Science of Computer Programming 6 (1986) 35-88
North-Holland

SCRIPT: A COMMUNICATION ABSTRACTION MECHANISM AND ITS VERIFICATION*

Nissim FRANCEZ
Department of Computer Science, The Technion, Haifa 32000, Israel

Brent HAILPERN
IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, U.S.A.

Gadi TAUBENFELD
Department of Computer Science, The Technion, Haifa 32000, Israel

Communicated by M. Sintzoff
Received October 1983
Revised December 1984

Abstract. In this paper, we introduce a new abstraction mechanism, called a *script*, which hides the low-level details that implement *patterns of communication*. A script localizes the communication between a set of *roles* (formal processes), to which actual processes *enroll* to participate in the action of the script. The paper discusses the addition of scripts to the languages CSP and ADA, and to a shared-variable language with monitors. Proof rules are presented for proving partial correctness and freedom from deadlock in concurrent programs using scripts.

1. Introduction

Abstraction mechanisms have been widely recognized as useful programming tools and have been incorporated into modern programming languages [4, 11, 19, 20, 25, 27]. The main subjects of abstraction suggested so far are
- *control sequencing abstractions*, which hide sequences of elementary transfers of control, such as looping constructs, if statements, procedures, and exception handling statements,
- *data abstractions*, as manifested in abstract data types, which hide the concrete representation of abstract objects (for example, is a stack implemented by a linked list with pointers or by an array?), and
- *synchronization abstractions*, as manifested in monitors and the like, which hide details of low-level mechanisms for enforcing mutual exclusion.

* Earlier versions of parts of this paper were presented at the Second ACM Symposium on Principles of Distributed Computing, Montreal, August 1983 and at the FSE-TCS, Bangalore, India, December 1984. This paper was originally published by Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, in the NATO ASI Series Vol. F 13. Nissim Francez was a WTVS at IBM Research (Yorktown) while on a sabbatical leave from the Technion, Haifa, Israel.

Besides hiding low-level information, abstraction mechanisms also restrict the use of such information to patterns that are generally recognized as well structured. They also save programming effort by enabling a single description to be used many times.

Recently, another low-level mechanism has emerged as playing an important role as a programming tool, namely *inter-process communication.* Several modern programming languages [4, 11, 14, 19, 20, 25, 27] support multiprocessing and distributed processing in one way or another. Every such language has a construct to support inter-process communications; some of the constructs are slightly higher-level than others, but all can be considered low-level because they handle some primitive communication between two partners at a time.

The purpose of this paper is to introduce an abstraction mechanism, whose subject is *communications.* To our knowledge, no such abstraction has been proposed. Such a mechanism will

- allow the hiding of low-level details concerning sequencing of communications, choice of partners, and larger scale synchronization (involving more than just a pair of processes),
- restrict the patterns of communication, which are arbitrary in current languages, to well-structured patterns, and
- enable a single definition of frequently used patterns, for example various buffering regimes.

Even when such well-structured patterns are identified and added as primitive constructs (as happened in the case of looping), it is still useful to permit a user to define, in an application-oriented way, his own abstractions. A trend toward such well-designed patterns exists already, for example 'idioms' [12].

In designing our communication abstraction mechanism, which we call a *script,* we adhered to the following design goals and restrictions:

- The abstraction will be designed in a context of a *fixed network.* Thus, we shall not deal in this paper with dynamic concurrency, where processes are dynamically generated, destroyed, or reconfigured;
- The abstraction should be modular and the behavior of an instance of an abstraction should not depend on the context of use, except by a predefined interface mechanism. We chose 'parameters passing' for our interface;
- The abstraction should be modularly verifiable [10, 18];
- The abstraction mechanism is biased toward models of disjoint processes (that is, no shared variables). Communication is achieved by some message passing actions or remote procedure calls.

We would like to emphasize that some techniques, with other goals, concerned with abstracting communication exist in the literature. For example, a general module interface hides the specific communication primitive being used from the user: procedure call (either local or remote), coroutine, message transfer, and so on [15, 23]. These techniques, however, are interested in encapsulating a *single* communica-

tion, while we are interested in encapsulating *patterns of communication* involving many primitive communication actions and many participants.

The rest of the paper is organized as follows. In Section 2, we informally describe the structure of the suggested communication abstraction mechanism and we discuss several alternatives regarding possible semantics. Some example scripts are informally described. In Section 3, we present the examples in a Pascal-like syntax. In Section 4, one of the example scripts, broadcast, is developed in three host languages: CSP, ADA[1], and a shared-memory language with monitors. Section 5 presents proof rules for partial correctness of programs using scripts. Section 6 extends the proof system toward proving absence of deadlocks. Section 7 concludes with some discussion of future work.

2. Scripts

In this section we introduce the *script*, the suggested mechanism to abstract from the internal details of the implementation of patterns of communication. Basically, a *script* is a parameterized program section, to which processes *enroll* in order to participate. The guiding idea behind the concept of enrollment is that the execution of the role (in a given script instance) is a logical continuation of the enrolling process, in the same way that a sequential subroutine is a continuation of the execution of its caller. If each process is allocated to a different processor, the role should be executed by the same processor on which the main body of the enrolling process is executed. Thus, no changes in the underlying communication network are needed to execute a script.

2.1. The structure of a script

A script consists of the following components:

- *Roles:* these are formal process parameters, to which (actual) processes enroll. We shall discuss the enrollment below. We also permit *indexed families of roles* in analogy to such families of actual processes.
- *Data parameters:* these are ordinary formal parameters (as in ordinary procedures); however, they are associated with the roles. Thus, each group of formal data parameters is bound at enrollment time to the corresponding actual parameters supplied by the enrolling process.
- *Body:* this is a concurrent program section, where for each role there is a corresponding part of the body, a process. All the roles are considered to be concurrent. The body describes the details of sequences of basic communications among the various roles. Thus, the body specifies the scenario in which the roles take place.

[1] ADA is a registered trademark of the U.S. Department of Defense.

As a typical example, we may consider a script implementing a scenario of (software) *broacast.* In this scenario, there is a role of a *transmitter*, with which is associated a (value) data parameter, x, to be transmitted. There is also a group of *recipient roles*, with each is associated a (result) data parameter, which will be assigned the value of x after the appropriate communication. The externally observable behavior of the script is that the value of x is passed to all the recipients and assigned to their corresponding data parameters.

The body of the script could hide the various broadcast strategies:
- A star-like pattern where the transmitter communicates directly with each recipient, either in some pre-specified order, or non-deterministically;
- A spanning tree, generating a wave of transmissions, where every role, on receiving x from its parent role, transmits it to every one of its descendant roles (again with different orderings);
- Others; see [24, 26] for a discussion of various broadcast patterns and their relative merits.

Sections 3 and 4 show how a broadcast script could be coded in our target languages.

One immediate question raised by considering the broadcast example is "how generic should a script be?" Should the type of x, in the broadcast script, be allowed to vary, or should a different script be needed to broadcast an integer, a stack, and so on? We shall not commit ourselves to a definite answer to these questions. Rather, we use the principle that a script is as generic as its host programming language allows. In a language that admits other forms of generic constructs, such as ADA, we could allow the script to contain the same.

Our second example is a replicated and distributed data base *lock manager* script. Consider n nodes in a network, each of which can hold a copy of a database. At any one time k nodes hold copies. The membership of this set of *active* nodes may change, but it always has k members. Readers and writers attempt to interact with this database through a lock manager script. The roles in this script consist of the k lock managers, a reader (possibly an indexed family of readers), and/or a writer. This script can hide various read/write locking strategies:
- Lock one node to read, all nodes to write;
- Lock a majority of nodes to read or write;
- Multiple granularity locking as described in [16].

The third example introduces a script that causes the rotation of data values around a ring of processes. The script hides the direction of rotation, the number of rotation steps, and the details of the rotation handling. We show in this example how different enrollment patterns to the same script achieve different effects.

Our next example is the remote server facility of the Accent operating system [22]. Accent is a communication-oriented operating system for a collection of processors connected by a network. The operating system attempts to make the location of resources in the system transparent to the user. All processes communicate

through local ports, with network servers to perform the actual communication across the network. We will model this port communication facility.

Our final example is a *recursive* script implementing the winning strategy for the *Tower of Hanoi* game. This example also shows the use of *nested enrollments*, where an enrollment to one script occurs within the body of another script.

2.2. Script enrollment

We next describe several possibilities for the semantics of enrollment of a process in an instance of a script.

Obviously, a process has to name the instance of the script in which it enrolls, and the name of the role it wishes to play. Furthermore, the process must supply actual data parameters for the formal parameters associated with that role. For the sake of verifiability, we assume that the data parameter passing modes are value, result, and value-result. The usual aliasing-preventing restrictions will be imposed.

We want to distiguish between two kinds of enrollment, based on the relationship between processes enrolling in the same (instance of a) script.

– *Partners-named enrollment.* A process not only names the role in which it enrolls, but also names the identities of (some or all of) the other processes it wants to communicate with in the script. Thus a process T may specify that it wishes to enroll in a broadcast script as a transmitter, while it wishes to see process P, Q, and R enroll as recipients in the same instance of the script. Similarly, process P might specify its enrollment in the broadcast script as a recipient with T as the transmitter. In such cases, the processes will jointly enroll in the script only when their enrollment specifications match, that is they all agree on the binding of processes to roles. It is also possible to have more elaborate naming conventions, for example by specifying that a given role should be fulfilled by either process A or process B. The partners-named enrollment generalizes the naming conventions of the host language for primitive communications, as in CSP's ! and ? or ADA's entry call.

– *Partners-unnamed enrollment.* Sometimes, a process does not care about, or does not need to know, the identities of its communication partners. In such a case, it will specify only its own role during enrollment; no matching is then needed for joint enrollment. In the broadcast example, a process T may wish to broadcast x to any process interested in receiving the value. Another reason for unnamed enrollment is that the host language may permit unnamed primitive communications: for example the accept statment in ADA, which accepts an entry call from any potential caller. A similar extension of the CSP primitives is proposed in [6].

Note that a mixture of the two enrollment regimes is also possible, where only a partial naming is supplied. In the broadcast example, P may specify the transmitter T, but not care about the other recipients.

If more than one process tries to enroll in the same role of the same instance of a script (each matching the naming conventions), then the choice of which process is actually enrolled is nondeterministic.

We impose the following stucture on enrollments, avoiding some ambiguities because of complex scoping: all processes enrolling to (the same instance of) a script must be roles in (an instance of) another script. As a result, the main program is regarded as a script. This restriction, together with another one imposed on basic communication within the bodies of the roles (described later) ensure that no process not enrolled in a script can affect the computations of that script.

2.3. Nested enrollments

If a group of roles in (an instance of) a script S_1 enroll into (an instance of) another script S_2, the corresponding part of the performance of S_1 is delayed until the performance of S_2 terminates.

In particular, in the recursive case, a new instance is opened upon a recursive enrollment. In direct recursion, all the roles of S must enroll (in some permutation) back into S. Mutual recursion is also permitted. Note that in direct recursion a role can recursively enroll into *any* role of the script. Thus, in a recursive enrollment, the new roles always form a permutation of the current roles.

2.4. Script initiation and termination

A basic question related to the execution of a script is "when will it start?" Again, two major possibilities can be distinguished.

– *Delayed initiation.* According to this method, processes must first enroll in *all* the roles of a given instance of a script; only then the execution of the script may begin. A process enrolled in a given role is delayed until all other partners are also enrolled. In the broadcast script, a delayed initiation will activate the script only after the transmitter and all recipients have enrolled. This method enforces global synchronization between large groups of processes (as a possible extension to CSP's synchronized communication between two processes). A consequence of this initiation strategy is that there is a one-to-one correspondence between (formal) roles and (actual) enrolling processes. No process may enroll in more than one role in one activation (of an instance) of a script.

– *Immediate initiation.* The script is activated upon the enrollment of its first participating process. Other processes may enroll while the script is in progress. A role is delayed only if it attempts to communicate with an unfilled role. This method may be easier to implement in existing host languages. Thus, in the broadcast example, after a transmitter has enrolled, each enrolling recipient may receive x independently of any other recipient having enrolled. This has a consequence that no role may assume that its script partner has sensed any effects of the script, unless it has communicated directly with that partner.

Similar considerations apply to the problem of terminating a script. A *delayed termination* will free (together) all the process enrolled in a script after all the roles are finished. An *immediate termination* will free each process when it completes its role. This distinction is crucial if script enrollment is to be allowed to act as a guard.

Note that immediate initiation combined with immediate termination allows a given process to enroll in several roles of the same script, where those roles do not communicate directly. With the combination of delayed initiation and delayed termination, the body of the script is treated as a closed concurrent block, similar to a **cobegin** ... || ... || ... **coend** construct. Delayed initiation is the more natural choice for partners-named enrollment. The kind of initiation and termination strategy used has consequences on the possibility of deadlock, as discussed in Section 7.

The declaration section of each script specifies the kind of initiation and termination strategies used in that script.

We call the collective activation of all the roles of an instance of a script a *performance.* If a performance has begun and some other process attempts to enroll in the script, a new instance of the script is invoked: a parallel performance begins.

We make no requirements about the fairness of script enrollments in the case of repeated attempts to enroll. We assume that the fairness properties are inherited from the host language. For example, in CSP no fairness is assumed. In ADA, repeated enrollments are serviced in order of arrival.

