \text{true} \} a_i := i \{ a_i = i \} E_k (a_i) \{ a_i = i + 1 \} \)

and we may choose \( S(2, 2) = \text{true} \).

for cooperation we must prove: \( \{ \Lambda (a_i = i) \} \{ \prod_{i=1}^{m} E_k (a_i) \} \{ \Lambda (a_i = i + 1) \} \)

(because \( \{ \prod_{i=1}^{m} E_k (a_i) \} \) is the same as \( \{ \prod_{i=1}^{m} E_k (a_i) \} \) we can interchange them)

We take \( C_i \) to be \( k_i \) and get: \( \{ \Lambda (a_i = k_i) \} B_{\text{rotate}} \{ \Lambda (a_i = k_i + 1) \} \)

Because \( k_i = m -(i-1) + z_i - k_i + 1 \), \( \{ \Lambda (a_i = k_i) \} B_{\text{rotate}} \{ \Lambda (a_i = k_i + 1) \} \)

By the enrollment rule we get:

\[
\frac{\Lambda (a_i = k_i) \{ \Lambda (a_i = k_i + 1) \} B_{\text{rotate}} \{ \Lambda (a_i = k_i + 1) \}}{\Lambda (a_i = k_i) \{ \Lambda (a_i = k_i + 1) \} B_{\text{rotate}} \{ \Lambda (a_i = k_i + 1) \}}
\]

and after substitution: \( \{ \Lambda (a_i = k_i) \} \{ \prod_{i=1}^{m} E_k (a_i) \} \{ \Lambda (a_i = k_i + 1) \} \) which is clearly the same as the required proof.

By parallel composition the proof is finished. []

Remark: other definitions of \( k_i \) can cause interesting results, such as rotate \( k \) times ....
6. DEADLOCK FREEDOM

In this section we deal only with the case where both initiation and termination are delayed. We assume every script has an unlimited number of identical instances. When there exist matching enrollments to a script, one of its instances (transparent to the enrolling processes) starts a performance, despite the possibility that other performances of that script are taking place at this moment. From the enrolling processes point of view the script is always available, and there is no need to wait till one performance terminates in order to start a new one. The assumption is essential for the proof system presented in the sequel.

We show how the proof system can be used for proving deadlock freedom of a given program. We assume that there exist a deadlock freedom proof system for the host language, (for example the proof systems presented in [AFR80,GR] for CSP and ADA, respectively)

As in [GR] we use a notion called \textit{frontiers of computation} (f.o.c) which characterizes the set of all commands executing at a given moment. Note that these commands may belong to different scripts, their number is bounded by the number of the (main) program processes, and no two commands may belong to the same process.

A script that started a performance and has not terminated yet is called an active script. A process of an active script, which has not terminated yet, is called an active process.

Deadlock means a state in which the execution cannot proceed, although the program is still active. In the context of scripts this means that at least one process is active, each active process waits in front of a communication command (either an enroll command or a communication primitive of the host language), and no process can proceed. Thus, at the f.o.c, neither primitive communication nor matching enrollment are present in a deadlock.
**Definition**: A program $P$ is deadlock free relative to a precondition $p$ if no execution of $P$, starting in an initial state satisfying $p$, ends in a deadlock.

The approach we use in proving freedom of deadlock is similar to that of the previous sections. Each script $S$ is proved to be deadlock free relative to some assertion denoted by $df(S)$.

Note that $df(S)$ and $pre(S)$ (from the partial correctness proof) need not necessarily be the same. For example for each script $S$, $\{true\}S\{true\}$ holds but if there exists an initial state in which $S$ ends in a deadlock then for proving deadlock freedom $df(S)$ has to be stronger than $true$. Similarly to $pre(S)$, the $df(S)$ predicate may only refer to value parameters, value result parameters and constants. It may not refer to free variables.

The approach we introduce is slightly different from the one introduced in [AFR80,OG76,GR] where, in order to prove deadlock freedom, first all possible deadlock situations (also called blocked situations in [AFR80,OG76] and blocked f.o.c in [GR]) are showed to be unreachable. Using such a method would have forced us to give up modularity handling together all the scripts instead of separating, as we wish to do.

The main idea is that before a script can end in a deadlock it has to pass through a situation which we call a *potentially blocked situation* (p.b.s). A necessary condition (but not sufficient) for a situation to be p.b.s is that each of the scripts own active processes is waiting in front of an enroll command. Note that in contradiction with the f.o.c which may include commands from different scripts, the p.b.s is characterized by a single script's own processes only. Proving deadlock freedom of a script now is done by identifying all its p.b.s and proving that they are unreachable.

