Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCS-83-4

1983-06-01

Multifaceted Distributed Systems Specification Using Processes and Event Synchronization

Gruia-Catalin Roman and Mark S. Day

A new approach to modelling distributed systems is presented. It uses sequential processes and event synchronization as the major building blocks and is able to capture the functionality, architecture, scheduling policies, and performance attributes of a distributed system. The approach is meant to provide the foundation for a uniform incremental strategy for verifying both logical and performance properties of distributed systems. In addition, this approach draws together work on performance evaluation, resource allocation, and verification of concurrent processes by reducing some problems from the first two areas to equivalent problems in the third. A language called CSPS (an extension... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

Multifaceted Distributed Systems Specification Using Processes and Event Synchronization

Gruia-Catalin Roman and Mark S. Day

Complete Abstract:

A new approach to modelling distributed systems is presented. It uses sequential processes and event synchronization as the major building blocks and is able to capture the functionality, architecture, scheduling policies, and performance attributes of a distributed system. The approach is meant to provide the foundation for a uniform incremental strategy for verifying both logical and performance properties of distributed systems. In addition, this approach draws together work on performance evaluation, resource allocation, and verification of concurrent processes by reducing some problems from the first two areas to equivalent problems in the third. A language called CSPS (an extension of Hoare's CSP) is used in the illustration of the approach. Employing CSP as a base allows modelled system to be verified using techniques already developed for verifying CSP programs.

MULTIFACETED DISTRIBUTED SYSTEMS
SPECIFICATION USING PROCESSES AND
EVENT SYNCHRONIZATION

Gruia-Catalin Roman

Mark S. Day

WUCS-83-4

June 1983

Department of Computer Science
Washington University
St. Louis, Missouri 63130

ABSTRACT

A new approach to modelling distributed systems is presented. It uses sequential processes and event synchronization as the major building blocks and is able to capture the functionality, architecture, scheduling policies, and performance attributes of a distributed system. The approach is meant to provide the foundation for a uniform incremental strategy for verifying both logical and performance properties of distributed systems. In addition, this approach draws together work on performance evaluation, resource allocation, and verification of concurrent processes by reducing some problems from the first two areas to equivalent problems in the third. A language called CSPS (an extension of Hoare's CSP) is used in the illustration of the approach. Employing CSP as a base allows modelled systems to be verified using techniques already developed for verifying CSP programs.

INTRODUCTION

A distributed system is an aggregate of hardware and software entities whose interactions and relationships may change with time. During its development the designer must be able to formulate and answer questions regarding its logical correctness, the impact of various faults and failures, response time, communication delay, hardware selection and utilization, scheduling policies, etc. This paper reports on a modelling technique that enhances the designer's ability to approach these issues.

Two main goals have influenced the nature of the models being employed. The first goal is to establish a foundation for a uniform incremental approach to verifying both logical and performance properties of distributed systems. The second goal is to bring together work on performance evaluation, resource allocation, and verification of concurrent processes by reducing some problems from the first two areas to equivalent problems in the third.

The following steps have been accomplished so far in the pursuit of this research:

(1) the selection of a minimal set of modelling concepts,
(2) the definition of the modelling technique,
(3) the selection of a tentative notation scheme,
(4) the sketching of an incremental proof technique, and
(5) the exercising of modelling, notation and proof techniques on a simple system.

Ongoing work will be outlined in the end of the paper.

We chose concurrent processes and event synchronization as the basic concepts of the model. They are simple, intuitive and, as shown in this paper, sufficient to carry out the modelling task. Furthermore, the similarity of the concepts employed by Hoare in the definition of CSP (process and I/O) [HOAR78] enables a wealth of formal results regarding CSP to apply directly to the models being built using our approach.

The models are called virtual systems and represent either abstractions of existing systems or definitions of proposed systems. A virtual system captures the functionality, architecture, scheduling policies and performance attributes of the system it models. This is accomplished by structuring the virtual system in terms of four communities of communicating processes that are related to each other via event synchronization rules. Communication between processes within a single community is also accomplished by a set of event synchronization rules.

In this paper, the virtual system is specified by a means of a language called CSPS (Communicating Sequential Processes with Synchronization). An extension of Hoare's CSP [HOAR78], CSPS allows synchronization between multiple processes in addition to the I/O primitives of CSP. Because of the similarity between CSPS and CSP, the verification techniques of [APT80, LEVI81] (with slight extensions) are still useful with CSPS. It should be noted, however, that we view CSPS

only as an experimental notation scheme that allows one to exercise the modelling concepts prior to the development of a distributed system design language. Clearly, in the design of such a language human engineering and design automation would play a critical role. These issues, although important in themselves, are ignored here in favor of presenting the modelling approach.

The following sections provide definitions for the concepts involved in the modelling process; review CSP; introduce the CSPS notation; explain the structure of the virtual system and motivation for that structure; and present in some detail the definition of functionality, architecture, scheduling, and performance for a virtual system through a simple ongoing example. The development of the example is accompanied by an outline of the incremental proof strategy used to verify conformance to various types of functional (e.g., logical correctness) and non-functional (e.g., fault tolerance and communication delay) system requirements.

FORMAL FOUNDATION

This section introduces the fundamental concepts required for the definition of a virtual system. They are processes, event synchronization, communities of processes, composite processes, and equivalence of processes. Each of these will be defined and tersely described below.

Process

A process q is defined as a 5-tuple $(V, V0, N, E, w)$ where V is a set of states, $V0$ is a set of initial states, N is a set of event labels, E is a set of events (labelled transitions between states), and w is the null event.

        q = (V, V0, N, E, w)
            E subset.of (N x V x V)
            w member.of N
            !A e member.of N: e ≠ w ==> !E (e, v1, v2) member.of E
            !A v member.of V: (w, v, v) member.of E

(NOTE: "!A x:", "?E y:" and "!E z:" should be read "for all x", "there is some y" and "there is a unique z", respectively.)

Events are named pairs of states. The non-w event label e can be used to refer to the unique event triplet $(e, v1, v2)$; alternately, e can be considered to be a partial function over the set of states V where

        e(v) = v2   for v = v1
             = undefined   otherwise

In contrast, the null event w, which marks that no event took place at a particular point in time, is defined for all possible states. Its usefulness will be apparent when event synchronization between processes is considered.

Two processes are isomorphic if they are identical except for a renaming of states and event labels.

Event Synchronization

We propose to use event synchronization as the sole means of defining communication between concurrent processes. When two or more events (in distinct processes) are synchronized, the effect is that of simultaneous occurrence of all the synchronized events. The set of synchronization rules for a set of processes

        Q = { qi = (Vi V0i Ni Ei w) for i=1, ..., n}

takes the form

        C subset.of N1 x N2 x ... x Nn

In each synchronization rule c in C, the non-w events named are

synchronized. Here the need for the null event w is apparent: it serves
as a placeholder in the n-tuple for a process which has no event
synchronized with the non-w events within the n-tuple. Each rule c can be
used as a vector function:

c(v1 v2 ... vn) = (e1 e2 ... en)[(v1 v2 ... vn)]
              = (e1(v1) e2(v2) ... en(vn))

For the sake of a uniform treatment of both synchronized and
unsynchronized events, C includes rules for all unsynchronized events as
well. These rules take the form

(w, ..., w, e, w, ..., w)

The consequence of this is that C must include each event of qi in at
least one rule. However, each event may appear in several rules.