2.5. Critical role set

For a performance of a script, it may not be necessary that all roles of the script is filled. Different subsets of the roles could participate in different performances. For example, in the database example, it is sufficient that all the lock-manager roles be filled, as well as either the reader or the writer (or both). So that such partially-filled performances do not conflict with the initiation and termination strategies, we add the *critical role set* to the declaration of the script. It specifies the possible subsets of roles that will enable a performance to begin. Thus the initiation and termination policies are always considered as relative to the appropriate critical role sets. If no such set is specified, it is taken to mean that the entire collection of roles is critical.

For example, consider a script S with roles p, q, and r and delayed initiation. If the critical role sets are (p, q) OR (q, r), then should processes enroll in p and q, then the performance could begin, with no r. If instead the roles are filled in the order p, r, q, then all the roles would be filled before the performance begins.

The critical role sets creates a problem: when can a performance, which was initiated upon enrollment of a critical set, be terminated? In principle, the performance would wait, possibly forever, for the enrollment of processes to the roles *not* in the critical set. We therefore assume a *critical moment*, after which no further enrollments to the current performance are allowed. This moment succeeds, but does not necessarily coincide with, the moment all processes in the critical role set have enrolled. Termination will depend only on roles that have enrolled up to that critical moment. A role not fulfilled up to the critical moment is considered to have terminated.

The use of critical role sets introduces yet another problem: individual roles do not know which of their partner roles are participating in a particular performance. When some of the roles of a script are not filled, then attempts to communicate with the unfilled roles would block. Similarly, roles waiting to service requests from unfilled roles would never terminate.

There are many solutions to this problem, none of which are fully satisfying. If a centralized mechanism is controlling enrollments and performances, then it could inform the active roles of the names of the inactive roles.[2] Alternatively, attempting to communicate with an unfilled role could return a distinguished value. Our database example will follow the latter solution.

2.6. Inter-role communication

Communication among the various roles of a script is described using the inter-process communication primitives of the host language. Every communication between roles causes, at run time, a corresponding communication among the processes enrolled to the roles. In particular, the naming conventions of the host-languages apply to the roles: a role may name another role explicitly, or may communicate with an anonymous role in exactly the same way that actual processes do.

Note. We prohibit roles from communicating with any process other than the roles of the script. This restriction is intended to avoid the deadlock caused by a role trying to communicate with itself as the enrolling process.

3. Sample scripts

In this section we present several example scripts. Our language is Pascal-like with extensions for communication (synchronized *send* and *receive* with the same semantics as the ! and ? instructions of CSP) and non-deterministic guarded commands (**if** and **do**). Role parameters will be designated *IN* for value parameters, *OUT* for result parameters, and *IN OUT* for the value-result parameters. Comments will use the PL/1 and C convention of /**/. Sets are indicated by braces { }.

3.1. Synchronized star broadcast

Our first example provides for a simple extension of the synchronized send and receive in the host language; it is shown in Fig. 1. The broadcast script has one transmitter and five recipients; a more general example would use a general indexed family of recipients. The script is fully synchronized, because of the initiation and termination clauses. When all participants are enrolled, the data passed to the sender

[2] The notion of a central administrator for a script, however, does not preserve our goal of not generating additional processes when executing a script.

```
SCRIPT StarBroadcast;
  INITIATION: DELAYED;
  TERMINATION: DELAYED;

  ROLE transmitter (IN r: item);
    BEGIN
      SEND r TO recipient_1;
      SEND r TO recipient_2;
      SEND r TO recipient_3;
      SEND r TO recipient_4;
      SEND r TO recipient_5
    END transmitter;

  5
  ROLE recipient_i (OUT t: item);
  i=1

    BEGIN
      RECEIVE t FROM transmitter
    END recipient
END StarBroadcast
```

Fig. 1. Synchronized star broadcast.

is sent, in turn, to each of the recipients. All wait until the last copy is sent. Note that the sender is never blocked while waiting for a recipient, because all the recipients are available and not waiting for any other I/O operations. The notation

$$\overset{5}{\underset{i=1}{\text{ROLE}}}\ \textit{recipient}_i(\cdots)$$

is an abbreviation for five copies of the recipient role. Within the role, i is replaced by the actual index.

A process would enroll as the transmitter by

ENROLL IN *broadcast* AS *transmitter*(*expression*);

A process would enroll as the first recipient by

ENROLL IN *broadcast* AS *recipient*$_1$(*variable*);

3.2. Pipeline broadcast

Our second example, shown in Fig. 2, is similar to the first in form, but not in action. Here the sender gives the message to the first recipient and is then finished. The first recipient waits for the second recipient to arrive, passes the message along, and finishes, and so on. The immediate initiation and termination permit processes

```
SCRIPT PipeBroadcast;
  INITIATION: IMMEDIATE;
  TERMINATION: IMMEDIATE;
  CONST n = 5;

  ROLE transmitter(IN t: item);
    BEGIN
      SEND t TO recipient_1
    END transmitter;

  ROLE recipient_1(OUT r_1: item);
    BEGIN
      RECEIVE r_1 FROM transmitter;
      SEND r_1 TO recipient_2
    END recipient_1;

  ROLE (i=2..n-1) recipient_i(OUT r_i: item);
    BEGIN
      RECEIVE r_i FROM recipient_{i-1};
      SEND r_i TO recipient_{i+1}
    END recipient_{2...n-1};

  ROLE recipient_n(OUT r_n: item);
    BEGIN
      RECEIVE r_n FROM recipient_{n-1}
    END recipient_n
END PipeBroadcast
```

Fig. 2. Pipeline broadcast.

to spend much less time in the script, than in the previous example. However, this technique allows roles to block at send or receive operations if the neighboring role is not available.

3.3. Database

Our third example implements a distributed, replicated data base locking scheme. The script consists of k lock managers roles, one reader role, and one writer role. Each lock manager maintains a table of locks granted. Readers and writers can request or release lock on data items. Depending on the locking scheme, readers and writers may need permission from more than one lock manager to access a particular data item. Our example requires one lock to read, k locks to write. One

```
SCRIPT lock;
  INITIATION: IMMEDIATE;
  TERMINATION: IMMEDIATE;
  CRITICAL ROLES: (manager_{1...k}, reader) OR (manager_{1...k}, writer);
  ROLE manager_i (IN OUT LockTable: LockType);  i = 1..k
    BEGIN
      DO ¬(reader.terminated AND writer.terminated) →
        IF RECEIVE release(data, id) FROM reader →
           LockTable.ReadUnlock(data, id)
        □ RECEIVE release(data, id) FROM writer →
           LockTable.WriteUnlock(data, id)
        □ RECEIVE lock(data, id) FROM reader →
           IF LockTable.AbleToRead(data) →
              LockTable.ReadLock(data, id);
              SEND granted TO reader
           □ ¬LockTable.AbleToRead(data) →
              SEND denied TO reader
           FI
        □ RECEIVE lock(data, id) FROM writer →
           IF LockTable.AbleToWrite(data) →
              LockTable.WriteLock(data, id);
              SEND granted TO writer
           □ ¬ LockTable.AbleToWrite(data) →
              SEND denied TO writer
           FI
        FI
      OD
    END manager;
```

Fig. 3. Database lock manager.

performance of this script would result in either a reader or a writer (or both) attempting to lock or release a data item.

Between performances of the script the identity of the lock managers may change, but we assume that the lock tables are preserved by such a change (so that, for example, if a reader is granted a read lock in one performance, some lock manager will have a record of that lock on a subsequent performance). There would be a separate script for lock managers to negotiate the entering and leaving of the active set. The database example is shown in Fig. 3 through Fig. 5.

```
ROLE reader (IN id : ProcessId; IN data : object; IN request : (lock, release);
    OUT status : (granted, denied));
  VAR
    done : ARRAY [1 ... k] OF boolean;
  BEGIN
    done := false; /*array assignment*/
    IF request = release →
        DO □_{i=1}^{k} ¬done[i]; SEND release(data, id) TO manager_i →
          done[i] := true
        OD
    □ request = lock →
        status := denied;
        DO □_{i=1}^{k} (status ≠ granted) ∧ ¬done[i];
          SEND lock(data,id) TO manager_i →
          RECEIVE status FROM manager_i;
            done[i] := true
        OD
    FI
  END reader;
```

Fig. 4. Database lock manager (continued).

We assume that the lock tables are abstract data types with the appropriate functions to lock and release entries in the table and to check whether read or write locks on a piece of data may be added. We also assume that each processor, when enrolling provides its unique processor identifier, so that locks may be identified unambiguously.

In Section 2 we discussed critical role sets and the termination problem. In this example we have made available the function *r.terminated*, which returns true if role *r* has terminated or if the role *r* will not be filled. Before the critical role set is filled, *r.terminated* is false for all unfilled roles. Once the critical set is filled, all unfilled roles have *r.terminated* set to true.[3]

3.4. Accent port communication facility

Our final example models the Accent port communication facility [22] as specified and verified in [10]. Accent is a communication-oriented operating system for a

[3] We make no claim that this termination function would be simple to implement without a central administrator for the script.

```
ROLE writer (IN id: ProcessId; IN data: object; IN request: (lock, release);
    OUT status: (granted, denied));
  VAR
    done: ARRAY [1 ... k] OF boolean;
    who: SET OF [1 ... k];
  BEGIN
    done := false; /*array assignment*/
    IF request = release →
        DO □ (i=1..k) ¬done[i]; SEND release(data, id) TO manager_i →
          done[i] := true
        OD
    □ request = lock →
        who := ∅;
        DO □ (i=1..k) ¬done[i]; SEND lock(data, id) TO manager_i →
          RECEIVE status FROM manager_i;
          done[i] := true;
          IF status = granted → who := who ∪ {i}
          □ status = denied → done := true /*one denial implies failure*/
          FI
        OD;
        IF status = denied →
            DO □ (i=1..k)
              i ∈ who; SEND release(data,id) TO manager_i → who := who − {i}
            OD
        □ status = granted → SKIP
        FI
    FI
  END writer
END lock
```

Fig. 5. Database lock manager (continued).

collection of processors connected by a network. Many processes can exist at each node (processor). Three goals of Accent are

(1) the location of resources in the distributed system should be transparent,

(2) it should be possible for any feature provided by the operating system kernel to be provided instead by a process, and

(3) all services, except the basic communication primitives, should appear to processes as being provided through a message-passing interface.

Processes communicate through *ports*. Associated with each port is a *queue* of messages sent to that port, but not yet received. Only one process at a time can have received access to a given port, though many processes can have send access to it. We do not deal here with process or port creation or destruction. We assume the existence of processes and ports. We also assume there is a static binding between process and ports. We will implement ports as scripts.

Because messages are sent to ports, rather than to processes, intermediate processes can be used to manage communication between distinct process groups. A prime example is a network server: if process A runs on node X and process B runs on node Y, the network server N can provide mirror ports in X and Y so that A and B can communicate. Consider the situation of A sending a message to B. The network server N on X has an alias port B_N also on X. Process A believes that B_N belongs to B, but in fact it belongs to N. Messages sent to B_N are read by N and forwarded to the actual input port of B on Y.

The example specified in [10] deals with a distributed virtual memory system. That is, paging can be done across the network. We will restrict our discussion to the port and network server portions of this system.

A port is a FIFO buffer that accepts transmissions from any number of senders, but sends them onto a single receiver. Hence, two-way communication requires two ports. We model the FIFO buffer portion of the port as a queue process. The communication portion of the port is implemented as a script. This port script has three roles: sender, buffer, and receiver. We associate the identity of the port with the queue process. That is to select a port, the sender or receiver names the queue process in a partners-named enrollment. Simultaneous non-interfering communications are allowed through parallel performances.

A (generic) queue process, contains a FIFO data structure (assumed to be a primitive data type of the language). The process repeatedly enrolls in the port script, in case its sender or receiver are attempting to speak with it. Optionally, it could name the receiver process in its enrollment. For the sake of simplicity, we will leave the naming of the queue to the sender and receiver processes. The body of the queue process is

```
TYPE PROCESS QUEUE:
  VAR q : FIFO_BUFFER OF BaseType;
  BEGIN
    q := empty;
    WHILE true DO
      ENROLL IN PortScript AS buffer(q);
  END;
```

The port script allows enrollment in pairs: sender and buffer, receiver and buffer. It must allow also for the case that all three roles are filled. As in the database example, we use the *terminated* attribute of a role name to determine if that role is

```
SCRIPT PortScript;
  INITIATION:IMMEDIATE;
  TERMINATION:IMMEDIATE;
  CRITICAL ROLES (sender, buffer) OR (receiver, buffer);

  ROLE sender(IN d:BaseType; OUT success:boolean);
    BEGIN
      SEND d TO buffer;
      RECEIVE success FROM buffer
    END sender;

  ROLE receiver(OUT d:BaseType; OUT success:boolean);
    BEGIN
      SEND 'fetch' TO buffer;
      RECEIVE (d, success) FROM buffer
    END receiver;

  ROLE buffer(IN OUT q:FIFO_BUFFER OF BaseType);
    VAR d:BaseType;
    BEGIN
      DO ¬sender.terminated; RECEIVE d FROM sender→
          IF q.full→SEND false /*failed*/ TO sender
          □ ¬q.full→
              q.enque(d);
              SEND true /*ok*/ TO sender
          FI
      □ ¬receiver.terminated; RECEIVE 'fetch' FROM receiver→
          IF q.empty→SEND (empty, false) /*failed*/ TO receiver
          □ ¬q.empty→
              q.deque(d);
              SEND (d,true) /*ok*/ TO receiver
          FI
      OD
    END buffer
END PortScript
```

Fig. 6. Accent port script.

inactive. The port script is shown in Fig. 6. It should be reasonably clear that the port script maintains the FIFO property of the underlying buffer data type and that the script will not deadlock on empty or full buffers. The most important feature of the port script, from the Accent point of view, is that the identities of the sender and receiver are hidden from each other.