When a script uses only primitive inter-role communication its deadlock freedom proof is done using a proof system for the host language. In case it uses enroll command, the system described below is used.
**Example:** The example demonstrates a $df(S)$ predicate associated with a script $S$ which uses CSP's primitive communication only. It is also used later to illustrate the new concept of p.b.s

**SCRIPT $S$:**

\[
\begin{align*}
[ & \text{ROLE } R_1(\text{VALUE-RESULT } x_1: \text{integer}):(x_1 \geq 5 \rightarrow R_2!x_1 \land x_1 \leq 5 \rightarrow R_2?x_1) \\
& \text{ROLE } R_2(\text{VALUE-RESULT } x_2: \text{integer}):(x_2 \geq 5 \rightarrow R_1?x_2 \land x_2 \leq 5 \rightarrow R_1!x_2)
\end{align*}
\]

Using the CSP proof system it is easy to prove that $S$ is deadlock free relative to $df(S) = (x_1 \geq 5 \land x_2 \geq 5) \lor (x_1 \leq 5 \land x_2 \leq 5)$.

The rest of this section is devoted to the formulation of a theorem which provides a sufficient condition for a script, using enroll commands, to be deadlock free. We assume that a specific proof outline is given for each process $P_i i = 1, \ldots, n$ and $SI$ is the script invariant associated with that proof.

**Definition:** A matching enrollment, $E_1^1, \ldots, E_m^m$, is $df$-matching enrollments if $\forall i \in I \left[ \text{pre} (E_i^1(a_i^1, b_i^1, c_i^1)) \land SI \right]$ (the conjunction of all the preassertion of the enroll commands and the script invariant of the enrolling processes) implies $df(t)(d, \delta, z, \gamma)$.

It is easy to see that a performance initiated by a $df$-matching enrollment will not end in a deadlock.

**Definition:** $<B_1, \ldots, B_m>$ are $df$-matching bracketed sections if they contain a df-matching enrollment $(E_1^1, \ldots, E_m^m)$ to some script $S$.

Next we introduce the new concept of potential blocking. Consider a situation of an active script where each of its own active processes waits in front of an enrollment command. Although the processes can not continue at the moment, the state is not necessarily a deadlock because there may be matching enrollments among the enroll commands.

Such a situation is characterized by an $n$-tuple of enrollment capabilities $(e,c)$ associated with the corresponding processes and defined as follows.
Assume that each process waits in front of enroll command or has terminated, then:

(i) in case it has terminated its e.c consists of signalling its termination

(ii) in case it waits in front of an enroll command then its e.c consists of the bracketed section surrounding this enroll command.

The bracketed sections forming an \( n \)-tuple may be partitioned in different ways to form matching bracketed sections. Such a composition of bracketed sections is called a combination. A number of different combinations may be obtained from an \( n \)-tuple, each one indicating a possible path of execution. Note that a combination which does not include any df-matching bracketed sections indicates an execution path which may end in a deadlock, where the script is still in the same situation (i.e., the situation characterize by the \( n \)-tuple that the above combination is obtained from).

**Definition:** A situation, as described above, is a p.b.s if the following two conditions hold:

(a) Among the combinations obtained from the \( n \)-tuple of an e.c there exists a combination which does not include any df-matching bracketed sections.

(b) Not all processes signalling their termination.

Formally, condition (a) is:

\[
\exists C \in \text{combination}(n\text{-tuple}) \forall <B_1>, \ldots, <B_{nt}> \in C \text{ where } (\forall i=1 \ldots nt (\text{pr}(<B_i>) \Delta SI \Rightarrow \text{df} (U)))
\]

where \( \text{combination}(n\text{-tuple}) \) is the set of all combinations obtained from the \( n \)-tuple of e.c.'s which characterize the above situation, \( C \) describes one of those combinations and \( <B_1>, \ldots, <B_{nt}> \) are some matching bracketed sections belonging to \( C \).

To illustrate the concept of potential blocking, consider the following examples with their proof outlines. All the enroll commands refer to the script \( S \) introduced in the previous example. The invariant is identically true in all the examples. In all the examples we consider the situation in which each process waits to
enroll, so condition (b) holds trivially.

(1) let \( P \colon \{a_1 = 6\} \{E_1 | \text{true}\} \parallel \{a_2 = 6\} \{E_2 | \text{true}\} \). There exists one combination only, including a matching enrollment which is a df-matching enrollment. Hence, condition (a) does not apply, and it is not a p.b.s.

(2) let \( P \colon \{a_1 = 6\} \{E_1 | \text{true}\} \parallel \{a_2 = 6\} \{E_1 | \text{true}\} \). There exists one combination only, which does not include any matching enrollments. Hence, condition (a) holds, and the situation is a p.b.s.