Consider events (e1, v1, v1'), (e2, v2, v2'), (e3, v3, v3') in
processes q1, q2, and q3 respectively. One set of synchronization rules C
over these processes might be:

        C = {(e1 w w)
             (e1 e2 w)
             (w e2 e3)
             (e1 e2 e3)
             (w w w)}

These particular rules allow event e1 to occur on its own, simultaneously
with e2, or simultaneously with both e2 and e3. On the other hand, the
rules do not allow event e2 to occur on its own: it must be synchronized
with e1, e3, or both. The last rule is the null vector, which is a
synchronization rule of little practical use since its effect is that
nothing happens in the group of processes.

Community of Processes

A community of processes K is a set of processes
Q = {q1, q2, ..., qn} and a set of synchronization rules C over Q.

        K = (Q,C)

The community has an overall state which is the cross product of the
state of all the processes in Q. For each process in Q, the events which
can take place from the current state (i.e. for which the current state
is the first element of the ordered pair of states) are called enabled. A
rule c member.of C for which all non-w events are enabled is called
applicable. To change the state of the community, a single applicable
rule is chosen and all the non-w events in that rule take place
simultaneously.

The community of processes can be regarded as a group of
communicating processes with the communication protocol determined by the
rules in C. Hoare's CSP [HOAR78] describes such a community of processes.

(For the reader unfamiliar with CSP, the next section outlines the syntax and semantics of the language.) In CSP, the parallel execution of defined processes has implicit synchronization at the beginning of execution, the end of execution, and for each matching I/O pair. For example:

```
[INPUT || OUTPUT]
INPUT::  *[c: character; true     --> get(c) OUTPUT!c]
OUTPUT:: *[d: character; INPUT?d --> put(d)]
```

The process INPUT continually uses the function 'get' to produce characters which are sent, one at a time, to OUTPUT, which then uses the function 'put' to get rid of each character.

```
                        get(c)
                  ┌──────────────┐
        Instart   │  OUTPUT!c  ▼   Inend
    (A)────+───► (B)◄───+────(C)────+───►(D)
            :           :           :
            :           :           :
    (E)────+───► (F)────+───►(G)────+───►(H)
    Outstart ▲    INPUT?d     │ Outend
             └──────────────┘
                   put(d)
```
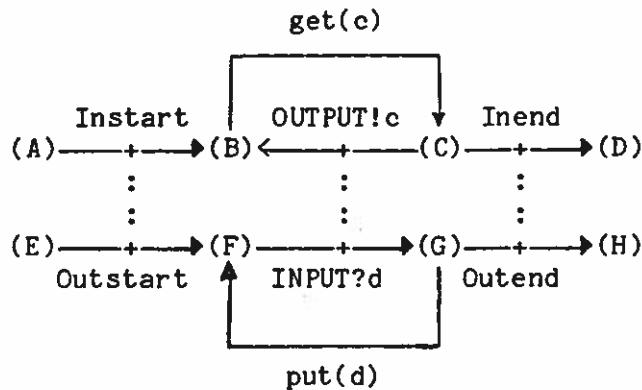
The CSP program can be described as the community of two processes shown above. 'A' through 'D' are states of the process INPUT, 'E' through 'H' are states of the process OUTPUT. Arrows connecting states are events; vertical lines connecting events indicate that those events are synchronized.

Composite Process

Given a community of processes K, the construction K' (defined below) or any process isomorphic to K' is called a composite process of K. Given the community of processes K = (Q,C), Q = {q1, q2, ..., qn} with qi = (Vi, VOi, Ni, Ei, w), K' is defined as follows:

```
K' = (V, VO, N, E, w)
   V  = V1 x V2  x ... x Vn
   VO = VO1 x VO2 x ... x VOn
   N  = N1  x N2  x ... x Nn
   E subset.of (N x V x V)
      ((e1 e2 ... en) (v1 v2 ... vn) (v1' v2' ... vn')) member.of E
      iff  (e1 e2 ... en) member.of C
           &
           (e1 e2 ... en)[(v1 v2 ... vn)] = (v1' v2' ... vn')
```

The notion of composite process is required in order to understand the concept of functional equivalence between two alternate system designs. Other uses for it will become apparent later.
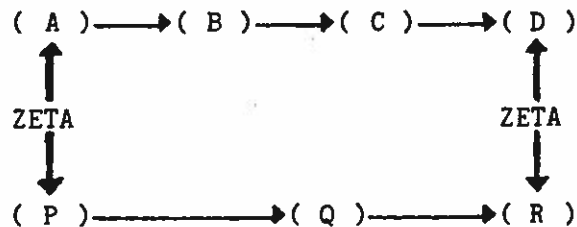
Equivalence

Equivalence of two processes is a weaker relationship than isomorphism: only a portion of the states of each process need map one-to-one to states in the other process.

Two processes $q1 = (V1, VO1, N1, E1, w)$, $q2 = (V2, VO2, N2, E2, w)$ are called _equivalent_ with respect to some mapping ZETA iff

1) ZETA: $V1 \longrightarrow V2$ is a partial one-to-one function

2) !A $(v1, v2)$ with $ZETA(v1) = v2$:
        v1' immediately reachable from v1
            ==> !E v2' immediately reachable from v2 & $ZETA(v1') = v2'$

In this definition, the phrase "v' immediately reachable from v" means that there is some finite sequence of events in the process starting from state v and ending in state v' where ZETA is defined for v and v', but for no intervening state.

```
( A )──────►( B )──────►( C )──────►( D )
  ▲                                   ▲
  │                                   │
ZETA                                ZETA
  │                                   │
  ▼                                   ▼
( P )──────────────►( Q )──────────►( R )
```

In the example above, ZETA maps one-to-one between states A and P, and between states D and R. State D is immediately reachable from state A, and state R is immediately reachable from state P. By applying the definition, we can see that the top process is equivalent to the bottom process under the function ZETA.

CSP may be shown to have the same expressive power as CSPS under this particular definition of equivalence but not under stricter ones.

CSP REVIEW

CSP is a language expressing concurrency of processes and communication between them. Hoare's original paper [HOAR78] gives a full grammar for the language; this section will simply present an outline of the subset used in this paper.

The command

    [P1 ¦¦ P2 ¦¦ ... ¦¦ Pn]

expresses concurrent execution of processes P1, P2, ..., Pn. As described by the syntax

    Pi :: S1; S2; ...; Sm

each process Pi consists of some sequence of statements S1, S2, ..., Sm. No statement in process Pi contains variables subject to change in process Pj for $i \neq j$. The assignment operator is ':='; input/output primitives are '!' (send) and '?' (receive), as in P5!x ('send x to process P5') and FOO?A(z,k) ('receive A(z,k) from process FOO'). For both assignment and I/O, the 'target' variable into which data is being transfered must 'match' the value being transferred in. A simple variable is a name with no list following, and matches any value of its type. A structured variable is a name (called a constructor) followed by a list of variable names in parentheses. A structured variable matches a structured value if they have the same constructor and each simple variable in the list matches its corresponding simple value.

