Script: A Communication Abstraction Mechanism

Nissim Francez † and Brent Hailpern

IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

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* in order to participate in the action of the script. The paper discusses the addition of scripts to the languages CSP and Ada, as well as to a shared-variable language with monitors.

CR Categories :

[D.1.3] programming techniques - concurrent programming.
[D.2.2] software engineering - tools and techniques.
[D.3.3] programming languages - language constructs.
[D.4.1] operating systems - process management

General Terms: communication primitives, abstraction mechanisms, concurrent programming.

Other Keywords: script, role, enrollment

I. Introduction

Abstraction mechanisms have been widely recognized as useful programming tools and have been incorporated into modern programming languages [1, 3, 9, 10, 13, 15]. 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,

---

† The first author is a WTVS on a sabbatical leave from the Technion, Haifa, Israel.

- *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.

In addition to 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 [1, 3, 5, 9, 10, 13, 15] support multiprocessing and distributed processing in one way or another. Every such language has a construct to support such communications; some of the constructs are slightly higher-level than others, but all of them can be considered low-level in that 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 towards such well-designed patterns exists already, for example "idioms" and their efficient implementation are discussed in [4].

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 not add functional power to the host language. Thus, every program expressed by means of such abstraction should be also expressible without it.

- The abstraction should be modular and the behavior of an instance of an abstraction should not depend on the context of use, except by means of a predefined interface mechanism. We chose "parameters passing" for our interface.

- The abstraction mechanism is biased towards models of disjoint processes (that is, no shared variables). Communication is achieved by some message passing actions or remote procedure calls.

- Some semantic issues are left open, to be decided when we have a better understanding of how the mechanism will be applied.

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 [6, 11]. These techniques, however, are interested in encapsulating a *single* communication, 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 II, 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 III, we present the examples in a Pascal-like syntax. In Section IV, one of the example scripts, broadcast, is developed in three host languages: CSP, Ada†, and a shared-memory language with monitors. Section V concludes with some discussion of future work.

---

† Ada is a registered trademark of the U.S. Department of Defense.

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. In the case that 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 in order to execute a script.

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 of 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.

As a typical example, we may consider a script implementing a scenario of (software) *broadcast.* 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 of the recipients and assigned to their corresponding data parameters.

The body of the script could hide the various broadcast strategies:

- a star-like pattern in which 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, upon receiving x from its parent role, transmits it to every one of its descendant roles (again with different orderings).

- others; see [12, 14] for a discussion of various broadcast patterns and their relative merits.

Sections III and IV 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 employ 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 database *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 by Korth [7].

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, as well as 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. The parameter passing modes are inherited from the host programming language, presumably the same modes as for procedure calls with data parameters.

We want to distinguish 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, 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 statement in Ada, which accepts an entry call from any potential caller. Francez [2] has proposed a similar extension of the CSP primitives.

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 non-deterministic.

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 of the roles are finished. An *immediate termination* will free each process as soon as 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.

Successive Activations

For the rest of this paper we assume that there is only one instance of a script. We could allow for multiple instances of a script in the same sense that Ada allows for multiple instances of a generic object. This would add no new power to the construct, and would permit the programmer to avoid coding the same script over and over again. For example, concurrent independent broadcasts could use separate instances of the same generic script.

Whichever conventions for initiation and termination are chosen, we assume that they meet a minimum requirement for the semantics of successive activations of a script. We call the collective activation of all the roles of a script a *performance.* Our minimum semantic requirement is that all of the roles of a given performance must terminate before a subsequent performance of the same script can begin. For example, consider a script with three rolls: p, q, and r. Assume, also, that there are six processes: A, B, C, D, E, F. In Figure 1 we see two consecutive performances of the script. Process D must wait for all of the processes of the first performance to finish their rolls before it can enroll, even though process A has completed its participation.