(3) let \( P \colon \{a_1 = 6\} \{E_1 | \text{true}\} \parallel \{a_3 = 6\} \{E_2 | \text{true}\} \). There exists one combination only, including a matching enrollment which, clearly, is not a df-matching enrollment. Hence, condition (a) holds, and it is a p.b.s.

(4) let \( P \colon \{a_1 = 6\} \{E_1 | \text{true}\} \parallel \{a_2 = 6\} \{E_1 | \text{true}\} \parallel \{a_3 = 6\} \{E_2 | \text{true}\} \parallel \{a_4 = 6\} \{E_2 | \text{true}\} \). Two combinations can be obtained. In the first, the third and second processes form a df-matching enrollment, while in the second, the third process can also form a matching enrollment, which is not a df-matching enrollment, with the first process. Hence, condition (a) holds, and it is a p.b.s.

(5) let \( P \colon \{a_1 = 6\} \{E_1 | \text{true}\} \parallel \{a_2 = 4\} \{E_1 | \text{true}\} \parallel \{a_3 = 6\} \{E_2 | \text{true}\} \parallel \{a_4 = 6\} \{E_2 | \text{true}\} \). Two combinations can be obtained, both include exactly two matching enrollments which are clearly not df-matching enrollments. Hence, condition (a) holds, and it is a p.b.s.

(6) let \( P \colon \{a_1 = 6\} \{E_1 | \text{true}\} \parallel \{a_2 = 4\} \{E_1 | \text{true}\} \parallel \{a_3 = 6\} \{E_2 | \text{true}\} \parallel \{a_4 = 6\} \{E_2 | \text{true}\} \). Two combinations can be obtained, both include exactly two matching enrollments where, one of them is a df-matching enrollment. Hence, condition (a) does not hold, and it is not a p.b.s.

(7) let \( P \colon \{a_1 = 4\} \{E_1 | \text{true}\} \parallel \{a_2 = 6\} \{E_1 | \text{true}\} \parallel \{a_3 = 4\} \{E_2 | \text{true}\} \parallel \{a_4 = 6\} \{E_2 | \text{true}\} \). Two combinations can be obtained. In the first combination, the first and second processes and the third and fourth processes form two df-matching enrollments, but the second combination includes two matching enrollments which are both not df-matching enrollments. Hence, condition (a) holds, and
it is a p.b.s

Note that if the n-tuple may form only one combination, which does not include any matching bracketed sections, then it is a state of deadlock (as in example (2)).

With each p.b.s we associate an n-tuple of assertions, consisting of the assertions associated with the corresponding processes.

The assertion \( p_i \) associated with a blocked process \( P_i \) is either \( \text{post}(P_i) \) in case it has signalled termination, or, otherwise, it is the pre-assertion of the bracketed section in front of which it waits.

We call an n-tuple \( \langle p_1, \ldots, p_n \rangle \) of assertions associated with a p.b.s a potentially blocked n-tuple.

It is now clear that a script has to pass through a p.b.s before it can end in deadlock. Thus, if it can be proved that all p.b.s are not reachable then deadlock cannot occur and the script is proved to be deadlock free.

This argument is formally expressed in a theorem (similar to theorem 1 in [AFR80, sec. 4]).

**Theorem:** Given a proof of \( \{ df(S) \mid S \} \) with a script invariant \( SI \), \( S \) is deadlock free (relative to \( df(S) \)) if for every potentially blocked n-tuple \( \langle p_1, \ldots, p_n \rangle, \neg \left( \bigwedge_{i=1}^{n} p_i \land SI \right) \) holds.

This theorem provides a method for proving deadlock freedom. The expressed condition is not a necessary one since it depends on a given proof.

In order to prove that \( S \) is deadlock free, we have to identify all potentially blocked n-tuples, and the \( SI \) should be such that a contradiction can be derived from the conjunction of the \( SI \) and the given potentially blocked n-tuple. The arguments supporting this theorem are similar to those appearing in previous discussion of proof of absence of deadlocks, e.g. [AFR80, p. 378].
7. RECURSIVE SCRIPTS AND THEIR VERIFICATION

7.1 The notation of recursive scripts

In [FH83] recursive scripts, where a role can enroll in its own script, were mentioned as natural generalization of scripts. The purpose of this section is to further investigate this option by extending the proof system for partial correctness and deadlock freedom of non-recursive scripts, presented in the previous sections, to apply also to recursive scripts. The presentation here is in terms of direct recursion, but the extension handles mutual recursion as well.