I/O transfers only take place when one process names another as its destination for output, and that process names the first as its source for input. Such a situation is called 'matching', but is different from the variable-to-value matching previously described. If one process is ready to send or receive before its target is ready, it waits for the other process.

Guarded commands control execution of statements. When a guard evaluates to 'true', the guarded statement may be executed. If it is 'false', the guarded statement will not be executed. Guarded commands are used to describe non-deterministic selection of one of several alternatives. The symbol ▯ separates these guarded commands, each of which takes the form

    b --> S

with b a Boolean expression and S some sequence of statements. In a guarded alternative command such as

    [b1 --> S1 ▯ b2 --> S2 ▯ b3 --> S3]

any one of the statements Si may be executed if its guard bi is true. If all the bi are false, the statement fails. A guarded repetitive command is similar,

```
*[ b1 --> S1 ] b2 --> S2 ] b3 --> S3 ]
```

but the effect is that of repetitively executing a guarded alternative command until all the guards are false, which causes the command to terminate.

One of the most important features of CSP is that the last element of a guard can be an I/O statement such as Pj?x or Pj!x. This primitive evaluates to 'true' if the I/O exchange takes place, and 'false' if the target process has terminated. Otherwise, the guard waits: if another guard is entirely true, its guarded command may be executed; but if all other guards are entirely false, the overall command does not terminate until the status of the I/O command is resolved.

EXTENDING CSP TO CSPS

Throughout the paper, communities of processes used in examples are defined in a language called CSPS, 'CSP with synchronization.' CSPS includes primitives that allow definition of synchronization between events (i.e. statements) in multiple sequential processes. This explicit synchronization is in addition to the I/O primitives of CSP.

This section introduces the notation for explicit synchronization between processes (implicit synchronization occures between the matching I/O statements). CSPS provides for synchronization between two or more processes, which is needed since the I/O primitives of CSP only synchronize two processes (sender and receiver) at a time. Throughout the paper, the I/O primitives will be used to define communication among the processes within one of several process communities making up the virtual system, while synchronization primitives will define interactions between these communities.

Statements of CSP correspond to events in the state-event model; the extension of CSP is a way of showing the synchronization of these statements. As defined in the previous section, event synchronization is controlled by a set of rules. To show that a particular statement or guard is synchronized by a particular rule, the statement is written as a synchronized command or synchronized guard, as specified by the grammar below:

```
<synchronized command> ::= <simple command> '$' <rule>
  <synchronized guard> ::= <guard> '$' <rule>
             <rule> ::= <rulename> | '(' <rulelist> ')'
         <rulelist> ::= <compoundrule> | <compoundrule> <OR> <rulelist>
     <compoundrule> ::= <rulename> | <rulename> <AND> <compoundrule>
             <OR> ::= '|'
            <AND> ::= '.'
```

Both <simple command> and <guard> are defined in the grammar for CSP [HOAR78].

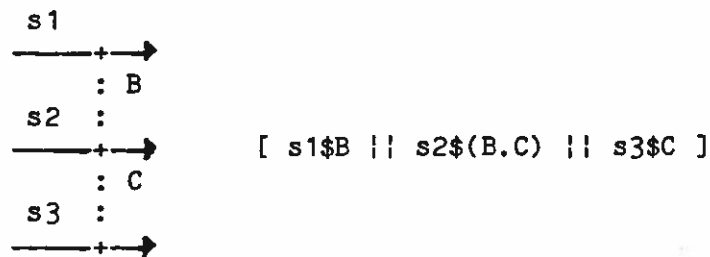Events which are synchronized will all be synchronized by the same name. For example, the statements s1, s2, s3 shown (each belonging to a different process) are synchronized by rule A:

```
      s1
    ———+——▶
         : A
      s2 :
    ———+——▶          [ s1$A || s2$A || s3$A ]
         :
      s3 :
    ———+——▶
```
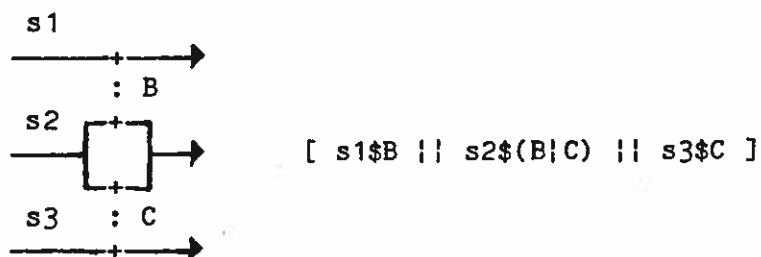
An alternate representation of the previous example might use the two rules B and C instead of the single rule A:

```
s1
————+——➤
      : B
s2   :
————+——➤
      : C          [ s1$B || s2$(B.C) || s3$C ]
s3   :
————+——➤
```

This is identical to the example using only rule A. There is no 'choice'
for e2 as to its synchronization: it takes place synchronized to e1 and
e3.

A 'choice' can be represented by joining rules by 'OR':

```
s1
————+——————➤
        : B
s2   ┌—+—┐
—————|   |——➤      [ s1$B || s2$(B|C) || s3$C ]
     └—+—┘
s3   : C
————+——————➤
```

In the example above, only two out of the three events may be synchronized
at one time. If rule B is applied, events s1 and s2 take place; if rule
C is applied, events s2 and s3 take place. Only one rule is applied at a
time, so it is impossible for all 3 events to take place even if they are
all enabled. The notation and diagram are not intended to imply any
preference or ordering of the two rules if all 3 events are enabled at the
same time: the choice of rule is non-deterministic.

It is possible for an event to be subject to some synchronization
rule, but also be free to proceed on its own. Since this means that it is
'synchronized to nothing else', the special rulename '-' is used to
express this. For example,

```
s1   ┌———┐
—————|   |——➤
     └—+—┘
      : A          [ s1$(A|-) || s2$A ]
s2   :
————+———➤
```
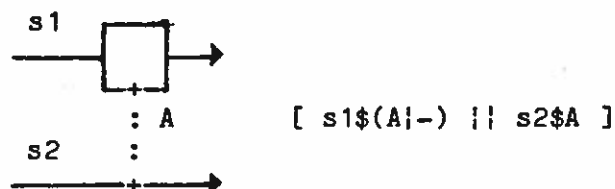
VIRTUAL SYSTEM

We view a distributed system as a collection of application software modules, allocated over hardware components. This allocation, generally the responsibility of some operating system, may be static or dynamic: in a dynamic allocation, the mapping between software and hardware changes with time; in a static allocation, this mapping is constant. In this view, the system behavior is determined primarily by the software. Parts of the system other than software can affect its performance, however. For example, faults in the hardware may limit system capabilities, or the interaction of software and hardware in a particular allocation may degrade system performance. The workload, which is determined by the environment with which the system interacts, also affects system behavior. A designer must consider all these factors when proposing or analyzing a distributed system and the model he employs ought to have the expressive and analytical power needed to address them.