This rule takes on particular importance when one process enrolls to a given roll more than once. In terms of our broadcast example, consider two processes, A and B, which contain the (partner-named) enrollments shown in Figure 2. We assume that there are many recipient roles, called recipient_1, recipient_2, and so on. The semantics must guarantee the effect that u=x and y=v (assuming, of course, that no other processes are attempting to enroll in the script with these same roles).

Note that a delayed-initiation or delayed-termination policy would guarantee that the successive activations rule is met.

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.

Critical Role Set

For a performance of an instance of a script, it may not be necessary that all roles of the script be 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). In order 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. In case no such set is specified, it is taken to mean that the entire collection of roles is critical.

The use of critical role sets does, however, introduce a new 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. (The notion of a central administrator for a script, however, does not preserve our goal of not generating additional process when executing a script.) Alternatively, attempting to communicate with an unfilled role could return a distinguished value. Our database example will follow the latter solution.

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.

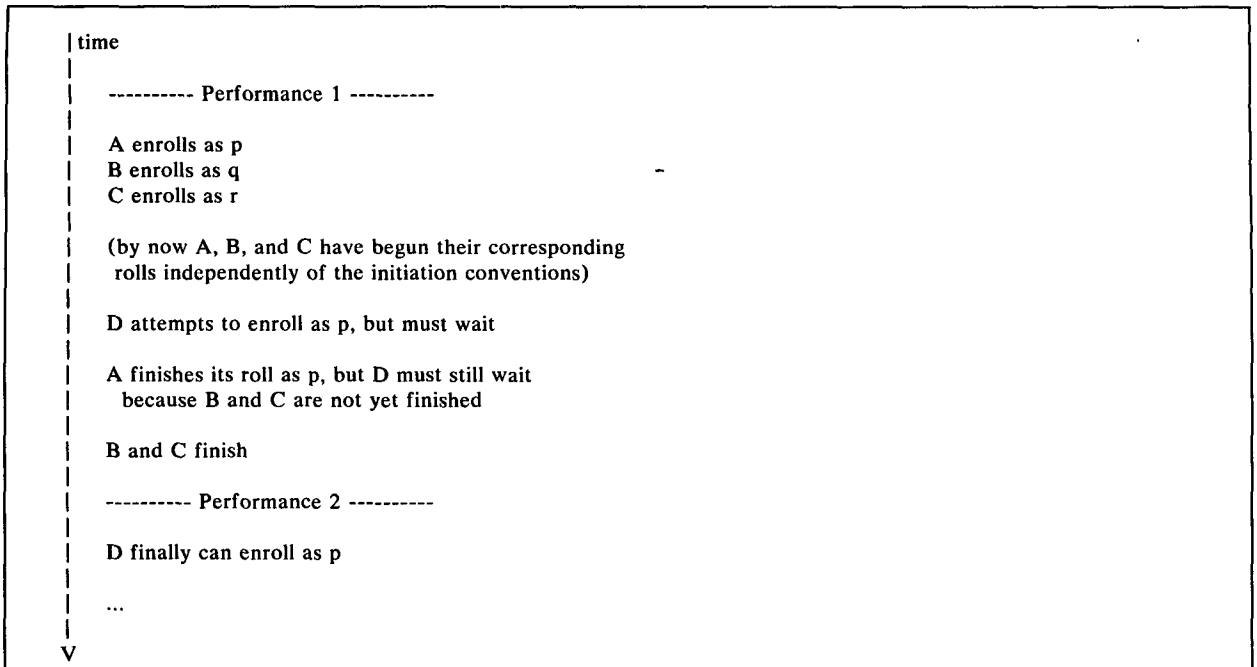

Figure 1. Consecutive Performances

```
in A:
        ENROLL IN broadcast AS transmitter(x)
                WITH B AS recipient_1; ...
        ENROLL IN broadcast AS transmitter(y)
                WITH B AS recipient_1; ...
in B:
        ENROLL IN broadcast AS recipient_1(u)
                WITH A AS transmitter; ...
        ENROLL IN IN broadcast AS recipient_1(v)
                WITH A AS transmitter; ...
```

Figure 2.