In case of recursion it is obvious that multiple instances of recursive scripts are assumed. The first two restrictions imposed on the script are now applied to each individual instance of a script.

(i) A role can directly communicate only with other roles of the same script instance, and

(ii) The processes enrolling to the same performance of some script are all roles in some other script instance.

All other restrictions and assumptions mentioned in section 2, remain intact.

When initiation is immediate a single recursive enrollment, where a role enrolls in its own script, is sufficient to open a new nested performance. In case of delayed initiation a nested performance is opened only when every role in a script recursively enrolls to its own script to compose a recursive matching enrollments.

Note that a role can recursively enroll to any one of its own script roles, thus the roles which have been enrolled to a nested performance are a permutation of the roles they enrolled to.

To further demonstrate the idea of recursive scripts we now introduce an extensive case study in full detail. Later on, this example will also be verified using the extended proof system presented in the sequel.
7.2 An example: The 'Towers of Hanoi'

The 'Towers of Hanoi' is a game played with three poles, named source, destination, and spare, and a set of discs. Initially all the discs are on the source pole where no disc is placed on top of a smaller one. The purpose of the game is to move all of the discs onto the destination pole. Each time a disc is moved from one pole to another, two constraints must be observed:

1. Only the top disc on a pole can be moved.
2. No disc may be placed on top of a smaller one.

The spare pole can be used as temporary storage.

First, we introduce the well-known sequential solution to the game. It makes use of a recursive procedure which has four parameters. Three of them represent the poles and the fourth is an integer to decide the number of discs to be moved. It consists of three steps. In step one, N-1 discs are moved, using a recursive call, from the source to the spare using the destination as spare. In step two, a single disc is moved from the source to the destination. In step three, N-1 discs are moved, using again a recursive call, from the spare to the destination, using the source as spare.

Next, a solution by a recursive script is introduced. It is similarly structured to the sequential one, and makes use of the same three steps. Although it is distributed, no parallel computation is involved. In a generalization of the game where more than three poles are allowed, parallel computation may take place.

The recursive script, named Hanoi, implementing a winning strategy for the game, is defined as follows: Each one of the three poles is 'in possession' of a different role, represented as a stack of discs. Due to this representation the first constraint is observed trivially. Each of the three roles has two parameters. The first parameter is the number of discs to be moved and the second parameter is the stack itself. We also use an auxiliary simple script named move, which has two roles, named give and take. Each role has one parameter of type stack of disks. The purpose of this script is to move a single element (disc) from the give-role
stack onto the take-role stack.

The strategy of the hanoi script which will correctly play the Towers of Hanoi game with three roles (named also source, destination and spare) and \( N \) discs is described using the same three steps introduced in the sequential solution.

**step 1:** If \( N > 1 \) then \( N-1 \) discs are moved from the source to the spare using the destination as spare. This is done by the source, destination and spare roles recursively enrolling to the source, spare and destination respectively, with first parameter equal to \( N-1 \), while the second parameter is the stack which the role possess.

**step 2:** A single disc is moved from the source to the destination. This is done by the source and destination roles respectively enrolling to the give and take roles in the move script.

**step 3:** If \( N > 1 \) then \( N-1 \) discs are moved from the spare to the destination, using the source as spare. This is done by the source, destination and spare roles recursively enrolling to the spare, destination and source role respectively, with first parameter equal \( N-1 \), the second parameter, as before, is the stack.

### 7.3 proving partial correctness

Next we extend the proof system presented before to recursive scripts. The new proof rule we introduce to deal with recursion is a natural generalization of that for recursive procedures (see [HO71, AP81]). Consider a (recursive) script declaration `SCRIPT S(x,y,z); B_s`, as in section 3.1, where \( B_s \) may include also recursive enrollments. The rule is referring to recursive script \( S \) and matching enrollments \( E_1, \ldots, E_n \).

**recursion rule**

\[
\{ \text{pre}(S) \} \left[ \bigwedge_{j=1}^n E_j(x_j^*,y_j^*,z_j^*) \right] \{ \text{post}(S) \} \quad \longrightarrow \quad \{ \text{pre}(S) \} \ B_s \{ \text{post}(S) \}
\]

\[
\{ \text{pre}(S) \} \left[ \bigwedge_{j=1}^n E_j(x_j^*,y_j^*,z_j^*) \right] \{ \text{post}(S) \}
\]
SCRIPT hanoi :
INITIATION : DELAYED
TERMINATION : DELAYED
[ ROLE source (VALUE n1: integer, VALUE RESULT A: stack of discs) :
  [ n1 = 1 → ENROLL IN hanoi AS source(n1-1, A) ] [ n1 = 1 → skip ]
  ENROLL IN move AS give (A);
  [ n1 = 1 → ENROLL IN hanoi AS spare(n1-1, A) ] [ n1 = 1 → skip ]
]