The modelling approach proposed here involves the building of a virtual system, a model whose scope covers, to a very large extent, the issues listed above. A virtual system consists of six components each abstracting some aspect of a distributed system. The functionality is an abstraction of the processes which carry out the system function (e.g. the applications software). The architecture captures the overall hardware organization distribution of the system. The scheduler, together with the allocation, defines the relationship between functionality and architecture, and the changes which the relationship undergoes with time. This concept, while related to the notion of partition present in [ESTR78] and [MARI79], is much more general covering static and dynamic allocation of functions (in the functionality) to processors (in the architecture) as well as the allocation of time and space on an individual processor to the functions associated with it. The performance specification is an abstraction of both workload characteristics (the environment model is an integral part of the virtual system) and measurement probes. The performance specification may be used to explicitly state the assumptions made by designers regarding the characteristics of the environment and of the system components to be utilized in the realization of the system. The performance specification is coordinated with the rest of the model via the instrumentation.

Formally, the virtual system is defined in terms of processes and synchronization rules. The definition is given below.

VS = (FUNC, ARCH, SCHED, ALC, PERF, INS)

FUNC = (FQ, FC)
    the functionality is defined as a community of processes
    where the processes in FQ are called functions and the
    synchronization rules set FC states the coordination
    taking place between the functions in FQ.

ARCH = (AQ, AC)
> the architecture is defined as a community of processes
> where the processes in AQ are called processors and the
> synchronization rules set AC states the coordination
> taking place between the processors in AQ.

SCHED = (SQ, SC)
> the scheduler consists of a community of processes
> where the processes in SQ are called schedules and the
> synchronization rules set SC states the coordination
> taking place between the schedules in SQ. Together with
> ALC these define the association between processes in the
> functionality and processors in the architecture.

ALC
> the allocation is defined as a set of synchronization rules
> over FQ, AQ and SQ, whose purpose is to capture the relation
> between functions and processors and the changes it exhibits.

PERF = (PQ, PC)
> the performance specification consists of a community
> of processes where the processes in PQ are called actors
> and the synchronization rules set PC states the coordination
> taking place between the actors in PQ. Together with INS,
> these define the performance measurement and behavior
> modification due to performance aspects for the virtual system.

INS
> the instrumentation is defined as a set of synchronization
> rules over FQ, AQ, SQ and PQ, whose purpose is to exercise
> control over the degree of non-determinism present in the
> virtual system and to define the computation rules for
> relevant performance attributes.

The next four sections will expand on the motivation behind the structural components of the virtual system model and will demonstrate how questions of logical correctness, fault tolerance, and performance may be addressed in a systematic manner through the use of program verification techniques. This approach promises to reduce resource allocation and performance problems to equivalent program correctness problems.

FUNCTIONALITY

This section continues the discussion of system functionality and starts the development of a simple example of a virtual system--an expanded producer/consumer type problem. Its functionality is stated in CSPS and its logical correctness is formally demonstrated. The result will be used in subsequent proofs of the virtual system.

The functionality is a closed community of functions that communicate with each other. It models the system, its operating environment and the interactions between them. (This particular work makes no distinction between system functions and environment functions, but such a distinction could be made.) The functionality represents the requirements for the application software. They are given by means of an operational model whose structural properties are relevant only when considered in relation to a particular architecture and scheduler, as shown two sections later. If, for instance, the functionality is defined in terms of three functions (f1, f2. f3) but the whole system runs on a single processor, any piece of software that exhibits the same input/output behavior as the functionality would be acceptable. If, however, f1 resides on processor p1 while f2 and f3 reside on p2, f1 represents the requirements for the software running on p1 and f2 together with f3 for the software on p2. Furthermore, any other equivalent group of functions may be substituted for the combination f2/f3 (or f1, respectively) without actually changing the requirements. This is not the case if the functionality is reformulated by replacing f1 and f2 with some new function f12—the separation between f1 and f2/f3 is a relevant structural property that must be also present in the software while the separation of f2 and f3 is only for purposes of presentation or analysis.

Although the designer choses to break down the functionality in one particular way in order to take best advantage of the available or postulated processors, there are many aspects of the functionality that may be evaluated by considering the functionality alone. We can use the methods of [APT80] to prove logical correctness with respect to some externally stated criteria, freedom from behavior anomalies such as deadlock, and some primitive performance characteristics (e.g., relative frequency of execution of certain functions for some class of environmental inputs). Such proofs may be used later to incrementally prove certain properties of the system as a whole.

While the use of concurrent processes (i.e., functions) is clearly adequate for defining the functionality of a distributed system, the use of event synchronization as the only mechanism for modelling communication may be questioned by some of the readers. Preliminary results indicate, however, that event synchronization suffices to model a large spectrum of communication protocols. Moreover, it enables the designer to encapsulate the communication protocol into functions dedicated to this purpose thus limiting the number of places where changes are required when altering the protocol definition.

To start the example let us consider a producer P which produces a value and passes it to a link L; the link L forwards the value to a consumer C; the consumer C receives the value from link L and consumes

it.   This functionality can be described as

$$[P \; || \; L \; || \; C]$$

```
P:: *[ true --> produce(x); L!x ]
L:: *[ true --> P?y; C!y ]
C:: *[ true --> L?z; consume(z) ]
```

In this example, logical correctness is expressed by the fact that the value consumed by C is the same as the value produced by P, i.e. that values are consumed in the same order as they are produced. To express this formally, we associate auxiliary variables i and $X(i)$ with the sending of the produced value, and associate j and $Z(j)$ with the receipt of a value to be consumed. P and C then become

```
P:: i:=0; *[ true --> produce(x)
                       < L!x; i:=i+1; X(i):=x >
           ]
C:: j:=0; *[ true --> < L?z; j:=j+1; Z(j):=z >
                       consume(z)
           ]
```

The auxiliary variables are associated with I/O statements and will be used to prove cooperation between the proofs associated with the individual processes. (Note: auxiliary variables are variables added to concurrent programs in order to assist in the development of proofs--they are not allowed to alter the computation carried out by the programs; bracketed sections, i.e., "<...>", are introduced for the purpose of treating a program statement and several assignments to auxiliary variables as a single atomic action as far as the proof is concerned.)

The logical correctness of the functionality may be expressed now by formulating the following global invariant in terms of the auxiliary variables introduced earlier:

$$!A \; n: n <= j ==> X(n) = Z(n)$$

(Note: a global invariant is a predicate that holds true in between any two atomic actions--statement or bracketed section--of any of the processes considered.) Unfortunately, there has been no relationship established between x and z; this version of the invariant cannot be proven as a postassertion of L, which contains neither x nor z. To build a provable invariant, the auxiliary variables k, 1, and $Y(k)$ are introduced into L.

```
L:: k:=0; 1:=0; *[ true --> < P?y; k:=k+1; Y(k):=y >
                             < C!y; 1:=1+1 >
                ]
```

With these new variables, the invariant can be specified as

```
I = { i=k & !A n: (0 < n <= k) ==> X(n) = Y(n)
    & l=j & !A m: (0 < m <= j) ==> Y(m) = Z(m)
    & j <= k
    }
```

The first line specifies that P sends a value to L simultaneously with L receiving a value from P, and that the value received is identical to the value sent. The second line places similar constraints on the transmission from L to C. The third line specifies that transmission from L to C always lags behind transmission from P to L. This invariant implies the simpler one devised before.

The verification technique, borrowed from [APT80], is one of incremental proofs: the preassertions and postassertions of individual processes are verified separately, then for each matching I/O pair a proof of cooperation is carried out for assertions and the invariant.