The implementation of the net server is now straightforward. If we assume one net server per remote node, then each server scans through the ports it maintains. If a message appears it appends the true location of the server (at the remote node) and passes that message to the network interface (through a port). If the net server detects a message from the network interface, it strips off the destination address and puts the message in the appropriate user port. If we are permitted only one net server on each node, then it must also maintain a table corresponding to its input ports, of the destination node *id* of the remote server.

4. Sample script in CSP, ADA, and with monitors

In this section, we describe how scripts could be added to existing programming languages. The rules and example given are intended to be existence proofs that such additions could be made without extending the base language in any way. As a result, not every language supports all features. Ideally, scripts would be added as an integral part of the base languages; these scripts would support all the options and features described above. In each of the examples of this section we extend the syntax of the native language to include scripts. These extensions include script declarations, role declarations, and enrollment statements. The syntax of these extensions were intended to conform to the normal syntax of the language.

4.1. Scripts in CSP

CSP [14] imposes strict naming conventions, where in every communication both parties explicitly name each other. We, therefore, adopt a restricted named-enrollment policy: each process, besides naming the role to which it enrolls, names the processes for all other roles in the script with which the role will directly communicate. All inter-role communication will also use explicit role naming.

The initiation policy will be immediate initiation, because CSP cannot synchronize more than two processes at a time. Similarly, the termination policy will be immediate. We will use the ability of CSP to define named arrays of processes that know their indices, to 'implement' arrays of roles. We take some notational liberties considering whole-array assignments. Figure 7 shows a broadcast script in CSP.

Now consider a parallel command $[\ldots \| p \| \ldots \| q \| \ldots]$, where p contains an enrollment of the form

ENROLL IN *broadcast* AS *transmitter*(*exp*) WITH
[*qa* AS $recipient_1$, *qb* AS $recipient_2$, *q* AS $recipient_3$,
qd AS $recipient_4$, *qe* AS $recipient_5$]; ...

Here *qa*, *qb*, *qd*, and *qe* are other process names in the same concurrent command.

In process *q*, we will have the following enrollment

ENROLL IN *broadcast* AS $recipient_3(u)$
WITH *p* AS *transmitter*; ...

SCRIPT *broadcast*::
INITIATION: IMMEDIATE;
TERMINATION: IMMEDIATE;
[ROLE *transmitter*(*x*: *item*):: VAR *sent*: ARRAY[1..5] OF *boolean* := 5**false*;
$*[\Box_{k=1}^{5} \neg sent[k]; recipient_k!x \rightarrow sent[k] := true]$
||
ROLE $recipient_i(y{:}item)$:: *transmitter*?*y*
$_{i=1}^{5}$
].

Fig. 7. Broadcast in CSP.

The use of arrays of roles here is rather strict: a process always enrolls to a specific role in an array. A suggestive idea is to allow the en block enrollment of an array of processes to an array of roles. The explicit, strict naming conventions make it difficult to hide details of communication. For example, if the body of the broadcast script were to be implemented as a pipeline, where $recipient_i$ $(1 < i < 5)$ receives the value of x from $recipient_{i-1}$ and transfers it to $recipient_{i+1}$ and $recipient_1$ receives x from the transmitter, then the enrollment would have to be different.

4.2. Translation into CSP

We now show that scripts with the restrictions mentioned above, do not transcend the direct expressive power of CSP. Since CSP is not explicit about local (intraprocess) procedures, we use an in-line translation. To avoid unintended matching between communication commands arising from the translation, we shall use unique, new message tags, which are assumed not to occur anywhere in the original program. Because CSP does not have instances of processes, we cannot implement parallel performances, instead we cause each performance of a script to execute separately. We therefore associate with each script s another process p_s, which will coordinate enrollments to s. Since this translation is only for the sake of proving expressibility in CSP, the centralized nature of the resulting implementation does not imply that the actual implementation needs to be centralized. One of the major directions of future research is to discover distributed algorithms to achieve such multiple synchronization based on a generalization of the current distributed algorithms for binary handshaking.

Consider $P = [p_1 \| \ldots \| p_n]$ and a script s, with roles $r_1, \ldots, r_m$.

Rules of translation: Replace every enrollment within a process p_i of the form

ENROLL IN s AS r(*params*) WITH $[p_{i_1}$ AS $r_1; \ldots p_{i_m}$ AS $r_m]$

```
p_s:: ready: ARRAY[1 ... m] of boolean := m * true;
  done: ARRAY[1 ... m] of boolean := m * false;
    *[ □(k=1..m) □(j=1..n)
        [ready[k];p_j?start_s( ) → ready[k] := false
        □
        ¬ready[k];p_j? end_s( ) → done[k] := true
        ]
      □ ∧(k=1..m) done[k] → ready := m*true;  done := m*false
    ].
```

Fig. 8. CSP script supervisor.

by the following:

(1) An output command $p\_s!start\_s( )$;

(2) The body of role r (in script s) with

(a) each role name r_j replaced by process name p_{i_j} according to the correspondence specified in the enrollment,

(b) the actual parameters, *params*, substituted for the formal script data parameters (as in call-by-value-result semantics),

(c) every communication command tagged with the script name, for example, $r_1!(x+y)$ becomes $p_{i_1}!s(x+y)$ and $r_2?u$ becomes $p_{i_2}?s(u)$;

(3) An output command $p\_s!end\_s( )$.

The process p_s will be concurrently composed with the enrolling processes, and is defined in Fig. 8. Note that the script supervisor p_s must address all other processes, since every process is a potential enroller to every role. This is another example of the usefulness of the extended naming conventions described in [6].

We defer discussion of enrollments with unspecified parties to the next section. It describes the incorporation of scripts in ADA, where such enrollments fit more naturally.

4.3. *Scripts in* ADA

One feature that distinguishes CSP from ADA is that ADA supports server tasks that need not know the names of the processes that call them, whereas in CSP each process must know the name of every process with which it communicates. We extend this notion of a server task to a server script, that is a script with a partners-unnamed enrollment policy. Of course, the partners-named policy could be accomplished in ADA as in CSP, using local procedures to represent the roles and a supervising task to coordinate entries.

Figure 9 shows a broadcast script in ADA. The script consists of six roles: a sender and five recipients. The recipients all share the same code, so a template

```
SCRIPT broadcast IS
  INITIATION:IMMEDIATE;
  TERMINATION:IMMEDIATE;
  ROLE sender(data:IN item);
  ROLE TYPE recipient(data: OUT item);
  r1, r2, r3, r4, r5:recipient;
END SCRIPT:

SCRIPT BODY broadcast IS

  ROLE sender(data: IN item) IS
    ENTRY receive(d:OUT item);
    completed:integer:=0;
    BEGIN
      WHILE completed<5 LOOP
        ACCEPT receive(d:OUT item) DO
          d:=data;
        END;
        completed:=completed+1;
      END LOOP;
    END sender;

  ROLE recipient(data:OUT item) IS
    BEGIN
      sender.receive(data);
END broadcast;

TASK s IS...ENROLL IN broadcast AS sender(expression);...
  END s;

TASK r IS...ENROLL IN broadcast AS r1(variable);...END r;
```

Fig. 9. Broadcast in ADA.

(role type) is used. Note that the script body contains a 'reverse broadcast' because the recipients call the transmitter, rather than the other way around. This is a result of ADA's naming conventions: calls to a task must name that task. But receptions of calls (entries) do not name the calling task. In addition, selections between alternative entries are allowed, but not selections between alternative calls. See [7] for the problems caused by the absence of such selections.

4.4. Translation into ADA

We now show how a subset of scripts can be added to ADA with the following translation to ADA (without scripts). Each role becomes a task and one additional

```
TASK s_supervisor IS
  ENTRY start(1 ... m);
  ENTRY stop(1 ... m);
END s_supervisor;

TASK BODY s_supervisor IS
  ready: ARRAY (1 ... m) OF boolean := (1 ... m => true);
  done: ARRAY (1 ... m) OF boolean := (1 ... m => false);
  all_done: ARRAY (1 ... m) of boolean := (1 ... m => true);
  BEGIN
    LOOP
      SELECT
      WHEN ready(1) => ACCEPT start(1) DO ready(1) := false; END;
      OR ...
      OR WHEN ready(m) => ACCEPT start(m) DO ready(m) := false; END;
      OR WHEN ¬done(1) => ACCEPT stop(1) DO done(1) := true; END;
      OR ...
      OR WHEN ¬done(m) => ACCEPT stop(m) DO done(m) := true; END;
      OR WHEN done = all_done =>
        done := (1 ... m => false);
        ready := (1 ... m => true);
      END SELECT;
    END LOOP:
  END s_supervisor;
```

Fig. 10. ADA script supervisor.

task is created to coordinate the enrollments. Because each role is representd by a task, the other roles can know its name. Each role is given a number, which it uses to call the start and stop (family of) entries of the supervisor. Figure 10 gives the general form of the supervisor, for a script s, where m is the number of roles in the script. We assume that the 'macro expansion' prevents ADA tasks from calling any task of the script except through enrollment. This task per role translation is similar to the procedure per role translation to provide ADA procedure variables in [17].

Consider processes $p_1, \ldots, p_n$ and script instance s, with roles $r_1, \ldots, r_m$.

Rules of translation:

(1) Replace every enrollment with a process p of the form

ENROLL IN s AS r(*in-param, out-param, inout-param*);

by the following:

s_r.start(*in-param, inout-param*);
s_r.stop(*out-param, inout-param*);

```
ROLE r_i(v1:IN t1; v2:OUT t2; v3:IN OUT t3) IS
  ENTRY e(parameter_list); ...  —entries to be called by other roles
  v4: t4 := value4; ...  —local variables
  BEGIN ...
    ACCEPT e(b,c) DO ... END;  —entry
      r_j.x(y,z); ...  —call to entry in another role
  END r_i;
```

ADA Role (before translation)

```
LOOP
  v4 := value4;  —initialize local variables
  s_supervisor.start(i);  —synchronize with supervisor
  ACCEPT start(v1:IN t1; v3:IN t3) DO  —synchronize with
    v1' := v1;      —enrolling task
    v3' := v3;
  END;
  B;
  ACCEPT stop(v2:OUT t2; v3:OUT t3) DO  —synchronize with
    v2 := v2';  —enrolling task
    v3 := v3';
  END;
  s_supervisor(i).stop;  —synchronize with supervisor
END LOOP;
```

ADA Role (after translation)

Fig. 11. ADA Role—Before and after.

(2) Replace each role r_i of script s by a task s_r_i.
  (a) The role r_i has the form shown in the top half of Fig. 11;
  (b) Task s_r_i has all the entries of r_i plus two additional entries

```
ENTRY start(v1 : IN t1; v3 : IN t3);
ENTRY stop(v2 : OUT t2; v3 : OUT t3);
```

  (c) Task s_r_i has all the local variables of r_i, without initialization, and one new local variable, $v1'$, $v2'$, $v3'$, for each formal parameter of the start/stop entry calls, $v1$, $v2$, $v3$.

(3) Let B bet the body of r_i.
  (a) The body of s_r_i is shown in the bottom half of Fig. 11;
  (b) In the body B, occurrences of $v1$, $v2$, $v3$ are replaced by $v1'$, $v2'$, $v3'$;
  (c) Calls to role entry $r_j.x(y, z)$ becomes calls to task entry $s\_r_j.x(y, z)$;
  (d) Accept statements of the body undergo no special change.

This translation has two unfortunate consequences. First, the number of processes grows from n (in the script) to $n+m+1$ in the translation; this growth makes it difficult to associate the execution of a role with the same processor that enrolls in the script. Second, the translation can convert a terminating program into a non-terminating one, because of the infinite loops in the role tasks. A realistic implementation would also require non-centralized coordination of roles, as mentioned in the section on CSP.

4.5. Scripts with monitors

Monitors can serve two purposes: encapsulation (abstraction) of information and mutual exclusion. Using monitors for data abstraction may lead to unnecessary restrictions on concurrency. Combining scripts and monitors allows the programmer to have the advantages of abstraction, without sacrificing all concurrency to the single-thread control of the monitor.

Consider a broadcast with mailboxes for each recipient. There are two monitor implementations of this scheme: the first uses a single monitor to house all the mailboxes, the second uses one monitor per mailbox. The first implementation is a unified abstraction, all details hidden in a single black box, but all access to any mailbox is serialized. The second implementation eliminates the unnecessary concurrency restrictions, but the components of the broadcast are no longer packaged together. Our script solution follows the multiple monitor scheme, but with the script providing the top-level packaging. The monitor implementation of a star broadcast, similar to Fig. 2 is shown in Fig. 12. Note that in this implementation, we assume that the critical role set includes the sender and all five recipients; this prevents the sender from waiting on a full mailbox. A monitor-based supervisor would most easily implement immediate initiation and termination. No translation rules are given, as they would be similar to those for ADA and CSP.

5. Proof rules for partial correctness of scripts

In this section we present a more formal definition of the script concept. We define *proof rules* for proving partial correctness assertions about concurrent programs using scripts. There are two main aspects of the script that dictate an approach toward the formulation of the required rules.