ROLE destination (VALUE n2: integer, VALUE RESULT B: stack of discs) :
[ n2 ≠ 1 → ENROLL IN hanoi AS spare(n2-1, B) ] [ n2 = 1 → skip ]

ENROLL IN move AS take (B);
[ n2 ≠ 1 → ENROLL IN hanoi AS destination(n2-1, B) ] [ n2 = 1 → skip ]

ROLE spare (VALUE n3: integer, VALUE RESULT C: stack of discs) :
[ n3 ≠ 1 → ENROLL IN hanoi AS destination(n3-1, C) ] [ n3 = 1 → skip ]

[ n3 ≠ 1 → ENROLL IN hanoi AS source(n3-1, C) ] [ n3 = 1 → skip ]

Figure 1. Towers of Hanoi
Figure 2. move

The reasoning presented by the recursion rule is the following: infer
\[ \{ \text{pre}(S) \} \{ \bigwedge_{j=1}^{n} E_j(x_j, y_j, z_j) \} \{ \text{post}(S) \} \] from the fact that \{ \text{pre}(S) \} R \{ \text{post}(S) \} can be proved (using the other rules and axioms) from the assumption
\[ \{ \text{pre}(S) \} \{ \bigwedge_{j=1}^{n} E_j(x_j, y_j, z_j) \} \{ \text{post}(S) \} \]

This is the usual circularity encountered when treated recursion.

We now supplement the proof system presented in sections 3 and 4, with the recursion rule. The extension is enough for dealing with recursion scripts. It may seem peculiar, but no further extension is needed for the cooperation test. The explanation is that once a recursive script is proved (such a proof also involves cooperation tests), that proof applied automatically to each instance of that script. It is so because of using free variables to denote the initial values of the parameter. Therefore, no matter which parameters are transferred to some nested performance, the proof ensures that the instance executing that performance will do it as expected.
7.4 Verifying the 'Towers of Hanoi'

Using the new rules presented, we can now verify the example presented before. First, consider the script move. Using the proof system for CSP ([AFR80]) we may prove:

\[
\{ X = s \cdot X_0 \land Y = Y_0 \} \quad \text{body}_{\text{move}} \quad \{ X = X_0 \land Y = s \cdot Y_0 \}
\]

\( X_0 \) and \( Y_0 \) represent ordered stacks of discs and \( s \) denotes a single disc. They are used to "freeze" the initial state of the stacks \( X \) and \( Y \). By \( s \cdot X_0 \) we mean that \( s \) is placed on top of the disc denoted by \( X_0 \).

It is required that the \( s \) disc will be smaller than any disc in the stacks \( X_0 \) or \( Y_0 \) and that initially no disc is placed on top of a smaller one. Note that these requirements are satisfied (by the actual parameters) when the move script is used (in step 2) by the hanoi script.

The proof outline for move script:

\[
[\text{give} \quad \{ X = s \cdot X_0 \}]
\]

\[
\text{temp}_1 = \text{pop} \ (X)
\]

\[
\{ \text{temp}_1 = s \land X = X_0 \}
\]

\[
\text{take} \ \text{! temp}_1
\]

\[
\{ X = X_0 \}
\]

\[
\text{take} \quad \{ Y = Y_0 \}
\]

\[
\text{give} \ \text{? temp}_2
\]

\[
\{ \text{temp}_2 = s \land Y = Y_0 \}
\]

\[
\text{mish} \ (Y, \text{temp}_3)
\]

\[
\{ Y = s \cdot Y_0 \}
\]

The script invariant \( S_I \) is identically true.

Cooperation is proved easily, using the communication axiom, preservation axiom and consequence rule. By applying the parallel composition rule the proof is finished.
It is simple to see that the constrain that no disc may be placed on top of a smaller one is observed by this script if the initial requirements are satisfied.

Finally, we verify the hanoi script. What we first prove is:

\[ (\star) \{ \text{source: } A = A[1..N] \land B = B_0 \land C = C_0 \land n_1 = n_2 = n_3 = N \} \]

\[ B_{\text{source}}(n_1, A) \land B_{\text{dest}}(n_2, B) \land B_{\text{source}}(n_3, C) \]

\[ \{ A = A[N+1..N] \land B = A[1..N] \land B = B_0 \land C = C_0 \land n_1 = n_2 = n_3 = N \} \]

A[1..W], B_0, C_0 are used for freezing the initial state of the stacks A, B, C respectively. A[1..W] denote an order stack of W discs, where for each i,j such that 1 \leq i < j \leq W, disk A[i] is smaller than disc A[j]. N is an integer 1 \leq N \leq W.