The proofs for the individual processes requires the establishment of an appropriate set of assertions. The assertions for the processes P, L and C are shown in braces in the annotated program below. The proofs of these assertions for the separate processes are omitted; they are quite simple.

```
P:: i:=0        *[ {true}
                    true --> produce(x)
                    {true}
                    <L!x; i:=i+1; X(i):=x>
                    {X(i)=x}
                 ]

L:: k:=0 l:=0 *[ {l=k}
                    true --> <P?y; k:=k+1; Y(k):=y>
                    {Y(k)=y & k=l+1}
                    <C!y; l:=l+1>
                    {Y(k)=y & l=k}
                 ]

C:: j:=0        *[ {true}
                    true --> <L?z; j:=j+1; Z(j):=z>
                    {Z(j)=z}
                    consume(z)
                    {true}
                 ]
```

To finish the proof, cooperation must be shown between P and L, and between L and C. For cooperation, the postassertions of matching bracketed sections and the global invariant must be proven, given the preassertions of matching bracketed sections and the global invariant.

For cooperation between P and L, the sections are

```
P::  ... {true} < L!x; i:=i+1; X(i):=x > {X(i)=x} ...
L::  ... {l=k} < P?y; k:=k+1; Y(k):=y > {Y(k)=y & k=l+1}
```

So the proof can be summarized as

PREMISES

```
Preassertion: l=k
   Invariant: i=k & !A n: (0<n<=k) ==> X(n)=Y(n)
           & l=j & !A m: (0<m<=j) ==> Y(m)=Z(m)
           & j<=k
```

CONCLUSIONS

```
Postassertion: l=k'-1 & X(i')=x & Y(k')=y
    Invariant: i'=k' & !A n: (0<n<=k') ==> X(n)=Y(n)
           & l=j & !A m: (0<m<=j) ==> Y(m)=Z(m)
           & j<=k'
```

The second version of the invariant has been written in terms of
i'=i+1 and k'=k+1, the values of the auxiliary variables after the
communication has taken place. The invariants {l=j} and {!A m: (0<m<=j)
==> Y(m)=Z(m)} are unaffected by the statements within the matching
section and are trivially true. The postassertions {l=k'-1}, {Y(k')=y},
{X(i')=x} and the invariants {i'=k'}, {j<=k'} are all clearly true. Proof
of

```
!A n: (0<n<=k') ==> X(n)=Y(n)
```

requires proof only for n=k', since it is given that it is true for
(0<n<=k) or (0<n<=k'-1). The communication axiom [APT80] states that the
target variables of each side of a matching pair are equal:

```
{true} P?x || L!y {x=y}
(for P?x a statement in L, L!y a statement in P)
```

By this axiom, x=y. Since X(i')=x and i'=k', X(k')=x. Taken together
with Y(k')=y, clearly X(k')=Y(k').

The same method can be used to prove cooperation between L and C.
With these cooperation proofs completed, it is proven that the value
consumed is the same as the value produced.

ARCHITECTURE

This section continues the producer/consumer example by giving the
CSPS specification for a possible underlying architecture and by
discussing some of its behavior characteristics. Formal proofs are
omitted due to their similarity to earlier ones but the path they follow
is outlined. A discussion of the role the architecture plays in the
overall context of the virtual system is provided in the beginning of the
next section. -

The example for this section is communication via failing lines. An
originator O selects a value to be sent to destination D, and sends it on
one of the unreliable interconnections I1, I2. This architecture may be
modelled as

```
                [ O ¦¦ I1 ¦¦ I2 ¦¦ D ]

   O:: *[ true --> select(a)
                   [ I1!a --> skip ┃ I2!a --> skip ]
          ]
   I1:: *[ true --> [ O?b1 --> D!b1 ] ┃
           true --> *[ true --> skip ]
          ]
   I2:: *[ true --> [ O?b2 --> D!b2 ] ┃
           true --> *[ true --> skip ]
          ]
   D:: *[ I1?c --> discard(c) ┃ I2?c --> discard(c) ]
```

(Note: "skip" denotes a null statement.) The unreliable nature of the
lines has been expressed in each as two alternative commands. Execution
of the first alternative results in transmission of a value from O to D.
Execution of the second alternative results in permanent (non-recoverable)
failure.

The model of the originator O (also the destination D) indicates that
the status of the line is checked prior to selecting it. If this were not
so, the model of the originator might take the form:

```
   O:: *[ true --> select(a); [ true --> I1!a ┃
                                true --> I2!a
                              ]
          ]
```

The change is that the I/O statements are guarded by 'true', rather than
being the guards. This corresponds to selecting a line for transmission
before testing it to determine if it is functioning, and can result in
deadlock: if the first guarded command is taken and I1 has failed, I1
will never ask for input from O and the originator will wait forever to
send.

With the first specification, it can be proven that so long as at
least one of the lines has not failed, messages sent by O will be received
at D and the ordering of these messages will be preserved. Further, when
both lines have failed, no further message traffic will ever take place.

This is most easily shown by breaking the parallel lines apart and separating the proofs for [O || I1 || D] where O and D can only use I1 and [O || I2 || D] where O and D can only use I2. When separated, the proof of message passing and ordering follows the method used in the previous section for proving the properties of the functionality.

Once these origin-line-destination triplets have been separately proven correct, the messages must be shown to be interleaved, i.e. that if a message goes on one line, then no message goes on the other line, and all messages go on one line or the other. Consider adding the auxiliary variables k1, k2, X1, X2, and X into the process O:

```
O:: k1:=0; k2:=0 *[ true --> select(a)
                          [ <I1!a --> k1:=max(k1,k2)+1;
                                       X1(k1):=a> []
                            <I2!a --> k2:=max(k1,k2)+1;
                                       X2(k2):=a>
                          ]
                          X(max(k1,k2)):=a
                  ]
```

Then this interleaving can be expressed as

$$(!A \ n) \ 0 < n < max(k1,k2) ==> X1(n)=X(n) \ XOR \ X2(n) = X(n)$$

The last step of the proof is to show that after failure of both links, no further communication takes place. Consider introducing several boolean variables to indicate the status of the links and the originator:

```
O:: *[ true --> select(a); wait1:=true; wait2:=true;
               [ <I1!a --> wait1:=false> []
                 <I2!a --> wait2:=false>
               ]
     ]

I1:: fail1:=false; down1:=false;
     *[ true --> [ O?b1 --> D!b1 ] []
        true --> fail1:=true;
                 *[ true --> down1:=(fail1 & wait1)]
     ]

I2:: fail2:=false; down2:=false;
     *[ true --> [ O?b2 --> D!b2 ] []
        true --> fail2:=true;
                 *[ true --> down2:=(fail2 & wait2)]
     ]
```

When wait1 or wait2 is true, the originator O has selected a value and is waiting for one of the lines to be available. When down1 or down2 is true, the corresponding line I1 or I2 has failed. Fail1 and fail2 are used to flag the line as unavailable: the line is not <u>failed</u> until the line is unavailable <u>and</u> O is waiting to use that line. When both lines have failed, no further message transmission takes place. This can be expressed as
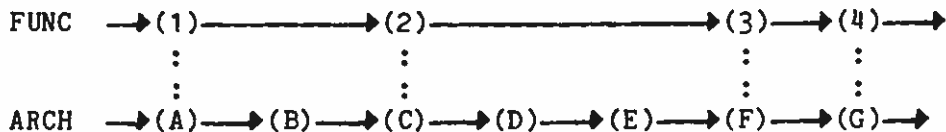
"IF down1 AND down2 BOTH BECOME TRUE THEN
    wait1 AND wait2 BOTH REMAIN TRUE FOREVER"