(1) The script, viewed as an abstraction, is a multi-party communication and synchronization construct. It generalizes the primitives found in most concurrent languages that involve *binary* communication and synchronization.

(2) The (joint) script enrollment of processes to roles in a script can be viewed as a generalization of the procedure-call mechanism. In the script case, 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 achieved through parameter passing.

```
SCRIPT broadcast;
  TYPE mailbox: MONITOR
    VAR contents: item;
      status: (full, empty);
    PUBLIC PROCEDURE put(i: item);
      BEGIN
        WAIT UNTIL status = empty;
        contents := i;
        status := full;
      END put;
    PUBLIC FUNCTION get: item;
      BEGIN
        WAIT UNTIL status = full;
        get := i;
        status := empty;
      END get;
    BEGIN
      status := empty;
    END mailbox;

  ROLE sender(data: item);
  VAR k: integer;
    BEGIN
      FOR k := 1 TO 5 DO recipient_k.mbox.put(data);
    END sender;

    5
  ROLE recipient_i(VAR data: item);
   i=1
    VAR PUBLIC mbox: mailbox;
        BEGIN
          mbox.get(data);
        END recipient;
END broadcast;
```

Fig. 12. Mailbox broadcast.

The task is to find a proper amalgam of proof rules, dealing with concurrency, communication, and 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 is known as *cooperation proofs*. We generalize both the sequential proof rules for a process (role), to deal with enrollment, and the notion of cooperation, to deal with 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.

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 enrolling environment by a generalization of a rule for procedure calls.

This section consists of two parts. The first part presents the verification ideas in a way that is independent of the host language. In the second part, we assume that CSP is the host language, and consider an augmentation of the proof system presented in [1] to our needs. CSP was chosen because of its natural suitability for our context, the availability of established proof system for it, and our familiarity with both. We devote a small discussion to adapting the ideas to a subset of ADA that deals with concurrency, for which cooperating proofs also exist. Nowhere is the dependency on the host language essential.

Because of the similarity between our proof sysem and that of CSP, we will use a more CSP-like notation for scripts, for the rest of the paper. In particular, note that roles are treated syntactically as processes; they are prefixed with *name::* and are separated by ||.

In this section we assume that the actual parameters, transferred by an actual process to a role, are expressions referring to distinct identifiers, thereby avoiding aliasing. We will not treat the case where both initiation and termination are immediate. We do not assume CSP's convention for distributed termination of loops. 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 communicate only by 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 discussed at the end of the section.

5.1. 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 in [2, 9, 13]. For each body of a script some assertion, relating pre- and post-conditions, is proved. Using these script assertions, an assertion about the main program is proved.

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

With each script we associate an invariant SI called the *script invariant.* Each SI expresses global information about its script. A script invariant may refer to the formal parameters and local variable 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 a proof system for the host

language. When it uses enroll commands (that is, there are nested enrollments) the system described below is used.

The procedure inference rule [13] is used as the interface between the procedure call and its body. Similarly, we present a new proof rule for scripts that is a generalization of the procedure rule.

The notation

$$\text{ROLE } r_j(\text{IN } \vec{x}_j;\ \text{IN OUT } \vec{y}_j;\ \text{OUT } \vec{z}_j)::B_j$$

defines a role r_j with value (IN) parameters $\vec{x}_j$, value-result (IN OUT) parameters $\vec{y}_j$, result (OUT) parameters $\vec{z}_j$, and body B_j. For a script s with roles as defined above, we use the notation

$$\text{SCRIPT } s(\vec{x},\ \vec{y},\ \vec{z})::B_s$$

to define a script. Here $\vec{x}$, $\vec{y}$, $\vec{z}$ denote the formal parameters of the roles $\vec{x}_1, \ldots, \vec{x}_{ns}$; $\vec{y}_1, \ldots, \vec{y}_{ns}$; $\vec{z}_1, \ldots, \vec{z}_{ns}$ respectively, where $ns = |s|$ denotes the number of roles in the script s. Also, B_s denotes the script body $(\|_{j=1}^{ns} B_j)$.

As mentioned above, as assertion

$$\{pre(s)\}\ B_s\ \{post(s)\}$$

can be associated with any given script s. Both $pre(s)$ and $post(s)$ 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 $pre(s)$ and $post(s)$ may only be $\vec{x}$, $\vec{y}$ and $\vec{y}$, $\vec{z}$, respectively. The predicates may also refer to constants and *free variables* to describe initial and final values (called *logical variables* in [9]). Note that $\vec{z}$ must be initialized inside B_s, which explains why $pre(s)$ may not refer to the result parameters. After termination of a performance, the value parameters, $\vec{x}$, have 'returned' to their initial state. Hence, they can not affect the final values of the script. Therefore, $post(s)$ may not refer to the value parameters. Note that the initial value of the value parameters can be accessed by $post(s)$ through free variables. These restrictions are motivated similarly to the analogous restrictions regarding procedures and do not restrict generality.

When applying the proof system presented in [1] (summarized in an appendix) to a script s, which uses CSP's primitive communication commands, the script roles and the predicate $pre(s)$ correspond, respectively, to the processes and a precondition over the initial state in CSP programs. Consider again the broadcast example with only two recipient roles. Using the proof system for CSP described in [1], we may prove

$$\{x_1 = C\}\ B_{\text{broadcast}}\ \{z_2 = z_3 = C\}.$$

See Fig. 13 for a proof outline. The free variable C *freezes* the initial value of the transmitter and final values of all the roles. Because $\{x_1 = C\}\ B_{\text{broadcast}}\ \{z_2 = z_3 = C\}$

[ROLE r_1::{$x_1 = C$} *sent*[2 . . . 3] := *false*;
LI: {$x_1 = C$}
$*[\overset{3}{\underset{k=2}{\square}} \neg sent[k];\ r_k!x_1 \rightarrow sent[k] := true$ {LI}
{LI}
‖
$\overset{3}{\underset{i=2}{\text{ROLE}}}\ r_i$::{*true*} $r_1?z_i$ {$z_i = C$}
]
In this case, SI ≡ *true*.
For estabishing cooperation we have to prove (for $k = i$):
{$x_1 = C$} $r_k!x_1 \| r_1?z_i$ {$x_1 = C \wedge Z_i = C$}
which is done by applying
- communication and preservation axioms,
- conjunction, parallel composition, and consequence rules.

Fig. 13. Broadcast proof.

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 $E_j^s(\vec{a}_i, \vec{b}_i, \vec{c}_i)$, where the variables $\vec{a}_i$, $\vec{b}_i$, and $\vec{c}_i$, are the arguments corresponding to the parameters $\vec{x}_j$, $\vec{y}_j$, and $\vec{z}_j$, respectively. The value arguments $\vec{a}_i$ can be expressions. The notation E_j^s is shorthand for ENROLL IN s AS r_j.

We define $E_1^s, \ldots, E_{ns}^s$ to be (*syntactically*) *matching enrollments*. By the assumption that initiation and termination are not both immediate, no two $E_i^s, E_j^s, i \neq j$ belong to the same process. This notion is a natural generalization of matching communication commands, used in verifying CSP programs [1]. Recall that by the restriction of enrollments in the script definition, matching enrollments consist only of enroll commands that 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. This rule naturally generalizes the *procedure rule* [2, 9, 13].

Enrollment Rule. For a script s and matching enrollments $E_1^s, \ldots, E_{ns}^s$,

$$\frac{\{pre(s)\}\ B_s\ \{post(s)\}}{\{pre(s)[\vec{a}; \vec{b}/\vec{x}; \vec{y}]\} \left[\overset{ns}{\underset{j=1}{\|}} E_j^s(\vec{a}_{k_j}, \vec{b}_{k_j}, \vec{c}_{k_j})\right] \{post(s)[\vec{b}; \vec{c}/\vec{y}; \vec{z}]\}}$$

where $\vec{a}$, $\vec{b}$, $\vec{c}$ denote $(\vec{a}_{k_1}, \ldots, \vec{a}_{k_{ns}})$, $(\vec{b}_{k_1}, \ldots, \vec{b}_{k_{ns}})$, $(\vec{c}_{k_1}, \ldots, \vec{c}_{k_{ns}})$, respectively. By definition all the processes P_{k_j} $(k_j = 1, \ldots, n)$ and the roles r_j $(j = 1, \ldots, ns)$ are

disjoint. Here $p[\vec{u}/\vec{v}]$ denotes the assertion obtained from p by substituting (simultaneously) $\vec{u}$ for all free occurrences of $\vec{v}$.

In other words, the script s operates on the actual parameters $\vec{a}$; $\vec{b}$; $\vec{c}$ in exactly the same way as the body B_s would do with the formal parameters $\vec{x}$; $\vec{y}$; $\vec{z}$. Thus it is expected that $post(s)[\vec{b}; \vec{c}/\vec{y}; \vec{z}]$ is true after execution of the script if $pre(s)[\vec{a}; \vec{b}/\vec{x}; \vec{y}]$ 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 (that is, parameter passing does not affect the invariance of SI).

As an example consider a program $P::[P_1 \| P_2 \| P_3]$ using the broadcast script specified above, where $P_1::E_1(5)$; $P_2::E_2(c_2)$; $P_S::E_3(c_3)$. Let E abbreviate $E^{\text{broadcast}}$. We can prove

$$\{true\}\,[P_1 \| P_2 \| P_3]\,\{c_2 = c_3 = 5\}.$$

Using the proof that

$$\{x_1 = C\}\,B_{\text{broadcast}}\,\{z_2 = z_3 = C\}$$

which was given before. We take C to be 5 and get

$$\{x_1 = 5\}\,B_{\text{broadcast}}\,\{z_2 = z_3 = 5\}.$$

By the enrollment rule we get

$$\frac{\{x_1 = 5\}\,B_{\text{broadcast}}\,\{z_2 = z_3 = 5\}}{\{x_1 = 5[5/x_1]\}\,[E_1(5) \| E_2(c_2) \| E_3(c_3)]\,\{z_2 = z_3 = 5[c_2, c_3/z_2, z_3]\}}.$$

After substitution we obtain

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

which completes the proof.

Note that, like the procedure-call rule [9], 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 examine 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. The actual state satisfies the script invariant (after substitution), so that the usual inductive argument can be applied to deduce the invariant upon total termination.

We need not, however, require synchronization at both initiation and termination. It suffices that at least one event, either initiation or termination, be delayed (synchronized). The other one may be immediate. The argument for showing this is a variation on the one used in [5], as each performance of a script under such conditions satisfies similar properties to those 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 [5] for further discussions.

The restrictions we have presented induce a pattern of execution: processes do local activities until all face enrollments, then, a whole group, forming a matching enrollment, advances in one 'big step'. This generalizes the execution of CSP programs induced by the [1] 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 [3].

Next we add an inference rule to deal with recursive scripts. it is a natural generalization of the rule for recursive procedures [13, 2]. Consider a (recursive) script declaration

$$\text{SCRIPT } s(\vec{x}, \vec{y}, \vec{z})::B_s,$$

where B_s may include recursive enrollments. The rule refers to recursive script s and matching enrollments $E^s_1, \ldots, E^s_{ns}$.

Recursion Rule.

$$\frac{\{pre(s)\}\left[\|_{j=1}^{ns} E^s_j(\vec{x}_j, \vec{y}_j, \vec{z}_j)\right]\{post(s)\} \vdash \{pre(s)\}\, B_s\, \{post(s)\}}{\{pre(s)\}\left[\|_{j=1}^{ns} E^s_j(\vec{x}_j, \vec{y}_j, \vec{z}_j)\right]\{post(s)\}}.$$

That is, we infer

$$\{pre(s)\}\left[\|_{j=1}^{ns} E^s_j(\vec{x}_j, \vec{y}_j, \vec{z}_j)\right]\{post(s)\}$$

from the fact that $\{pre(s)\}\, B_s\, \{post(s)\}$ can be proved (using the other rules and asxioms) from the assumption

$$\{pre(s)\}\left[\|_{j=1}^{ns} E^s_j(\vec{x}_j, \vec{y}_j, \vec{z}_j)\right]\{post(s)\}.$$

This is the usual circularity encountered when treating recursion. The generalization to mutual recursion is clear.

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 [2]. Both of them refer to script s and matching enrollments $E^s_1, \ldots, E^s_{ns}$.

Parameter Substitution Rule.

$$\frac{\{p\}\left[\underset{j=1}{\overset{ns}{\|}} E_j^s(\vec{x}, \vec{y}, \vec{z})\right]\{q\}}{\{p[\vec{d}; \vec{e}/\vec{x}; \vec{y}]\}\left[\underset{j=1}{\overset{ns}{\|}} E_j^s(\vec{d}_{k_j}, \vec{e}_{k_j}, \vec{f}_{k_j})\right]\{q[\vec{e}; \vec{f}/\vec{y}; \vec{z}]\}}$$

where $var(\vec{d}; \vec{e}; \vec{f}) \cap free(p, q) \subseteq \{\vec{x}, \vec{y}, \vec{z}\}$.

$p[\vec{d}; \vec{e}/\vec{x}; \vec{y}]$ stands for simultaneous substitution of the expressions from $\vec{d}$ and $\vec{e}$ for the variables $\vec{x}$ and $\vec{y}$,

$var(\vec{d}; \vec{e}; \vec{f})$ denotes the set of all variables appearing in $\vec{d}$, $\vec{e}$, and $\vec{f}$.