For the sake of the proof we assume that any one of the A[1..W] discs is smaller then any disc of B_0 or C_0. Later we explain why that assumption can be removed. Based on the game definition we assume that initially, no disc is placed on top of a smaller one.

By the recursion rule it suffices to prove:

\[ (\star) \rightarrow \{ \text{source: } A = A[1..N] \land B = B_0 \land C = C_0 \land n_1 = n_2 = n_3 = N \} \]

\[ \text{Body}_{\text{hanoi}} \]

\[ \{ A = A[N+1..N] \land B = A[1..N] \land B = B_0 \land C = C_0 \land n_1 = n_2 = n_3 = N \} \]

Assume \( (\star) \).

The proof outline for the hanoi script:

\[ \{ \text{source: } A = A[1..W] \land n_1 = N \} \]

\[ \begin{align*}
\text{if } n_1 > 1 & \rightarrow \text{Body}_{\text{hanoi}}(n_1 - 1, A) \{ A = A[N..W] \land n_1 = N \} \\
& \quad \text{skip} \quad \{ A = A[N..W] \land n_1 = N \}
\end{align*} \]

\[ \{ A = A[N..W] \land n_1 = N \} \]

\[ \{ \text{move: } A \} \]

\[ \{ A = A[N+1..W] \land n_1 = N \} \]

\[ \begin{align*}
\text{if } n_1 > 1 & \rightarrow \text{Body}_{\text{hanoi}}(n_1 - 1, A) \{ A = A[N+1..W] \land n_1 = N \} \\
& \quad \text{skip} \quad \{ A = A[N+1..W] \land n_1 = N \}
\end{align*} \]
The script invariant is again identically true.

There are exactly three matching enrollments corresponding to steps 1-3, which must be shown to pass the cooperation test.

**step (1):** we must prove

1. \( A = A[1..W] \land B = B_0 \land C = C_0 \land n_1 = n_2 = n_3 = N \)
The proof starts with (\textit{*}).

By variable substitution rule, preservation, conjunction and consequence rules, (exchanging \( N \) with \( N-1 \)),

\[ \{ A = A[N..W] \land B = B_0 \land C = A[1..N-1] \land n_1 = n_2 = n_3 = N-1 \} \]

Now, by the parameter substitution rule (substituting \( B, C, n_2, n_3 \), for \( C, B, n_3, n_2 \) respectively) and variable substitution rule (substituting \( B_0, C_0 \) for \( C_0, B_0 \) respectively),

\[ \{ A = A[N..W] \land B = B_0 \land C = C_0 \land n_1 = n_2 = n_3 = N-1 \} \]

Finally, by the parameter substitution rule (substituting \( n_1-1, n_2-1, n_3-1 \) for \( n_1, n_2, n_3 \) respectively), the required result is obtained. \textit{end step (1)}.

\textbf{step (2): we must prove}:

\[ \{ A = A[N..W] \land B = B_0 \land n_1 = n_2 = N \} \]

Using the the proof that \( \{ x = s \land Y = Y_0 \} \) \textit{Body}move \( \{ X = X_0 \land Y = s \land X_0 \} \) which was given before, we take \( s \land X_0, Y_0 \) to be \( A[N], A[N+1..W], B_0 \) respectively and get:

\[ \{ X = A[N..W] \land Y = B_0 \} \textit{Body}move \{ X = A[N+1..W] \land Y = A[N] \} \]

Note that \( A[N], A[N+1..W], B_0 \) satisfy the initial requirements of the move script.

By the enrollment rule we get,

\[ \{ X = A[N..W] \land Y = B_0 \} \textit{Body}move \{ X = A[N+1..W] \land Y = A[N] \} \]

and after substitution:

\[ \{ A = A[N..W] \land B = B_0 \} \textit{Body}move \{ A = A[N+1..W] \land B = A[N] \} \]

By the preservation axiom: \( \{ n_1 = n_2 = N \} \)
Using the conjunction rule the required cooperation is obtained. \textit{end step (2)}.

\textbf{step (3):} we must prove:

\begin{equation}
\{ A = A[N+1..W] \land B = A[N] \land C = A[1..N-1] \land n_1 = n_2 = n_3 = N \}
\end{equation}

By the parameter substitution rule (substituting respectively, $A, B, C$ for $B, C, A$ and $n_1-1, n_2-1, n_3-1$ for $n_2-1, n_3-1, n_1-1$), and the variable substitution rule (substituting $A[N+1..W]; A[N]B, C_B$ for $B, C, A[N..W]$ respectively)

the required result is obtained. \textit{end step 3}

By applying parallel composition rule, the required result about the body \textit{(Body\textsubscript{hanoi})} is obtained.