An array with steadily incremented index can serve as a type of diary for the state of the system, and the proof must demonstrate that for values of the index above the point where the links have failed, the system remains 'failed'. Alternately, the proof can be carried out using temporal logic.
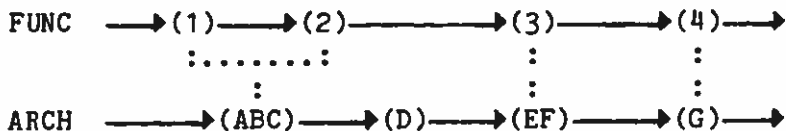
SCHEDULER AND ALLOCATION

This section elaborates the definition of the scheduler concept: its origin, its ability to model static and dynamic allocation of functions to processors as well as time and space allocation between the functions assigned to the same processor. Further clarification of the part played by architecture in the specification and analysis of a virtual system is also provided. The example is expanded to cover the scheduler.
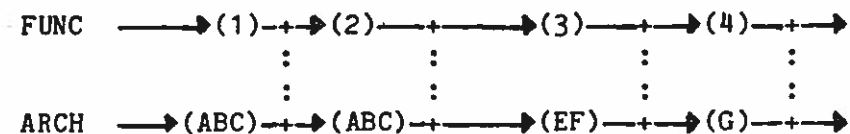
The scheduler defines the relationship (i.e. state dependencies) between the functionality and architecture of a system by defining synchronizations between them. To show how these synchronizations are determined, consider an APL function as a realization of functionality, and an APL machine (hardware interpreter) as a realization of architecture. Both functionality and architecture have states: for the function, the state depends on the value of local data and and the line number of the statement being interpreted; for the interpreter, the state depends on the program being interpreted, the interpreter's local data, and the program counter of the APL machine. In this common situation, the states of the function are mapped onto the states of the machine so that for every state of the function, there is a corresponding distinct state of the machine. There may be unmapped intervening states in the machine, but not in the function. This mapping of functionality and architecture is shown below:

```
FUNC  ──▶(1)─────────────▶(2)──────────────────▶(3)──▶(4)──▶
          :               :                      :     :
          :               :                      :     :
          :               :                      :     :
ARCH  ──▶(A)──▶(B)──▶(C)──▶(D)──▶(E)──▶(F)──▶(G)──▶
```

This work describes the architecture in terms of abstractions of the states of the machines involved, so some of these states are lumped together. Such an abstraction is shown below:

```
FUNC  ──▶(1)──▶(2)────────▶(3)──────▶(4)──▶
          :........:        :         :
              :             :         :
ARCH  ──────▶(ABC)──────▶(D)──▶(EF)──────▶(G)──▶
```

Next, the unmapped states of the abstract architecture are dropped from the description, and the state mappings are replaced by synchronization on the transitions.

```
FUNC  ──────▶(1)─+─▶(2)───+──────▶(3)───+──▶(4)──+─▶
               :        :           :         :
               :        :           :         :
ARCH  ──▶(ABC)─+─▶(ABC)─+──────▶(EF)───+──▶(G)──+─▶
```

This last transformation changes the view of the problem to one easily expressed in CSPS. (We may further simplify this description by dropping synchronizations that are irrelevant to a particular discussion.) The process of abstraction that was followed up to this point generalizes to the case where one deals with multiple functions and processors. These kind of models are of interest during system design and analysis when low

level architectural details are being ignored.

Next we will show how system level static and dynamic allocation and single processor time/space sharing may be modelled using the scheduler concept.

The static allocation of functions to processors does not require schedules. Thus, if no dynamic reallocation can occur in the system, the set of schedules is empty. Static allocation of a function to a processor can be determined from the synchronization of an event in the function to an event in the processor. In the example below, F1 and F2 are allocated to P1, while F3 is allocated to P2.
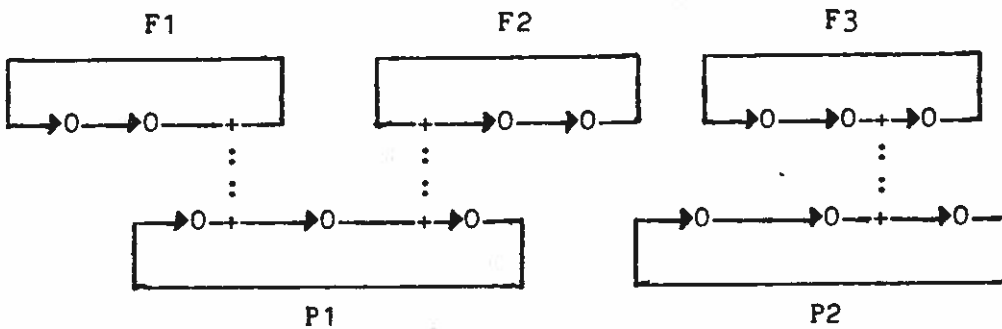
If an event in F3 were synchronized to an event in P1 as well, then the allocation would not be well-formed. All functions must be allocated to only one processor; i.e.,

(!A F) !E P to which F is allocated.

As an example, consider the static allocation of the the producer-consumer functionality to the origin-destination architecture, allocating link L to interconnection I1. The overall process set is written as

[P || L || C || O || I1 || I2 || D]

```
P:: *[ true --> produce(x)$h L!x ]
L:: *[ true$e1 --> P?y C!y ]
C:: *[ true --> L?z consume(z)$(g1|g2) ]

O:: *[ generate(a)$h [ I1!a --> skip [] I2!a --> skip ]]
I1:: *[ true$e1 --> [ O?b1 --> D!b1] [] true --> *[ true --> skip ]]
I2:: *[ true --> [ O?b1 --> D!b1 ] [] true --> *[ true --> skip ]]
D:: *[ I1?c --> use(c)$g1 [] I2?c --> use(c)$g2 ]
```

The production of x in P is synchronized to the generation of a in O, so P is allocated to O. Similarly, L is allocated to I1 and C is allocated to D. Note that this static scheduling does not allow I2 to be used at all.