$free(p, q)$ denotes the set of all free variables of p and q. A similar restriction appears and is explained in [2, p. 464].

Variable Substitution Rule.

$$\frac{\{p\}\left[\underset{j=1}{\overset{ns}{\|}} E_j^s(\vec{a}_{k_j}, \vec{b}_{k_j}, \vec{c}_{k_j})\right]\{q\}}{\{p[\vec{t}/\vec{r}]\}\left[\underset{j=1}{\overset{ns}{\|}} E_j^s(\vec{a}_{k_j}, \vec{b}_{k_j}, \vec{c}_{k_j})\right]\{q[\vec{t}/\vec{r}]\}}$$

where $var(\vec{t}; \vec{r}) \cap var(\vec{a}; \vec{b}; \vec{c}) = \emptyset$.

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 enroll command.

Both rules are necessary only when recursion is allowed. Examples using the rules appear below.

5.2. Proving properties of enrollments

We now introduce the method for proving pre- and post-assertions about a script that uses enroll commands. This proof system is structured similarly to the one for CSP introduced in [1].

We use the term *process* for both a role and an external process. 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 implies 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 for CSP's communication commands [1]. There the arbitrariness of q is explained in more detail.

Using the enrollment axiom and the first eight rules of inference (I1–I8), which are listed in the appendix, we can establish separate proofs for each process. This is presented, as in [21], by a *proof outline* in which each sub-statement of a process is preceded and followed by a corresponding assertion.

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 using a cooperation test.

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

(1) 'small guess'—as present in proof system for CSP in the form of a communication axiom [1]. The 'guess' is over the effect of a single communication.

(2) 'moderate guess'—as presented in the proof system for an ADA subset (for concurrency) using the call-accept primitives [8]. Here the 'guess' is over a chain of entry calls, when an *accept* or *call* appears within the body of another *accept*.

(3) 'big guess'—as present in our system, where we 'guess' the effect of an enrollment, which 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*. We define a process P_i to be *bracketed* if the brackets '⟨' and '⟩' are interspersed in its text so that

(1) for each program section ⟨B⟩, B is of the form B_1; E; B_1' where B_1 and B_1' do not contain any enroll commands, and

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

The purpose of the brackets 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 [1].

With each proof of $\{p\}\,[P_1 \| \ldots \| P_n]\,\{q\}$ we associate a script in variant SI and an appropriate bracketing. The proof rule concerning parallel composition has the following form:

Parallel Composition Rule.

$$\frac{\text{proofs of } \{p_i\}\ P_i\ \{q_i\},\ i = 1, \ldots, n, \text{ cooperate}}{\{p_1 \wedge \cdots \wedge p_n \wedge \mathrm{SI}\}\,[P_1 \| \ldots \| P_n]\,\{q_1 \wedge \cdots \wedge q_n \wedge \mathrm{SI}\}}$$

provided no variable free in SI 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 SI associated with it. We define $\langle B_1 \rangle, \ldots, \langle B_{ns} \rangle$ to be matching bracketed sections if they contain matching enrollment $E_1^s, \ldots, E_{ns}^s$ to some script s.

We further define the proofs $\{p_i\}\ P_i\ \{q_i\}$, $i = 1, \ldots, n$, to *cooperate* if

(1) the assertions used in the proof of $\{p_i\}\ P_i\ \{q_i\}$ have no free variables suject to change in P_j for $i \neq j$,

(2) the statement

$$\left(\bigwedge_{j=1}^{ns} pre(B_j) \wedge \mathrm{SI} \right) \left[\parallel_{j=1}^{ns} B_j \right] \left\{ \bigwedge_{j=1}^{ns} post(B_j) \wedge \mathrm{SI} \right\}$$

holds for all matching bracketed sections $\langle B_1 \rangle, \ldots, \langle B_{ns} \rangle$.

The following axiom and proof rules are needed to establish cooperation: enrollment axiom, enrollment rule, recursion rule, parameter substitution rule, and variable substitution rule as described above.

Rearrangement Rule.

$$\frac{\{p\}\ B_1; \ldots; B_{ns}\ \{p_1\},\ \{p_1\} \left[\parallel_{j=1}^{ns} E_j^s \right] \{p_2\},\ \{p_2\}\ B_1'; \ldots; B_{ns}'\ \{q\}}{\{p\} \left[\parallel_{j=1}^{ns} (B_j; E_j^s; B_j') \right] \{q\}}$$

provided $B_1, B_1', \ldots, B_{ns}, B_{ns}'$ do not contain any enroll commands and $E_1^s, \ldots, E_{ns}^s$ 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 to $B_1, \ldots, B_{ns}$, 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 'formation rule' in [1].

For proving cooperation we also need the preservation rule (I9, in the appendix). Finally to complete the proof system the substitution rule (I10) and the auxiliary variable rule (I11) are needed.

For example, consider the program $P::[P_1 \| P_2 \| P_3]$, where

$P_1::E_2(a_1)$

$P_2::a_2 := 5;\ E_1(a_2+1)$

$P_3::E_3(a_3)$.

For the rest of the section $E \equiv E^{\text{broadcast}}$.

Note that P_2 enrolls as the transmitter and P_1, P_3 enroll as recipients. Using the system above we can prove:

$$\{true\}\ [P_1 \| P_2 \| P_3]\ \{a_1 = a_3 = 6 \wedge a_2 = 5\}.$$

The proof outline is:

$$P_1\colon \{true\}\ E_2(a_1)\ \{a_1 = 6\}$$

$$P_2\colon \{true\}\ a_2 := 5\ \{a_2 = 5\}\ E_1(a_2+1)\ \{a_2 = 5\}$$

$$P_3\colon \{true\}\ E_3(a_3)\ \{a_3 = 6\}$$

and we may choose $SI \equiv true$. There is only one matching enrollment, so for cooperation we must prove

$$\{a_2 = 5\}\ [E_1(a_2+1) \| E_2(a_1) \| E_3(a_3)]\ \{a_1 = a_3 = 6 \wedge a_2 = 5\}.$$

Using the proof that

$$\{x_1 = C\}\ B_{\text{broadcast}}\ \{z_2 = z_3 = C\}$$

which was given above, we take C to be 6 and get

$$\{x_1 = 6\}\ B_{\text{broadcast}}\ \{z_2 = z_3 = 6\}.$$

By the enrollment rule we get

$$\frac{\{x_1 = 6\}\ B_{\text{broadcast}}\ \{z_2 = z_3 = 6\}}{\{x_1 = 6[a_2+1/x_1]\}\ [E_1(a_2+1) \| E_2(a_1) \| E_3(a_3)]\ \{z_2 = z_3 = 6[a_1, a_3/z_2, z_3]\}}$$

and after substitution

$$\{a_2 + 1 = 6\}\ [E_1(a_2+1) \| E_2(a_1) \| E_3(a_3)]\ \{a_1 = a_3 = 6\}.$$

By the preservation axiom, we can prove

$$\{a_2 = 5\}\ [E_1(a_2+1) \| E_2(a_1) \| E_3(a_3)]\ \{a_2 = 5\}.$$

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

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) [21].

Consider the enrollments shown in Fig. 14. In this example there are four syntactically matching enrollments (denoted 1, 2, 3, 4). Two of them, namely 3 and 4, are not semantically matching enrollments (that is, they will *never* take place). The other two, namely 1 and 2, are semantically matching. To verify the program, three auxiliary variables i, j, and k are used. See the proof outline in Fig. 15. We choose $SI \equiv i = j = k$.

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

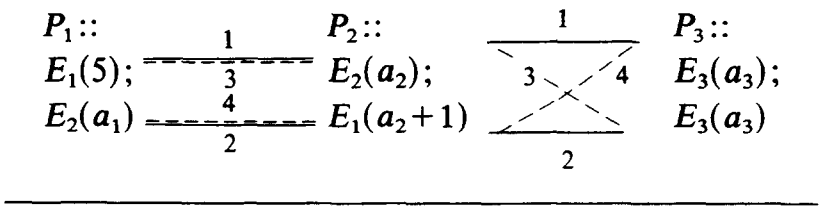

Fig. 14. Matching enrollments.

of the preconditions contradicts the invariant, so it trivially passes the cooperation test.

(1) We must prove

$$\{\mathrm{SI} \wedge i=j=k=0\}$$

$$[\langle E_1(5); i:=1\rangle \| E_2(a_2); j:=1\rangle \| \langle E_3(a_3); k:=1\rangle]$$

$$\{\mathrm{SI} \wedge a_2=5 \wedge i=j=k=1\}.$$

Taking C to be 5, we get by the enrollment rule

$$\{\mathit{true}\}\,[E_1(5) \| E_2(a_2) \| E_3(a_3)]\,\{a_2=a_3=5\}.$$

By the assignment and preservation axioms.

$$\{a_2=5\}\; i:=1; j:=1; k:=1\;\{i=j=k=1 \wedge a_2=5\}.$$

By applying the consequence and rearrangement rules the proof of (1) is finished.

(2) We must prove

$$\{\mathrm{SI} \wedge a_2=5 \wedge i=j=k=1\}$$

$$[\langle E_1(a_2+1)\rangle \| \langle E_2(a_1)\rangle \| \langle E_3(a_3)\rangle]$$

$$\{\mathrm{SI} \wedge a_1=a_3=6 \wedge a_2=5\}.$$

From the previous example, we know that

$$\{a_2=5\}\,[E_1(a_2+1) \| E_2(a_1) \| E_3(a_3)]\,\{a_1=a_3=6 \wedge a_2=5\}.$$

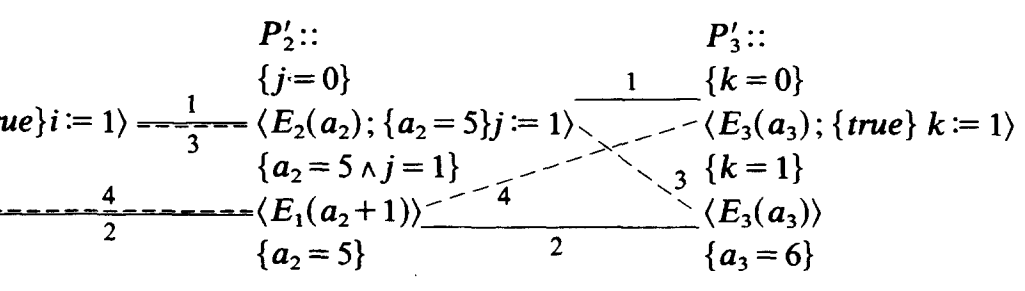

Fig. 15. Proof outline for bracketed program.

We finish the proof of (2) by applying the preservation axiom and the conjunction rule. Hence, by the parallel composition, consequence, and auxiliary variables rules

$$\{i=0 \wedge j=0 \wedge k=0\}\,[P_1 \| P_2 \| P_3]\,\{a_1=a_3=6 \wedge a_2=5\}.$$

Finally by applying the substitution rule we obtain

$$\{true\}\,[P_1 \| P_2 \| P_3]\,\{a_1=a_3=6 \wedge a_2=5\},$$

which completes our proof.

Before ending this section we want to clarify a point concerning the extension of the proof system for ADA [8], to any mixture of primitive call-accept communications and script enrollments. Such an 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 [8, Section 3] by restricting the notation of bracketing in such a 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 a bracketed task; the rest of the details in the extension are rather technical.

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; CALL $T.a$(*arguments*); B_2,
   (b) B_1; ENROLL IN s AS r_j(*arguments*); B_2,
   (c) ACCEPT b(*parameters*) DO B_1,
   (d) B_2 ENDACCEPT;

where B_1 and B_2 do not contain any entry call or accept or enroll, and may be null statements,

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

5.3. Example: Rotate

We now present a script and two different patterns of enrollment to this script, yielding two different effects in the enrolling program. The script *Rotate* consists of m roles arranged as a ring configuration. Each role R_i has a formal parameter x_i with an initial value denote 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 of each role transferring its initial value to its right neighbor is called rotate right. The script is shown in Fig. 16.

```
SCRIPT rotate::

[ROLE_{i=1}^{m} R_i (IN OUT 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 → send_i := true
   □
   ¬receive_i; R_{i-1}?temp_i → receive_i := true
  ];
 x_i := temp_i
]
```

Fig. 16. Rotate script.