Finally, by the recursion rule the proof of (4) is obtained. []

Consider, again, the constrain that \textit{no disc may placed on top of a smaller one}. The only place where that constraint has to be checked is within the move script. It was pointed out that if the initial requirements of the move script are satisfied this constrain is observed, and that always (step 2) the requirements are satisfied. Thus we informally proved that the constrain is observed within the hanoi script, which means that it is an invariant, as required by the game definition.

Consider again the definition of the game. The claim we have just proved is stronger then needed. So, if we now take (4) and use the consequence rule and variable substitution rule to substitute, \textit{empty, empty, empty} for $A[N+1..W], B, C$, where \textit{empty} denote an empty stack, we get:

\begin{equation}
\{ A = A[1..N] \land B = C = \text{empty} \land n_1 = n_2 = n_3 = N \}
\end{equation}

Technion - Computer Science Department - Technical Report CS0332 - 1984
which is exactly what was defined as the object of the game.

Note that the last formula cannot be proved directly using the recursion rule because of step 3. Note also that once the proof is finished the assumption assuming that any one of the A[1..W] discs is smaller than any disc of B₀ or C₀ is not needed any more. That is so because of the substitution of empty stacks for B₀ and C₀.

7.5 Deadlock freedom

In this section we extend the proof system introduced for proving deadlock freedom of non recursive scripts, presented in section 6, to apply also to recursive scripts. All assumption made there are also adopted here.

In such an extension it has to be shown how to prove that a recursive script S is deadlock free relative to some assertion denoted by df(S). When using the 'old' proof system the problem which arises is how to decide whether a recursive matching enrollments is a df-matching enrollments or not. Such a decision is based on knowing the assertion relative to which the script, which the matching enrollments enroll to, is deadlock free. In the case of recursive matching enrollments that assertion (df(S)) is actually the one to be proved. The solution is the one usually encountered when treating recursion, to permit the use of the desired conclusion about an enrollment, as an assumption in the proof of the body.

Thus, to decide whether a recursive matching enrollments to script S is a df-matching enrollment or not, we assume that S is deadlock free relative to df(S). After all the recursive matching enrollments have been decided, we 'forget' the above assumption and continue as usual. If from that point, using the already known proof system, it is provable that S is deadlock free relative to df(S), then indeed it is.
B. CONCLUSION

We have presented a proof system for proving partial correctness and deadlock-freedom of concurrent programs using scripts.

By separating the proof of the whole program and handling each script separately we achieved the goal of modularizing the proof system to the same degree of modularity introduced by the script construct. To achieve this modularity in the system the enrollment rule and the recursive rule, which are used as interface between the enrolling processes and the script, are introduced. Those proof rules are generalization of the adaptation rule and recursive rule for procedure call. Also, the idea of cooperating proofs was extended to our context. Although we have mentioned only CSP and ADA, we do believe that the basic ideas of the proof system can be applied to most concurrent programming languages when augmented with scripts.

In future research, the question of the of the completeness of the proof system should be studied, as well as its extension for proving termination. Another issue is extending the enrollment mechanism to serve as a guard. Finally, efficient implementations, especially distributed ones, are of primary concern.
APPENDIX

Notations

$S$ : script named $S$.

$|S|, n_S$ : number of roles in the script $S$.

$E_f (\alpha)$ : enroll in $S$ as $R_f (\alpha)$.

$R_f (x_i^s)$ : role $R_f$ in script $S$ with formal data parameters $x_i^s$, and body $B_f$.

$B_s$ : Body of $S \big( \bigcup_{j=1}^{n_S} B_j \big)$.

$\text{pre}(R_f)$ : pre-condition of $R_f$.

$\text{post}(R_f)$ : post-condition of $R_f$.

$S_I$ : script invariant.

$\text{pre}(S)$ : pre-condition of $B_s$.

$\text{post}(S)$ : post-condition of $B_s$. ($\bigwedge_{j=1}^{n_S} \text{post}(R_f) \land S_I \rightarrow \text{post}(S)$).

$d_f(S)$ : predicate relative to which $S$ is proved to be deadlock free.

(Indices are not used when not necessary).

Axioms and proof rules.