III. Sample Scripts

In this section we present three example scripts: star broadcast, pipeline broadcast, and a "one lock to read" database. Our language is Pascal 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**).

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 Figure 3. The broadcast script has one sender 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 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 of the recipients are available and not waiting for any other I/O operations. (The notation "ROLE recipient [i:1..5] (...)" 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 sender by

```
ENROLL IN broadcast AS sender(expression);
```

A process would enroll as the first recipient by

```
ENROLL IN broadcast AS recipient[1](variable);
```

```
SCRIPT star_broadcast;
    INITIATION: DELAYED;
    TERMINATION: DELAYED;

    ROLE sender (data : item);
    BEGIN
        SEND data TO recipient[1];
        SEND data TO recipient[2];
        SEND data TO recipient[3];
        SEND data TO recipient[4];
        SEND data TO recipient[5]
    END sender;

    ROLE recipient [i:1..5] (VAR data : item);
    BEGIN
        RECEIVE data FROM sender
    END recipient;

END star_broadcast;
```

Figure 3. Synchronized Star Broadcast

Pipeline Broadcast

Our second example, shown in Figure 4, 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 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.

Database

Our final example implements a distributed, replicated database 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 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 Figure 5.

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 II 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. (We make no claim that this termination function would be simple to implement without a central administrator for the script.)

IV. 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 examples given are intended to be existence proofs that such additions could be made without extending the base language in 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 of the options and features described above.

Scripts in CSP

CSP [5] imposes strict naming conventions, where in every communication both parties explicitly name each other. We,

```
SCRIPT pipe_broadcast;
    INITIATION: IMMEDIATE;
    TERMINATION: IMMEDIATE;

    ROLE sender (data : item);
    BEGIN
        SEND data TO recipient[1]
    END sender;

    ROLE recipient [i:1..5] (VAR data : item);
    BEGIN
        IF i=1 THEN
            RECEIVE data FROM sender
        ELSE
            RECEIVE data FROM recipient[i-1];

        IF i<5 THEN
            SEND data TO recipient[i+1]
    END recipient;

END pipe_broadcast;
```

Figure 4. Pipeline Broadcast

```
SCRIPT lock;
    INITIATION: IMMEDIATE;
    TERMINATION: IMMEDIATE;
    CRITICAL_ROLES: (manager[1..k], reader) OR manager[1..k], writer;

    ROLE manager [i:1..k] (VAR lock_table : lock_type);
    BEGIN
        DO ¬ (reader.terminated AND writer.terminated) →
            IF RECEIVE release(data, id) FROM reader →
                    lock_table.read_unlock(data, id)
            □ RECEIVE release(data, id) FROM writer →
                    lock_table.write_unlock(data, id)
            □ RECEIVE lock(data, id) FROM reader →
                    IF lock_table.able_to_read(data) THEN
                        BEGIN
                            lock_table.read_lock(data, id);
                            SEND granted TO reader
                        END
                    ELSE
                        SEND denied TO reader
            □ RECEIVE lock(data, id) FROM writer →
                    IF lock_table.able_to_write(data) THEN
                        BEGIN
                            lock_table.write_lock(data, id);
                            SEND granted TO writer
                        END
                    ELSE
                        SEND denied TO writer
            FI
        OD
    END manager;
```