Using the CSP proof system, we prove

$$\left\{\bigwedge_{i=1}^{m}(x_i = C_i)\right\} B_{\text{rotate}} \left\{\bigwedge_{i=1}^{m}(x_i = C_{i-1})\right\}.$$

To verify the script, two auxiliary variables s_i and r_i are introduced for each role R_i. The proof outline for the rotate script is shown in Fig. 17. We choose the script invariant to be

$$\text{SI} \equiv \bigwedge_{i=1}^{m}[(s_i \wedge r_{i+1}) \rightarrow temp_{i+1} = C_i].$$

Note that SI can refer to local variables. The meaning of SI is 'whenever R_i has sent and R_{i+1} has received, then $temp_{i+1}$ holds the value C_i".

```
R_i : {x_i = C_i ∧ s_i = r_i = false}
  send_i := false; receive_i := false;
  LI_i : {x_i = C_i ∧ send_i = s_i ∧ receive_i = r_i}
* [¬send_i; ⟨R_{i+1}!x_i → s_i := true; send_i := true⟩{LI_i}
  □
  ¬receive_i; ⟨R_{i-1}?temp_i → r_i := true; receive_i := true⟩{LI_i}
  ]{LI_i ∧ receive_i ∧ send_i}
  x_i := temp_i{s_i ∧ r_i ∧ x_i = temp_i}
```

Fig. 17. Proof outline for *rotate*

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

$$\{\neg send_i \wedge \neg receive_{i+1} \wedge \mathrm{LI}_i \wedge \mathrm{LI}_{i+1} \wedge \mathrm{SI}\}$$

$$[\langle R_{i+1}!x_i \to S_i := true;\ send_i := true\rangle \| \langle R_i?temp_{i+1} \to r_{i+1} := true;\ receive_{i+1} := true\rangle]$$

$$\{\mathrm{LI}_i \wedge \mathrm{LI}_{i+1} \wedge \mathrm{SI}\}.$$

By the arrow rule [1], it remains to be proved that

$$\left\{\neg send_i \wedge \neg receive_{i+1} \wedge \mathrm{LI}_i \wedge \mathrm{LI}_{i+1} \wedge \bigwedge_{\substack{j=1 \\ j\neq i}}^{m} [(s_j \wedge r_{j+1}) \to temp_{j+1} = C_j] \wedge temp_{i+1} = x_i\right\}$$

$$s_1 := true;\ send_i := true;\ r_{i+1} := true;\ receive_{i+1} := true$$

$$\{\mathrm{LI}_i \wedge \mathrm{LI}_{i+1} \wedge \mathrm{SI}\}$$

holds, where the above precondition is the postcondition of $R_{i+1}!x_i \| R_i?temp_{i+1}$. It is inferred from the axioms of communication and preservation.

Using the assignment axiom and consequence rule the required cooperation is obtained. By using the parallel composition rule we obtain

$$\left\{\mathrm{SI} \wedge \bigwedge_{i=1}^{m} [x_i = C_i \wedge s_i = r_i = \mathrm{false}]\right\} B_{\mathrm{rotate}} \left\{\mathrm{SI} \wedge \bigwedge_{i=1}^{m} [r_i \wedge s_i \wedge x_i = temp_i]\right\}.$$

The post-assertion $(\mathrm{SI} \wedge \bigwedge_{i=1}^{m} [r_i \wedge s_i \wedge x_i = temp_i])$ implies $(\bigwedge_{i=1}^{m} [x_i = C_{i-1}])$. So, finally, by the consequence, auxiliary variables, and substitution rules the required result is obtained.

We now show two enrollment patterns of 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 program, using a different pattern of enrollment to the rotate-script, the effect of 'rotate left' is achieved. For the rest of this section let $E \equiv E^{\mathrm{rotate}}$.

Rotate right

The rotate right enrollment is

$$P :: \left[\bigparallel_{i=1}^{m} P_i\right]$$

$$P_i :: a_i := i;\ E_i(a_i)$$

We prove that

$$\{true\}\ P \left\{\bigwedge_{i=1}^{m} (a_i = i-1)\right\}.$$

The proof outline is

$$P_i: \{true\}\ a_i := i\ \{a_i = i\}\ E_i(a_i)\ \{a_i = i-1\}$$

and we may choose SI $\equiv$ *true*.

For cooperation we must prove

$$\left\{\bigwedge_{i=1}^{m} (a_i = i)\right\}\left[\|_{i=1}^{m} E_i(a_i)\right]\left\{\bigwedge_{i=1}^{m} (a_i = i-1)\right\}.$$

We take C_i to be i and get

$$\left\{\bigwedge_{i=1}^{m} (x_i = i)\right\} B_{\text{rotate}} \left\{\bigwedge_{i=1}^{m} (x_i = i-1)\right\}.$$

By the enrollment rule, we obtain

$$\frac{\left\{\bigwedge_{i=1}^{m} (x_i = i)\right\} B_{\text{rotate}} \left\{\bigwedge_{i=1}^{m} (x_i = i-1)\right\}}{\left\{\bigwedge_{i=1}^{m} (x_i = i)[a_i/x_i]\right\}\left[\|_{i=1}^{m} E_i(a_i)\right]\left\{\bigwedge_{i=1}^{m} (x_i = i-1)[a_i/x_i]\right\}}$$

which after substitution yields the required result. By the parallel composition rule the proof is finished.

Rotate left

The rotate left enrollment is

$$P :: \left[\|_{i=1}^{m} P_i\right]$$

$$P_i :: a_i := i;\ E_{m-i+1}(a_i)$$

For simplicity, we denote $m-i+1$ by k_i. Note that $\{k_1, \ldots, k_m\}$ is a permutation of $\{1, \ldots, m\}$, so P has exactly one matching enrollment.

We prove that

$$\{true\}\ P \left\{\bigwedge_{i=1}^{m} (a_i = i+1)\right\}.$$

The proof outline is

$$P_i: \{true\}\ a_i := i\ \{a_i = i\}\ E_{k_i}(a_i)\ \{a_i = i+1\}$$

and we may choose SI $\equiv$ *true*.

Note that $[\|_{i=1}^{m} E_{k_i}(a_i)]$ is the same as $[\|_{i=1}^{m} E_i(a_{k_i})]$, so we can interchange them.

For cooperation we must prove

$$\left\{\bigwedge_{i=1}^{m} (a_i = i)\right\}\left[\|_{i=1}^{m} E_i(a_{k_i})\right]\left\{\bigwedge_{i=1}^{m} (a_i = i+1)\right\}.$$

We take C_i to be k_i and get

$$\left\{\bigwedge_{i=1}^{m}(x_i = k_i)\right\} B_{\text{rotate}} \left\{\bigwedge_{i=1}^{m}(x_i = k_{i-1})\right\}.$$

Because $k_{i-1} = m-(i-1)+1 = k_i+1$,

$$\left\{\bigwedge_{i=1}^{m}(x_i = k_i)\right\} B_{\text{rotate}} \left\{\bigwedge_{i=1}^{m}(x_i = k_i+1)\right\}.$$

By the enrollment rule, we obtain

$$\frac{\left\{\bigwedge_{i=1}^{m}(x_i = k_i)\right\} B_{\text{rotate}} \left\{\bigwedge_{i=1}^{m}(x_i = k_i+1)\right\}}{\left\{\bigwedge_{i=1}^{m}(x_i = k_i)[a_{k_i}/x_i]\right\}\left[\parallel_{i=1}^{m} E_i(a_{k_i})\right]\left\{\bigwedge_{i=1}^{m}(x_i = k_i+1)[a_{k_i}/x_i]\right\}}.$$

After substitution we get

$$\left\{\bigwedge_{i=1}^{m}(a_{k_i} = k_i)\right\}\left[\parallel_{i=1}^{m} E_i(a_{k_i})\right]\left\{\bigwedge_{i=1}^{m}(a_{k_i} = k_i+1)\right\}$$

which is clearly the same as the required conclusion. The proof is finished by the parallel composition rule.

Other definitions of k_i can cause interesting results, such as rotate k positions.

5.4. A recursive 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 such that 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 constrains 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.

The well-known conventional solution to the game makes use of a recursive procedure with four parameters. Three of the parameters represent the poles and the fourth is an integer specifying the number of discs to be moved. The algorithm 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 temporary. In step two, a single disc is moved from the source to the destination. In step three, $N-1$ discs are moved, using a recursive call, from the spare to the destination, using the source as temporary.

We now introduce a solution using a recursive script. It is similarly structured to the conventional one, and makes use of the same three steps. Although it is distributed, no parallel computation is involved. Parallel computation may take place in a generalization of the game where more than three poles are allowed.

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 stack 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 auxilary simple script named *move*, which has two roles, named give and take. Each move 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 with three roles (named source, destination, and spare) and N discs is described by the same three steps used in the conventional solution.

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

(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.

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

The hanoi script is shown in Fig. 18. The move script is shown in Fig. 19.

We now verify this example. First consider the script *move*. Using the proof system for CSP [1], we can prove

$$\{X = s \cdot X_0 \wedge Y = Y_0\}\ \mathit{Body}_{\mathrm{move}}\ \{X = X_0 \wedge Y = s \cdot Y_0\},$$

where 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 stacks X and Y. By $s \cdot X_0$ we mean that s is placed on top of the stack of discs denoted by X_0.

It is required that the s disc 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 those requirements are satisfied (by the actual parameters) when the move script is used (in Step 2) by the hanoi script. The proof outline of *move* is shown in Fig. 20. It is simple to see that the constraint that "no disc may be placed on top of a smaller one" is observed by this script if the initial requirements are satisfied.

```
SCRIPT hanoi::
  INITIATION:DELAYED;
  TERMINATION: DELAYED;
  [ROLE source(IN n1: integer, IN OUT 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(IN n2: integer, IN OUT 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(IN n3: integer, IN OUT C: (stack of discs)::
     [n3 ≠ 1 → ENROLL IN hanoi AS destination(n2 − 1, B) □ n3 = 1 → skip];
     [n3 ≠ 1 → ENROLL IN hanoi AS source(n3 − 1, C) □ n3 = 1 → skip]
  ].
```

FIg. 18. Towers of Hanoi script.

```
SCRIPT move::
  INITIATION:DELAYED;
  TERMINATION: DELAYED;
  [ROLE give(IN OUT X: stack of discs)::
     VAR temp1: integer;
     temp1 := pop(X);
     take!temp1
  ||
     ROLE take(IN OUT Y: stack of discs)::
     VAR temp2: integer;
     give?temp2;
     push(Y, temp2)
  ].
```

Fig. 19. Move script.

$[give:\{X = s \cdot X_0\}$
$\quad temp_1 := pop(X);$
$\quad \{temp_1 = s \wedge X = X_0\}$
$\quad take!temp_1$
$\quad \{X = X_0\}$
$\|$
$take:\{Y = Y_0\}$
$\quad give?temp_2;$
$\quad \{temp_2 = s \wedge Y = Y_0\}$
$\quad push(Y, temp_2)$
$\quad \{Y = s \cdot Y_0\}$
$]$.

The script invariant is SI $\equiv$ *true*.

Cooperation is proved easily using:

the communication axiom, the preservation axiom, and the consequence rule.

All that remains is the application of the parallel composition rule.

Fig. 20. Move script proof outline.

Finally we verify the hanoi script. We first prove

$$\{A = A[1\,..\,W] \wedge B = B_0 \wedge C = C_0 \wedge n_1 = n_2 = n_3 = N\}$$
$$[E^{\text{hanoi}}_{\text{source}}(n_1, A) \| E^{\text{hanoi}}_{\text{dest}}(n_2, B) \| E^{\text{hanoi}}_{\text{spare}}(n_3, C)] \qquad (*)$$
$$\{A = A[N+1\,..\,W] \wedge B = A[1\,..\,N] \cdot B_0 \wedge C = C_0 \wedge n_1 = n_2 = n_3 = N\}$$

where $A[1\,..\,W]$, B_0, C_0 are used to freeze the initial state of the stacks A, B, and C. The term $A[1\,..\,W]$ denotes an ordered stack of W discs, where for each i, j such that $1 \leqslant i < j \leqslant W$, disc $A[i]$ is smaller than disc $A[j]$. The term N is an integer such that $1 \leqslant N \leqslant W$.

For the sake of the proof we assume that any one of the $A[1\,..\,W]$ discs is smaller than 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 that

$$(*) \vdash \{A = A[1\,..\,W] \wedge B = B_0 \wedge C = C_0 \wedge n_1 = n_2 = n_3 = N\}$$
$$Body_{\text{hanoi}}$$
$$\{A = A[N+1\,..\,W] \wedge B = A[1\,..\,N] \cdot B_0 \wedge C = C_0 \wedge n_1 = n_2 = n_3 = N\}.$$

The proof outline of the hanoi script is given in Fig. 21.

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

Assume (*).
Let $\alpha(k) \equiv A = A(k \ldots W] \wedge n_1 = N.$
$\beta_1 \equiv B = B_0 \wedge n_2 = N.$
$\beta_2 \equiv B = A[N] * B_0 \wedge n_2 = N.$
$\beta_3 \equiv B = A[1 \ldots N] \cdot B_0 \wedge n_2 = N.$
$\gamma_1 \equiv C = C_0 \wedge n_3 = N.$
$\gamma_2 \equiv C = A[1 \ldots N-1] \cdot C_0 \wedge n_3 = N.$

[*source*: $\{\alpha(1)\}$
$[n_1 \neq 1 \rightarrow E^{\text{hanoi}}_{\text{source}}(n_1 - 1, A)\{\alpha(N)\}$
$\square\ n_1 = 1 \rightarrow skip\ \{\alpha(N)\}$
$]\ \{\alpha(N)\}$
$E^{\text{move}}_{\text{give}}(A);\ \{\alpha(N+1)\}$
$[n_1 \neq 1 \rightarrow E^{\text{hanoi}}_{\text{spare}}(n_1 - 1, A)\ \{\alpha(N+1)\}$
$\square\ n_1 = 1 \rightarrow skip\ \{\alpha(N+1)\}$
$]\ \{\alpha(N+1)\}$
‖
dest: $\{\beta_1\}$
$[n_2 \neq 1 \rightarrow E^{\text{hanoi}}_{\text{spare}}(n_2 - 1, B)\ \{\beta_1\}$
$\square\ n_2 = 1 \rightarrow skip\ \{\beta_1\}$
$]\ \{\beta_1\}$
$E^{\text{move}}_{\text{take}}(A);\ \{\beta_2\}$
$[n_2 \neq 1 \rightarrow E^{\text{hanoi}}_{\text{dest}}(n_2 - 1, B)\ \{\beta_2\}$
$\square\ n_2 = 1 \rightarrow skip\ \{\beta_3\}$
$]\ \{\beta v_3\}$
‖
spare: $\{\gamma_1\}$
$[n_3 \neq 1 \rightarrow E^{\text{hanoi}}_{\text{dest}}(n_3 - 1, C)\ \{\gamma_2\}$
$\square\ n_3 = 1 \rightarrow skip\ \{\gamma_2\}$
$]\ \{\gamma_2\}$
$[n_3 \neq 1 \rightarrow E^{\text{hanoi}}_{\text{source}}(n_3 - 1, C)\ \{\gamma_1\}$
$\square\ n_3 = 1 \rightarrow skip\ \{\gamma_1\}$
$]\ \{\gamma_1\}$
].
The script invariant is SI $\equiv$ *true*.