11. Assignment axiom

$$\{p[t/x]\} x := t \{p\}$$

12. Skip axiom

$$\{p\} \text{skip} \{p\}.$$

13. Alternative Command rule

$$\frac{\{p \lor b\} S_i \{g\}, i = 1, \ldots, m}{\{p\} \lor \bigwedge_{i=1}^{m} (i = 1, \ldots, m) b_i \rightarrow S_i \{g\} \{p\}}$$

14. Repetitive Command rule

$$\frac{\{p \lor b\} S_i \{p\}, i = 1, \ldots, m}{\{p\} \lor \bigwedge_{i=1}^{m} (i = 1, \ldots, m) b_i \rightarrow S_i \{p \lor (b_1 V \ldots V b_m)\} \{p\}}.$$
15. Composition rule

\[ \{p\} S_1 \{q\}, \{q\} S_2 \{r\} \]
\[ \{p\} S_1 \cup S_2 \{r\} \]

16. Consequence rule

\[ \{p\} \rightarrow_0 \{p_1\}, \{q_1\} \rightarrow_0 \{q\}, \{p\} S \{q\} \]

17. Conjunction rule

\[ \{p\} S \{q\} \]

18. Disjunction rule

\[ \{p_1\} S \{q\}, \{p_2\} S \{q\} \]
\[ \{p_1 \cup p_2\} S \{q\} \]

19. Preservation axiom

\[ \{p\} S \{p\} \]

provided no free variable of \( p \) is subject to change in \( S \).

Note that the skip axiom is subsumed by the preservation axiom.

20. Substitution rule

\[ \{p\} S \{q\} \]
\[ \{p[t/z]\} S \{q\} \]

provided \( z \) does not appear free in \( S \) and \( q \).

The substitution rule is needed to eliminate auxiliary variables from the preassertion.

21. Auxiliary Variables rule

Let \( AV \) be a set of variables such that \( x \in AV \) implies \( x \) appears in \( S \) only in assignments \( y := t \), where \( y \in AV \). Then if \( q \) does not contain any variables from \( AV \), and \( S \) is obtained from \( S \) by deleting all assignments to variables in \( AV \),

\[ \{p\} S \{q\} \]
\[ \{p\} S \{q\} \]
list of the new rules

enrollment rule

\[ \{\text{pre}(S)\} \to \text{B}_a \to \{\text{post}(S)\} \]

\[\{\text{pre}(S)[d;\hat{\epsilon} \mapsto x;\hat{\eta}]; \to \] \[\sum_{j=1}^{n_a} E_f(a_{k_j}, b_{k_j}, c_{k_j}); \to \] \[\{\text{post}(S)[d;\hat{\epsilon} \mapsto x;\hat{\eta}]\} \]

parameter substitution rule

\[ \{p\} \to \] \[\sum_{j=1}^{n_a} E_f(a_{k_j}, b_{k_j}, c_{k_j}); \to \] \[\{q\} \]

\[\{p[d;\hat{\epsilon} \mapsto x;\hat{\eta}]; \to \] \[\sum_{j=1}^{n_a} E_f(a_{k_j}, b_{k_j}, c_{k_j}); \to \] \[\{q[\hat{\epsilon};\hat{\eta}]\} \]

where \(\text{var}(d;\hat{\epsilon};\hat{\eta}) \cap \text{free}(p, q) = \text{var}(d;\hat{\epsilon};\hat{\eta})\).

variable substitution rule

\[ \{p\} \to \] \[\sum_{j=1}^{n_a} E_f(a_{k_j}, b_{k_j}, c_{k_j}); \to \] \[\{q\} \]

\[\{p[s;\hat{\epsilon} \mapsto x;\hat{\eta}]; \to \] \[\sum_{j=1}^{n_a} E_f(a_{k_j}, b_{k_j}, c_{k_j}); \to \] \[\{q[s];\hat{\eta}]\} \]

where \(\text{var}(s;\hat{\epsilon}) \cap \text{var}(d;\hat{\epsilon};\hat{\eta}) = \emptyset\).

enrollment axiom

\[ \{p\} \to \{q\} \]

rearrangement rule

\[ \{p\} B_1 \cdots B_n \to \{p_1\} \to \] \[\sum_{j=1}^{n_a} E_f(a_{k_j}, b_{k_j}, c_{k_j}); \to \] \[\{p_1 \to B_1 \cdots B_n; \to \{q\}\} \]

\[\{p_2\} \to \{q\} \]

recursion rule

\[ \{\text{pre}(S)\} \to \] \[\sum_{j=1}^{n_a} E_f(x_j, y_j, z_j); \to \] \[\{\text{post}(S)\} \to \] \[\text{pre}(S) \to \] \[\text{B}_a \to \] \[\text{post}(S)\]

\[\{\text{post}(S)\} \to \] \[\sum_{j=1}^{n_a} E_f(x_j, y_j, z_j); \to \] \[\{\text{post}(S)\} \to \]
REFERENCES