Dynamic allocation can be introduced to allow the link L to use I1 until I1 fails, when it changes to using I2. For dynamic allocation, the synchronization of events in functions and processors is indirect, using the schedules. For example, a schedule can allow a function to be

synchronized to one of two different processes, as below:

```
             ┌────────────────────────────────────┐
FUNCTION     └─►O─+───────────────+─►O─┘
                 :    ($A ¦ $B)   :
                 : A              : B
                 :                :
  •              :                :
              ┌──┐            ┌──┐
              │  │            │  │
              │  ▼            │  ▼
SCHEDULE      │  O <─────────> O │
              │                  │
              └──+            └──+
                 :                :
                 : A              : B
                 :                :
           ┌─►O─+──►O─┐    ┌►O─+─►O─┐
PROCESSORS │         │    │        │
           └─────────┘    └────────┘
```
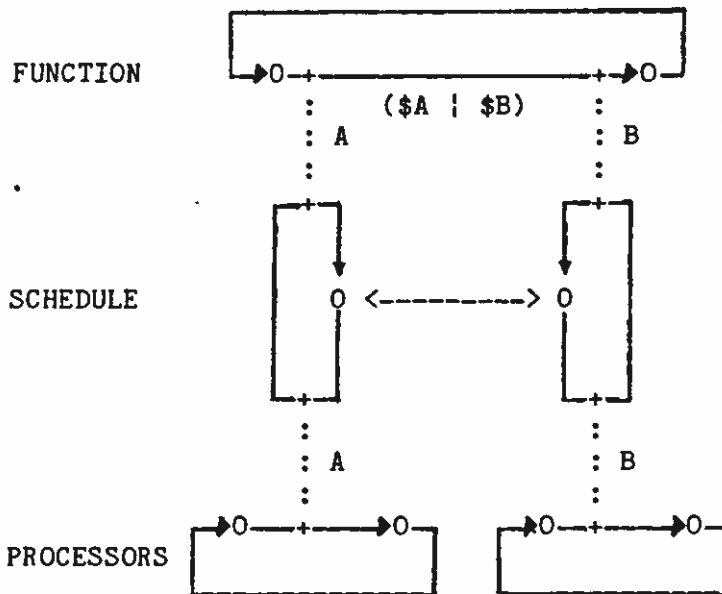
An alternate model that employs two schedules that are being coordinated is the following:

```
             ┌───────────────────────────────────────────┐
FUNCTION     └─►O─+──────────────────────────+─►O─┘
                 :         ($A ¦ $B)          :
                 : A                          : B
                 :                            :
              ┌──┐  ┌───┐          ┌───┐  ┌──
              │  │  │   │     B    │   │  │
              │  ▼  │   ▼  +·············+  │
SCHEDULES     │  O──┘     O  │
              │  +·············+  O  │
              │     A         │  │
              └──+            └──+
                 :                :
                 : A              : B
                 :                :
           ┌─►O─+──►O─┐    ┌►O─+─►O─┐
PROCESSORS │         │    │        │
           └─────────┘    └────────┘
```
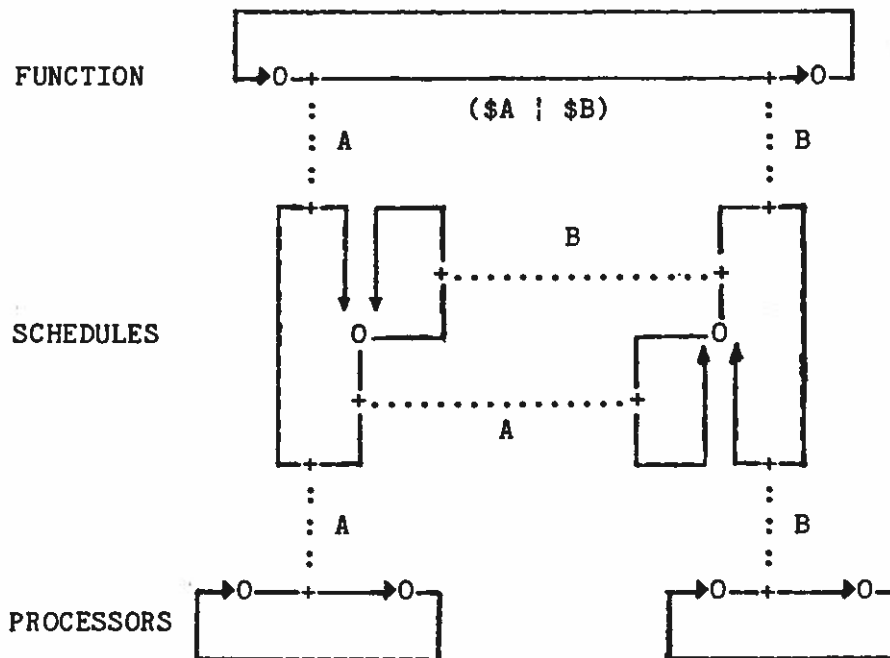
In both cases the function has two separate synchronizations to schedules in the scheduler.

The dynamic nature of scheduling complicates the issue of allocation. In fact, in the most general case, allocation of functions to processors can only be determined at instants of synchronization, and the concept of an allocation as something spanning a period of time becomes useless.

However, the schedules can be written in such a way as to partition the allocation based on the states of the schedule, and this is desirable for system design. For example, the diagram given previously for a function scheduled on one of two processors clearly shows that when the scheduler is in the left subgraph, the function is allocated to the left processor; when the scheduler is in the right subgraph, the function is allocated to the right processor.

Consider a schedule S which will use connection I1 for link L until I1 fails, and will then use I2:

```
S:: [ LviaI1 := true
     *[ LviaI1$e.e1 --> skip ▯ LviaI1$e1' --> LviaI1 := false ]
     LviaI2 := true
     *[ LviaI2$e.e2 --> skip ▯ LviaI2$e2' --> LviaI2 := false ]
   ]
```

e1 synchronizes to successful transmission via I1, while e1' synchronizes to failure of I1; e2 and e2' perform similar synchronizations on the actions of I2; e synchronizes L via S to I1 and I2. L, I1 and I2 become now

```
L:: *[ true$e --> P?y C!y ]

I1:: *[ true$e1 --> [ O?b1 --> D!b1 ] ▯
        true$e1' --> *[ true --> skip ]
     ]
I2:: *[ true$e2 --> [ O?b2 --> D!b2 ] ▯
        true$e2' --> *[ true --> skip ]
     ]
```

The synchronizations in P, C, O, and D are the same as in the previous listing of these processes.

It may now seem that the second portion of I1 and I2 is redundant: since the schedule ensures that a failed link is never usable again, need this permanent failure be included in the architecture description? The answer depends on the particular use of the model. If an architecture is being designed to a particular set of specifications, the statements modelling it may be partitioned arbitrarily. On the other hand, if an architecture has been given to the designer, clearly it is not modifiable. Since this architecture has been considered in some detail in the previous section, it will be taken as given.

It can be shown that with this particular scheduler, the x produced by P will get through to C to be consumed as z so long as at least one interconnection I has not failed. The formal methods have been presented in previous sections, so this proof outline will be informal and emphasize the incremental nature of these proofs.

1. A property of the scheduler itself can be proven. If the first guarded command with LviaI1 true is called send1 and the second such guarded command is called quit1, and similarly the sections with LviaI2 true are called send2 and quit2, it can be shown that the behavior of the
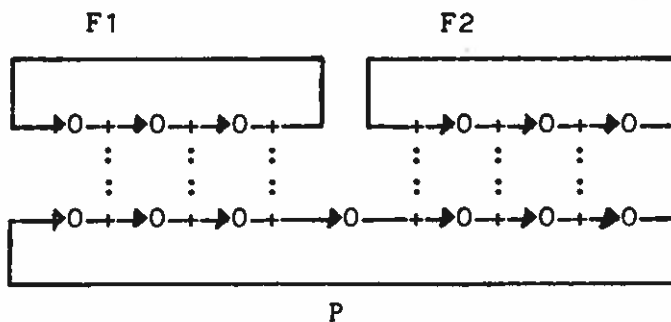
schedule is

(send1)* quit1 (send2)* quit2.