Figure 5a. Database Lock Manager

therefore, adopt a restricted named-enrollment policy: each process, in addition to 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.

```
ROLE reader (id : process_id; data : object; request : (lock, release);
                    VAR status : (granted, denied));
VAR
    done : ARRAY [1..k] OF boolean;
    who : SET OF [1..k];
    reply : (granted, denied);
BEGIN
    IF request = release THEN
        BEGIN
            done := false;  { array assignment }
            DO □(i = 1..k)
                ¬ done[i]; SEND release(data, id) TO manager[i] →
                    done[i] := true
            OD
        END
    ELSE { request = lock }
        BEGIN
            who := [ ];
            done := false;
            DO □(i = 1..k)
                (who = [ ]) AND ¬ done[i]; SEND lock(data, id) TO manager[i] →
                    RECEIVE reply FROM manager[i];
                    done[i] := true;
                    IF reply = granted THEN who := who + [i]
            OD
            IF who <> [ ] THEN
                status := granted
            ELSE
                BEGIN
                    status := denied;
                    DO □(i = 1..k)
                        i IN who; SEND release(data, id) TO manager[i]→
                            who := who - [i]
                    OD
                END
        END
END reader;
```

Figure 5b. Database Lock Manager (continued)

The initiation policy will be that of immediate initiation, because CSP dose not have the ability to 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 6 shows a broadcast script in CSP.

Now consider a parallel command [... ‖ p ‖ ... ‖ q ‖ ...], 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; ...
```

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 bloc enrollment of an array of processes to an array of rolls.) 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. This raises several interesting problems about script specification, which we do not address in this paper.

```
    ROLE writer (id : process_id; data : object; request : (lock, release);
                        VAR status : (granted, denied));
    VAR
        done : ARRAY [1..k] OF boolean;
        who : SET OF [1..k];
        reply : (granted, denied);
    BEGIN
        IF request = release THEN
            BEGIN
                done := false;  { array assignment }
                DO □(i = 1..k)
                    ¬ done[i]; SEND release(data, id) TO manager[i] →
                        done[i] := true
                OD
            END
        ELSE { lock }
            BEGIN
                done := false;
                who := [ ];
                DO □(i = 1..k)
                    ¬ done[i]; SEND lock(data, id) TO manager[i] →
                        RECEIVE reply FROM manager[i];
                        done[i] := true;
                        IF reply = granted THEN who := who + [i] ELSE done := true
                OD;
                IF SIZE(who) = k THEN
                    status := granted
                ELSE
                    BEGIN
                        status := denied;
                        DO □(i = 1..k)
                            i IN who; SEND release(data, id) TO manager[i]→
                                who := who - [i]
                        OD
                    END
            END
    END writer;
END lock;
```

Figure 5c. Database Lock Manager (continued)

```
SCRIPT broadcast::
    [ ROLE transmitter (x: item)::
            VAR sent: ARRAY[1..5] OF boolean := 5*false;
            *[□(k=1,5) ¬sent[k];recipient[k]!x→sent[k]:=true]
      ||

        ROLE (i=1,5) recipient(y_i)::
            transmitter?y_i
    ].
```

Figure 6. Broadcast in CSP

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. Also, in order to ensure the semantics of successive activations, we will associate to each script instance 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 = [$p_1 \| ... \| p_n$] and a script instance s, with roles r_1, ..., r_m.

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

ENROLL IN s AS r(params) WITH [p_{i1} AS r_1;...p_{im} AS r_m],

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 replace by process name p_{ij} according to the correspondence specified in the enrollment.

   b. the actual parameters, params, substituted for the formal script data parameters (as in call-by-reference semantics).

   c. every communication command tagged with the script instance name, for example, r_1!(x+y) becomes p_{i1}!s(x+y) and r_2?u becomes p_{i2}?s(u) .

The process p_s will be concurrently composed with the enrolling processes, and is defined in Figure 7. 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 by Francez [2].

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.

Scripts in Ada

One feature that differentiates 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 that of 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 8 shows a broadcast script in Ada. The script consists of six rolls: a sender and five recipients. The recipients all share the same code, so a template (roll type) is used. Note that the script body contains a "reverse broadcast" in that 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.

Translation into Ada

To show that the script is not more powerful than Ada, we give the following translation to Ada (without scripts). Each role becomes a task and one additional task is created to coordinate the enrollments. Because each role is represented 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 9 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 Lamb and Hilfinger's procedure per role translation to provide procedure variables in Ada [8]).

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

Rules of Translation: Replace every enrollment within 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);
```