Fig. 21. Hanoi script proof outline.

Step 1. We must prove

$$\{A = A[1\,..\,W] \wedge B = B_0 \wedge C = C_0 \wedge n_1 = n_2 = n_3 = N\}$$
$$[E^{\text{hanoi}}_{\text{source}}(n_1 - 1, A) \| E^{\text{hanoi}}_{\text{dest}}(n_3 - 1, C) \| E^{\text{hanoi}}_{\text{spare}}(n_2 - 1, B)] \tag{1}$$
$$\{A = A[N\,..\,W] \wedge B = B_0 \wedge C = A[1\,..\,N-1] \cdot C_0 \wedge n_1 = n_2 = n_3 = N\}.$$

The proof starts with (*).

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

$$\{A = A[1\,..\,W] \wedge B = B_0 \wedge C = C_0 \wedge n_1 = n_2 = n_3 = N - 1\}$$
$$[E^{\text{hanoi}}_{\text{source}}(n_1, A) \| E^{\text{hanoi}}_{\text{dest}}(n_2, B) \| E^{\text{hanoi}}_{\text{spare}}(n_3, C)]$$
$$\{A = A[N\,..\,W] \wedge B = A[1\,..\,N-1] \cdot B_0 \wedge C = C_0 \vee n_1 = n_2 = n_3 = N - 1\}.$$

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

$$\{A = A[1\,..\,W] \wedge B = B_0 \wedge C = C_0 \wedge n_1 = n_2 = n_3 = N - 1\}$$
$$[E^{\text{hanoi}}_{\text{source}}(n_1, A) \| E^{\text{hanoi}}_{\text{dest}}(n_3, C) \| E^{\text{hanoi}}_{\text{spare}}(n_2, B)]$$
$$\{A = A[N\,..\,W] \wedge B = B_0 \wedge C = A[1\,..\,N-1] \cdot C_0 \wedge n_1 = n_2 = n_3 = N - 1\}.$$

Finally, by the parameter substitution rule ($n_1 - 1$, $n_2 - 1$, $n_3 - 1$ for n_1, n_2, n_3), the required result is obtained.

Step 2. We must prove

$$\{A = A[N\,..\,W] \wedge B = B_0 \wedge n_1 = n_2 = N\}$$
$$[E^{\text{move}}_{\text{give}}(A) \| E^{\text{move}}_{\text{take}}(B)] \tag{2}$$
$$\{A = A[N+1\,..\,W] \wedge B = A(N) \cdot B_0 \wedge n_1 = n_2 = N\}.$$

Using the proof that

$$\{X = s \cdot X_0 \wedge Y = Y_0\}\ \mathit{Body}_{\text{move}}\ \{X = X_0 \wedge Y = s \cdot Y_0\},$$

which was given earlier, we take s, X_0, Y_0 to be $A[N]$, $A(N+1\,..\,W]$, B_0, and get

$$\{X = A[N\,..\,W] \wedge Y = B_0\}\ \mathit{Body}_{\text{move}}\ \{X = A[N+1\,..\,W] \wedge Y = A(N) \cdot B_0\}.$$

Note that $A(N)$, $A(N+1\,..\,W]$, B_0 satisfy the precondition of the move script. By the enrollment rule we get

$$\frac{\{X = A[N\,..\,W] \wedge Y = B_0\}\ \mathit{Body}_{\text{move}}\ \{X = A[N+1\,..\,W] \wedge Y = A[N] \cdot B_0\}}{\begin{array}{l}\{X = A[N\,..\,W] \wedge Y = B_0[A, B/X, Y]\} \\ [E^{\text{move}}_{\text{give}}(A) \| E^{\text{move}}_{\text{take}}(B)] \\ \{X = A[N+1\,..\,W] \wedge Y = A[N] \cdot B_0[A, B/X, Y]\}\end{array}}$$

and after substitution

$$\{A = A[N\,..\,W] \wedge B = B_0\}$$
$$[E^{\text{move}}_{\text{give}}(A) \| E^{\text{move}}_{\text{take}}(B)]$$
$$\{A = A[N+1\,..\,W]B = A[N] \cdot B_0\}.$$

By the preservation axiom

$$\{n_1 = n_2 = N\}\,[E^{\text{move}}_{\text{give}}(A) \| E^{\text{move}}_{\text{take}}(B)]\,\{n_1 = n_2 = N\}.$$

Using the conjunction rule, the required cooperation is obtained.

Step 3. We must prove

$$\{A = A[N+1\,..\,W] \wedge B = A(N) \cdot B_0 \wedge C = A[1\,..\,N-1] \cdot C_0 \wedge n_1 = n_2 = n_3 = N\}$$
$$[E^{\text{hanoi}}_{\text{source}}(n_3 - 1, C) \| E^{\text{hanoi}}_{\text{dest}}(n_2 - 1, B) \| E^{\text{hanoi}}_{\text{spare}}(n_1 - 1, A)]$$
$$\{A = A[N+1\,..\,W] \wedge B = A[1\,..\,N] \cdot B_0 \wedge C = C_0 \wedge n_1 = n_2 = n_3 = N\}. \quad (3)$$

The proof starts with (1).

By the parameter substitution rule (A, B, C for B, C, A and n_1, n_2, n_3 for n_2, n_3, n_1) and the variable substitution rule ($A[N+1,..\,W]$, $A[n] \cdot B_0$, C_0 for B_0, C_0, $A[N\,..\,W]$) the required result is obtained.

By applying the parallel composition rule, the required result about the body of the hanoi script is obtained. Finally by the recursion rule, the proof of (*) is obtained.

Consider, again, the constraint that no disc may be 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 constraint is observed. Furthermore, the requirements (Step 2) are always satisfied. Thus we informally proved that the constraint is observed within the hanoi script, which means that it is an invariant.

Consider, again, the definition of the game. The claim we have just proved is stronger than needed. So, if we now take (*) and use the consequence rule and variable substitution rule to substitute, 'empty, empty, empty' for $A[N+1\,..\,W]$, B_0, C_0, where 'empty' denotes an empty stack, we get

$$\{A = A[1\,..\,N] \wedge B = C = \text{empty} \wedge n_1 = n_2 = n_3 = N\}$$
$$[H^{\text{hanoi}}_{\text{source}}(n_1, A) \| E^{\text{hanoi}}_{\text{dest}}(n_2, B) \| E^{\text{hanoi}}_{\text{spare}}(n_3, C)]$$
$$\{A = \text{empty} \wedge B = A[1\,..\,N] \wedge C = \text{empty}\}$$

which is exactly what was defined as the objective of the game.

Note that the last formula cannot be proved directly using the recursion rule because of Step 3. Note also that when we have assumed empty stacks for B_0 and C_0, the assumption that any one of the $A[1\,..\,W]$ discs is smaller than any disc of B_0 or C_0 is vacuous.

6. Deadlock freedom

In this section we deal only with the case where both initiation and termination are delayed. 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 aways available, and there is no need to wait till one performance terminates in order to start a new one. The multiplicity of instances is essential for the deadlock-freedom proof system presented below.

We show how the proof system can be used for proving deadlock freedom of a given program. We assume that there exists a deadlock freedom proof system for the host language (for example, the proof systems presented in [1, 8] for CSP and ADA, respectively).

As in [8] we use a notion called *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. 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 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.

We define a program P to be *deadlock free relative to a precondition p* if no execution of P, starting in an initial state satisfying p, ends in a deadock. The approach we use in proving freedom of deadlock is similar to that of the previous section. 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 be the same. For example for each script s, $\{true\}\ s\ \{true\}$ holds but if there exist an initial state in shich s ends in a deadlock, then for proving deadlock freedom, $df(s)$ has to be stronger then 'true'. As with $pre(s)$, the $df(s)$ predicate may refer only 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 [1, 8, 21] where, in order to prove deadock freedom, first all possible deadlock situations (also called blocked situation in [1, 21] and blocked f.o.c. in [8]) are showed to be unreachable. Using such a method would have forced us to give up modularity handling all the scripts at once instead of separating them, as we wish.

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

```
SCRIPT s::
  [ROLE r1 (IN OUT x1 : integer)::
     [x1 > 5 → r2!x1 □ x1 ≤ 5 → r2?x1]
   ||
   ROLE r2 (IN OUT x2 : integer)::
     [x2 > 5 → r1?x2 □ x2 ≤ 5 → r1!x2]
  ].
```

Fig. 22. Demonstrating $df(s)$.

(but not sufficient) for a situation to be p.b.s. is that each of the script's 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 only by the processes belonging to one script. We prove deadlock freedom of a script by identifying all its p.b.s. and showing 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 an enroll command, the system described below is used.

An example, shown in Fig. 22, will demonstrate a $df(s)$ predicate associated with a script s that uses CSP's primitive communication only. It is also used later to illustrate the concept of p.b.s. Using the CSP proof system it is easy to prove that s is deadlock free relative to

$$df(s) \equiv (x_1 > 5 \wedge x_2 > 5) \vee (x_1 \leq 5 \wedge x_2 \leq 5).$$

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.

We define a matching enrollment, $E^t_1, \ldots, E^t_{nt}$, to be a *df-matching enrollment* if

$$\bigwedge_{i=1}^{nt} [pre(E^t_i(\vec{a}_{k_i}, \vec{b}_{k_i}, \vec{c}_{k_i}))] \wedge \mathrm{SI},$$

(the conjunction of all the preassertions of the enroll commands and the script invariant of the enrolling processes) implies

$$df(t)[\vec{a}, \vec{b}/\vec{x}, \vec{y}].$$

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

We define $\langle B_1\rangle, \ldots, \langle B_{ns}\rangle$ to be *df-matching bracketed sections*, if they contain a *df*-matching enrollment $(E^s_1, \ldots, E^s_{ns})$ to some script s.

We now introduce the 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 cannot 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

(1) If it has terminated, its e.c. is empty;

(2) If 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 t-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.

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

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

(2) Not all processes have empty e.c.'s.

Formally, condition (1) of p.b.s. is

$$\exists C \in \mathit{combination}(n\_tuple) \underset{\text{match}}{\forall} \langle B_1\rangle, \ldots, \langle B_{nt}\rangle \in C$$
$$\left[\neg\left(\bigwedge_{i=1}^{nt} (\mathit{pre}(\langle B_i\rangle)) \wedge \mathrm{SI} \rightarrow \mathit{df}(t)\right)\right]$$

where $\mathit{combination}(n\_tuple)$ is the set of all combinations obtained from the n-tuple of e.c.'s that characterize the above situation, C describes one of those combinations and $\langle B_1\rangle, \ldots, \langle B_{nt}\rangle$ 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 intoduced 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 (2) holds trivially.

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

(2) Let P::[$\{a_1 = 6\}$ E_1 $\{true\} \| \{a_2 = 6\}$ E_1 $\{true\}$]. There exists one combination only, which does not include any matching enrollments. Hence, condition (1) holds, and the situation is a p.b.s.

(3) Let P::[$\{a_1 = 6\}$ $E_1\{true\} \| \{a_2 = 4\}$ E_2 $\{true\}$]. There exists one combination only, including a matching enrollment, which is not a *df*-matching enrollment. Hence, condition (1) holds, and again we have a p.b.s.

(4) Let $P::[\{a_1=4\}\ E_1\ \{true\}\|\{a_2=6\}\ E_1\ \{true\}\|\{a_3=6\}\ E_2\ \{true\}]$. Two combinations can be obtained. In the first combination, the third and second processes form a *df*-matching enrollment, while in the second combination the third and first processes can also form a matching enrollment, which is not a *df*-matching enrollment. Hence condition (1) holds, and it is a p.b.s.

(5) Let

$$P::[\{a_1=4\}\ E_1\ \{true\}\|\{a_2=4\}\ E_1\{true\}\|\{a_3=6\}\ E_2\ \{true\}\|\{a_4=6\}\ E_2\ \{true\}].$$

Two combinations can be obtained, both include exactly two matching enrollments, which are not *df*-matching enrollments. Hence condition (1) holds, and it is a p.b.s.

(6) Let

$$P::[\{a_1=6\}\ E_1\ \{true\}\|a_2=4\}\ E_1\ \{true\}\|\{a_3=6\}\ E_2\ \{true\}\|\{a_4=6\}\ E_2\ \{true\}].$$

Two combinations can be obtained, both include exactly two matching enrollments where one of them is a *df*-matching enrollment. Hence condition (1) does not hold, and it is not a p.b.s.

(7) Let

$$P::[\{a_1=4\}\ E_1\ \{true\}\|\{a_2=6\}\ E_1\ \{true\}\|\{a_3=4\}\ E_2\ \{true\}\|\{a_4=6\}\ E_2\ \{true\}].$$

Two combinations can be obtained. In the first combination, the first and third processes and the second 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 (1) 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 is associated with a blocked process P_i is either post (P_i) if it has an empty e.c. or it is the preassertion 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.'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 [1, Section 4]).

Theorem. *Given a proof of* $\{df(s)\}\ s\ \{q\}$ *with a script invariant* SI, *s is a deadlock free (relative to* $df(s)$*) if for every potentially blocked n-tuple* $\langle p_1, \ldots, p_n\rangle$, $\neg\bigwedge_{i=1}^{n} p_i \wedge \mathrm{SI})$ *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 discussions of proof of absence of deadlocks [1, p. 378].

In the recursive case, we must show how to prove that a recursive script s is deadlock free relative to some assertion $df(s)$. The problem that arises is how to decide if a recursive matching enrollment is a df-matching enrollment. Such a decision is based on knowing the assertion relative to which the script is deadlock free, where 'the script' is the one the matching enrollments enroll to. In the case of recursive matching enrollments, $df(s)$ is the assertion that must be proved. The solution is the standard one when treating recursion: permit the use of the desired conclusion about an enrollment as an assumption in the proof of the body.

Thus to decide if a recursive matching enrollment to script s is a df-matching enrollment, we assume that s is deadlock free relative to $df(s)$. After all the recursive matching enrollments have been decided, we 'forget' the assumption and continue as usual. If from that point, using the known proof system, it is provable that s is deadlock free relative to $df(s)$, then indeed it is.

7. Future work

More work needs to be done with scripts to explore their potential for simplifying the programming of concurrent systems. Other issues such as distributed control of performances and practical implementation within various host languages have to be addressed.

There are many natural extensions to scripts. One such is a dynamic arrays of roles, where the number of roles is not fixed until run-time. We term these dynamic arrays *open-ended scripts.* They would allow different instances of a script to take place with somewhat different role structures. The question of the completeness of the proof system and the extension of the system for proving termination should be studied. Another issue involves extending the enrollment mechanism to serve as a guard. Enrolling to computed scripts, extending [6], is worth considering.

Appendix

Notation

S: script named S.
$|S|$, ns: number of roles in the script S.
$E_j^S(\vec{a})$: enroll in S as $R_j(\vec{a})$.
$R_j^S(\vec{x}_j)$: role R_j in script S with formal data parameters $\vec{x}_j$, and body B_j.
B_S: body of $S(\|_{j=1}^{ns} B_j)$.
$pre(R_j^S)$: pre-condition of R_j^S.
$post(R_j^S)$: post-condition of R_j^S.

SI: script invariant.
$pre(S)$: pre-condition of B_S.
$post(S)$: post-condition of B_S $(\bigwedge_{j=1}^{ns} post(R_j^S) \wedge \mathrm{SI} \rightarrow post(S))$.
$df(S)$: predicate relative to which S is proved to be deadlock free.

Axioms and proof rules

I1. Assignment Axiom.

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

I2. Skip Axiom.

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

I3. Alternative Command Rule.

$$\frac{\{p \wedge b_i\}\ S_i\ \{q\},\ i = 1, \ldots, m}{\{p\} \left[\underset{i=1}{\overset{m}{\square}}\ b_i \rightarrow S_i \right] \{q\}}$$

I4. Repetitive Command Rule.

$$\frac{\{p \wedge b_i\}\ S_i\ \{p\},\ i = 1, \ldots, m}{\{p\} * \left[\underset{i=1}{\overset{m}{\square}}\ b_i \rightarrow S_i \right] \{p \wedge \neg(b_1 \vee \cdots \vee b_m)\}}$$

I5. Composition Rule.

$$\frac{\{p\}\ S_1\ \{q\}, \{q\}\ S_2\ \{r\}}{\{p\}\ S_1; S_2\ \{r\}}$$

I6. Consequence Rule.

$$\frac{p \rightarrow p_1, \{p_1\}\ S_1\ \{q_1\}, q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

I7. Conjunction Rule.

$$\frac{\{p\}\ S\ \{q\}, \{p\}\ S\ \{r\}}{\{p\}\ S\ \{q \wedge r\}}$$

I8. Disjunction Rule.

$$\frac{\{p_1\}\ S\ \{q\}, \{p_2\}\ S\ \{q\}}{\{p_1 \vee p_2\}\ S\ \{q\}}$$

I9. 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.

I10. Substitution Rule.

$$\frac{\{p\}\, S\, \{q\}}{\{p[t/z]\}\, S\, \{q\}}$$

provided z does not appear from in S and q. The substitution rule is needed to eliminate auxiliary variables from the pre-assertion.

I11. 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 free any variables from AV, and S is obtained from S' by deleting all assignments to variables in AV,

$$\frac{\{p\}\, S'\, \{q\}}{\{p\}\, S\, \{q\}}$$

I12. Communication Axiom.

$$\{\text{true}\}\ \mathrm{P_i}?x \| P_j!\mathrm{y}\ \{x = y\}$$

provided $P_i?x$ and $P_j!y$ are taken from P_j and P_i, respectively.

I13. Arrow Rule.

$$\frac{\{p\}\, (\alpha; S) \| S_1\, \{q\}}{\{p\}\, (\alpha \to S) \| S_1\, \{q\}}$$

where α stands for any input/output command.

I14. Parallel Composition Rule.

$$\frac{\text{proofs of } \{p_i\}\ P_i\ \{q_i\},\ i = 1, \ldots, n, \text{ cooperate}}{\{p_1 \wedge \cdots \wedge p_n \wedge \mathrm{SI}\}\ [P_1 \| \ldots \| P_n]\ \{q_1 \wedge \cdots \wedge q_n \wedge \mathrm{SI}\}}$$

New rules

Enrollment Rule. For a script s and matching enrollments $E_1^s, \ldots, E_{ns}^s$,

$$\frac{\{pre(s)\}\ B_s\ \{post\ (s)\}}{\{pre(s)[\vec{a}; \vec{b}/\vec{x}; \vec{y}]\} \left[\underset{j=1}{\overset{ns}{\|}} E_j^S(\vec{a}_{k_j}, \vec{b}_{k_j}, \vec{c}_{k_j}) \right] \{post(s)[\vec{b}; \vec{c}/\vec{y}; \vec{z}]\}}$$

Parameter Substitution Rule.

$$\frac{\{p\}\left[\underset{j=1}{\overset{ns}{\|}} E_j^s(\vec{x}, \vec{y}, \vec{z})\right]\{q\}}{\{p[\vec{d}; \vec{e}/\vec{x}; \vec{y}]\}\left[\underset{j=1}{\overset{ns}{\|}} E_j^s(\vec{d}_{k_j}, \vec{e}_{k_j}, \vec{f}_{k_j})\right]\{q[\vec{e}; \vec{f}/\vec{y}; \vec{z}]\}}$$

where $var(\vec{d}; \vec{e}; \vec{f}) \cap free(p, q) \subseteq \{\vec{x}, \vec{y}, \vec{z}\}$.

Variable Substitution Rule.

$$\frac{\{p\}\left[\underset{j=1}{\overset{ns}{\|}} E_j^s(\vec{a}_{k_j}, \vec{b}_{k_j}, \vec{c}_{k_j})\right]\{q\}}{\{p[\vec{t}/\vec{r}]\}\left[\underset{j=1}{\overset{ns}{\|}} E_j^s(\vec{a}_{k_j}, \vec{b}_{k_j}, \vec{c}_{k_j})\right]\{q[\vec{t}/\vec{r}]\}}$$

where $var(\vec{t}; \vec{r}) \cap var(\vec{a}; \vec{b}; \vec{c}) = \emptyset$.

Enrollment Axiom.

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

Rearrangement Rule.

$$\frac{\{p\}\ B_1; \ldots; B_{ns}\ \{p_1\}, \{p_1\}\left[\underset{j=1}{\overset{ns}{\|}} E_j^S\right]\{p_2\}, \{p_2\}\ B_1'; \ldots; B_{ns}'\ \{q\}}{\{p\}\left[\underset{j=1}{\overset{ns}{\|}} (B_j; E_j^s; B_j')\right]\{q\}}$$

Recursion Rule.

$$\frac{\{pre(s)\}\left[\underset{j=1}{\overset{ns}{\|}} E_j^s(\vec{x}_j, \vec{y}_j, \vec{z}_j)\right]\{post(s)\} \vdash \{pre(s)\}\ B_s\ \{post(s)\}}{\{pre(s)\}\left[\underset{j=1}{\overset{ns}{\|}} E_j^s(\vec{x}_j, \vec{y}_j, \vec{z}_j)\right]\{post(s)\}}$$

Acknowledgment

We would like to thank the attendees of the 1983 IFIP Working Group 2.2 Meeting on Formal Description of Programming Concepts for their comments and suggestions on a early draft of this report. We also thank Shmuel Katz and Amir Pnueli for various discussions concerning the proof rules. Thanks are also due to K. R. Apt and an anonymous referee for their comments on an earlier draft of this paper.

Part of the first author's work was supported by the fund for the promotion of research, the Technion.

References

[1] K.R. Apt, N. Francez and W.P. De Roever, A proof system for communicating sequential process, *ACM Trans. Programming Languages and Systems* **2**(3) (1980) 359-385.

2] K.R. Apt, Ten years of Hoare logic: A survey (part 1), *ACM Trans. Programming Languages and Systems* **3**(4) (1981) 431-483.

[3] K.R. Apt, Formal justification of a proof system for communicating sequential processes, *J. ACM* **30**(1) (1983) 197-216.

[4] P. Brinch Hansen, The programming language Concurrent Pascal, *IEEE Trans. Software Engrg.* **1**(2) (1975) 199-207.

[5] T. Elrad and N. Francez, Decomposition of distributed programs into communication-closed layers, *Sci. Comput. Programming* **2**(3) (1982) 155-173.

[6] N. Francez, Extended naming conventions for communicating processes, *Proc. 9th Annual ACM Symposium on Principles of Programming Languages*, Albuquerque (1982) 40-45.

[7] N. Francez and S. Yemini, A fully abstract and composable inter-task communication construct, *ACM Trans. Programming Languages and Systems* (1985), to appear.

[8] R. Gerth and W.P. de Roever, A proof system for concurrent ADA programs, *Sci. Comput. Programming* **4**(2) (1984) 159-204.

[9] D. Gries and G. Levin, Assignment and procedure call proof rules, *ACM Trans. Programming Languages and Systems* **2**(4) (1980) 564-579.

[10] B. Hailpern and S. Owicki, Modular verification of concurrent programs, *Proc. 9th ACM Syposium on Principles of Programming Languages*, Albuquerque (1982) 322-336.

[11] P. Hilfinger, G. Feldman, R. Fitzgerald, I. Kimura, R.L. London, K.V.S. Prasad, V.R. Prasad, J. Rosenberg, M. Shaw, and W.A. Wulf (Ed.), An informal definition of Alphard (preliminary), Technical Report CMU-CS-78-105, Carnegie-Mellon University, February 1978.

[12] P.N. Hilfinger, Implementation strategies for Ada tasking idoms, *Proc. ACM-AdaTEC Conference on Ada*, Arlington (1982).

[13] C.A.R. Hoare, Procedures and parameters: An axiomatic approach, in: E. Engeler, Ed. *Symposium on Semantics of Algorithmic Languages*, Lecture Notes in Mathematics **188** (Springer, Berlin, 1971) 102-116.

[14] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21**(8) (1978) 666-677.

[15] M. Joseph, Schemes for communication, Technical Report CMU-CS-81-122, Carnegie-Mellon University, June 1981.

[16] H.F. Korth, Edge locks and deadlock avoidance in distributed systems, *Proc. ACM Symposium on Principles of Distributed Computing*, Ottawa (1982) 173-182.

[17] D.A. Lamb and P.N. Hilfinger, Simulation of procedure variables using Ada tasks, *IEEE Trans. Software Engrg.* **9**(1) (1983) 13-15.

[18] L. Lamport, Specifying concurrent program modules, *ACM Trans. Programming Languages and Systems* **5**(2) (1983) 190-222.

[19] B.H. Liskov, R.A. Atkinson, T. Bloom, J.E. Schaffert, R.W. Scheifler and A. Snyder, *CLU Reference Manual*, Lecture Notes in Computer Science **114** (Springer, Berlin, 1981).

[20] J.G. Mitchell, W. Maybury and R. Sweet, Mesa language manual (version 5.0), CSL-79-3, Xerox Palo Alto Research Center, April 1979.

[21] S.S. Owicki and D. Gries, An axiomatic proof technique for parallel programs, *Acta Informat.* **6** (1976) 319-340.

[22] R. Rashid and G. Robertson, Accent: A communication oriented network operating system kernel, *Proc. 8th ACM Symposium on Operating Systems Principles*, Asilomar (1981) 64-75.

[23] L.G. Reid, Control and communication in programming systems, Technical Report CMU-CS-80-142, Carnegie-Mellon University, September 1980.

[24] J. Skansholm, Multicast and synchronization in distributed systems, Research Report, Department of Computer Science, University of Goteborg, 1981.

[25] United States Department of Defense, Reference manual for the Ada programming language, ACM-AdaTEC, July 1982.

[26] D.W. Wall, Mechanisms for broadcast and selective broadcast, Ph.D. Thesis, Stanford University, 1980. Available as technical Report 190, Computer Systems Laboratory, Stanford University, June 1980.

[27] N. Wirth, Modula: A language for modular multiprogramming, *Software Practice and Experience* 7(1) (1977) 3-35.