Replace each role r_i of script s by a task s_r_i. The role r_i has the form shown in Figure 10. Task s_r_i has all of the entries of r_i plus two additional entries

```
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
           ].
```

Figure 7. CSP Supervisor

```
SCRIPT broadcast IS
    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;
                completed := completed + 1;
            END;
        END LOOP;
    END sender;

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

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

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

Figure 8. Broadcast in Ada

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

Task s_r_i has all of the local variables of r_i, without initialization, as well as one new local variable, v1', v2', v3', for each formal parameter of the start/stop entry calls, v1, v2, v3. Let B bet the body of r_i. The body of s_r_i is shown in Figure 11. In the body B, occurrences of v1, v2, v3 are replaced by v1', v2', and v3'. Calls to role entry r_j.x(y,z) become calls to task entry s_r_j.x(y,z). 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.

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.

```
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);
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 = (1..m => true) =>
                  done := (1..m => false);
                  ready := (1..m => true);
      END SELECT;
   END LOOP;
END s_supervisor;
```

Figure 9. Ada Supervisor

```
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 roll
END r_i;
```

Figure 10. 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;
```

Figure 11. Ada Role (after translation)

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 of 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 that of Figure 4, is shown in Figure 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 mail box. 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.

V. Future Work

More work needs to be done with scripts to explore their potential for simplifying the programming of concurrent systems. In particular, we need to determine if there is a unifying structure that encompasses the various enrollment, initiation, and termination strategies presented in this paper. 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. Another generalization would be recursive scripts, where a roll could enroll in its own script. A related notion is nested enrollment, where one roll can enroll in some other script.

We also intend to explore issues of specification and verification of concurrent programs using scripts. In particular, we believe scripts will simplify the specification of communication subsystems and make the verification of such systems more practical.

Acknowledgments

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 paper.

VI. References

1] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering,* **SE-1** (2): 199-207, June 1975.

2] N. Francez. Extended naming conventions for communicating processes. *Ninth Annual ACM Symposium on Principles of Programming Languages* (Albuquerque), pages 40-45, January 1982.

3] P. Hilfinger, G. Feldman, R. Fitzgerald, I. Kimura, R. L. London, KVS Prasad, VR Prasad, J. Rosenberg, M. Shaw, and W. A. Wulf (editor). An informal definition of Alphard (preliminary). Technical report CMU-CS-78-105, Carnegie-Mellon University, February 1978.

4] P. N. Hilfinger. Implementation strategies for Ada tasking idioms. *Proceedings of the ACM-AdaTEC Conference on Ada* (Arlington), October 1982.

5] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM,* **21** (8): 666-677, August 1978.

6] M. Joseph. Schemes for communication. Technical report CMU-CS-81-122, Carnegie-Mellon University, June 1981.

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

8] D. A. Lamb and P. N. Hilfinger. Simulation of procedure variables using Ada tasks. *IEEE Transactions on Software Engineering,* **SE-9** (1):13-15, January 1983.

9] 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,* volume 114. Springer-Verlag, 1981.

10] J. G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual (version 5.0). CSL-79-3, Xerox Palo Alto Research Center, April 1979.

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

12] J. Skansholm. Multicast and synchronization in distributed systems. Research report, Department of Computer Science, University of Goteborg, 1981.

13] United States Department of Defense. *Reference Manual for the Ada Programming Language.* ACM-AdaTEC, July 1982.

```
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 put;
    BEGIN
        status := empty;
    END mailbox;


    ROLE sender (data : item);
        VAR r : integer;
    BEGIN
        FOR r := 1 TO 5 DO
            recipient[r].mbox.put(data);
    END sender;

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

Figure 12. Mailbox Broadcast

14] 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.

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