2. Independent of the scheduler and based on the static part of the allocation, proofs of the functionality and architecture from previous sections demonstrate that if a message generated by O reaches D and is used (i.e. provided there is some functioning interconnection) then a corresponding message is produced by P and reaches C to be consumed there. This is based on the synchronization h between O and P and the synchronizations g1 and g2 between D and C.

3. With the scheduler in place it is quite simple to see that, based on synchronizations e1 and e2, for every usage of L there is a corresponding usage of I1 or I2.

4. By combining the proofs of scheduler behavior and interconnection usage (the first and third proofs above), it can be shown that the process L is synchronized to I1 until I1 fails; then it is synchronized to I2 until I2 fails.

5. By combining the proofs of message passing and sequence of interconnection usage (the second and fourth proofs above), it can be shown that so long as I1 or I2 is functioning, messages will get from P to C.

The sharing of one processor by several functions may be illustrated by the figure below in which the functions F1 and F2 take orderly turns to the use of the processor P.

No intervening schedule is required here due to the simple nature of the scheduling policy. In general, however, schedules are used to model more complex policies involving priorities and preemption.

PERFORMANCE AND INSTRUMENTATION

This section discusses and illustrates the dual role played by the performance specification. First, it controls non-determinism in the functionality, architecture, and scheduler. Second, it records the assumptions made by the designer about the characteristics of the existing or envisioned system components. This is accomplished by stating the rules by which various relevant performance attributes are being computed.

Non-determinism may enter the model because of modelling a non-deterministic activity in the system or its environment (e.g., entry of jobs into a system) or a deterministic process whose original deterministic nature has been lost in the process of abstraction into the model. In many cases, something is known about the statistical nature of the non-determinism present. Actors may be defined so as to have a probabilistic behavior which, through instrumentation (i.e., synchronization), may be imposed on functions, processors and schedules.

Let us consider, for instance, that probabilities of failure for the interconnections I1 and I2 are known to be p1 and p2, respectively. Two actors, A1 and A2, may be constructed and synchronized with I1 and I2 (via instrumentation rules a1, a1', a2 and a2') so as to force them to exhibit the desired probabilistic behavior:

```
I1:: *[    true$a1   --> [ O?b1 --> D!b1 ] ▯
           true$a1' --> *[ true --> skip ]         ]
I2:: *[    true$a2   --> [ O?b2 --> D!b2 ] ▯
           true$a2' --> *[ true --> skip ]         ]

A1:: [   z1:boolean
     *[    z1:=random(p1)
           [  ~z1 --> skip$a1 ▯
              z1 --> skip$a1'    ]              ]  ]
A2:: [   z2:boolean
     *[    z2:=random(p2)
           [  ~z2 --> skip$a2 ▯
              z2 --> skip$a2'    ]              ]  ]
```

(Note: "random(x)" is a mathematical function which returns <u>true</u> with the probability x and <u>false</u> with the probability (1-x).)

Actors may be also employed in a information gathering capacity similar to the reporting components of simulation languages. Events in the functionality, architecture, scheduler and even performance specification may trigger predefined actions in the information recording actors. Take, for instance, the computation of the average delay on the link L. Its value depends on the delays on the interconnections I1 and I2 and the way in which L is allocated between the two. Actor A3 could be used to carry out the computation. By properly synchronizing it with the instrumentation rules a1 and a2, A3 is able to compute the average delay time given the assumptions about the delays on I1 and I2:

```
A3:: [    n:integer; delay:integer;
          n:=0 delay:=0
          *[   true$a1 --> delay:=delay+d1; n:=n+1;
                           average(n):=delay/n       ∏
               true$a2 --> delay:=delay+d2; n:=n+1;
                           average(n):=delay/n                ]    ]
```

It should be pointed out that fundamental to obtaining correct results is to assure that a recording actor does not actually interfere with with the behavior of the processes with which it is synchronized. Proving this for A3 is trivial: A3 is always ready to synchronize with a1 and a2 without imposing any restrictions in their ordering. A4 is provided below as a counterexample.

```
A4:: [    n:integer; delay:integer;
          n:=0 delay:=0
          *[   skip$a1;    delay:=delay+d1; n:=n+1;
                           average(n):=delay/n
               skip$a2;    delay:=delay+d2; n:=n+1;
                           average(n):=delay/n                ]    ]
```

This actor forces alternate use of I1 and I2 in addition to computing the average delay—the result is deadlock after the first use of the interconnection I1 by the link L. A sufficient condition for non-interference is to assure that the synchronized statements in a recording actor can be executed in arbitrary order, e.g., A3 permits the behavior {a1, a2}* while A4 is limited to the behavior {a1.a2}*. (Note: the period is used here to denote concatenation of symbols in a sequence.)

Existing discrete event simulation languages such as SIMULA [BIRT73] already provide the means by which one may define random distributions for event arrival and processing time as well as reporting features. The fact that event synchronization can offer the same capability is of some merit. Beyond this, however, event synchronization should be given serious consideration as a potential mechanism for integrating discrete event simulation in design specification languages. The analogy to actual instrumentation of system components has a certain intuitive appeal. The ability to integrate the performance and functional issues while still maintaining a strong separation of concerns is desirable and has been attempted already by others (e.g., SREM [BELL77]).

CONCLUSIONS

The way the virtual system is structured presents several significant advantages:

        <u>Coverage of key technical considerations</u>. Broad and integrated coverage of key technical considerations facing the distributed system designer is provided, including software and hardware organization, static and dynamic allocation, and performance modelling.

        <u>Total system design perspective</u>. Distributed systems are complex hardware/software aggregates whose design requires careful consideration of the relationship between functionality and architecture in order to meet highly demanding constraints. The virtual system has the ability to capture precisely the essence of this relationship.

        <u>Separation of concerns</u>. The components of the virtual system correspond to well established areas of design expertise (e.g., software design, architecture design, resource allocation, performance modelling). The process communities used to model them capture design decisions that may be evaluated (up to a certain point) independent of the overall design or in a limited context. Dependencies between these components are expressed via sets of synchronization rules that tie them together; e.g. functionality, architecture and scheduling policies (all having independent existence) are brought together by the allocation (i.e., ALC).

        <u>Complexity control</u>. The model permits independent elaboration, analysis and modification of its components while allowing easy identification of the entities affected by particular changes, when present. For instance, the architecture can be modified without changing the functionality, to determine how best to implement a required set of functions. Alternatively, the functionality can be restructured without changing the architecture, to determine how best to take advantage of a particular architecture or feature of the system hardware. In either case, the definition of the allocation may be used to determine the consequences of that change for the other components.

        <u>Design flexibility</u>. There are systems whose software or hardware are given and others for which both must be selected by the designer. In the latter case, sometimes the architecture and other times the software becomes the dominant concern with its design turning into a constraint for the other. All these cases and combinations thereof are accommodated by the structure of the virtual system.

Aside from its structure, the primitives employed by the model (process and event synchronization) provide a high degree of analyzability of proposed or modelled systems. It is out intent to use CSPS and the

associated incremental proof strategy as a stepping stone in the development of a distributed systems specification language. While CSPS will provide the means for defining the semantics of the language, the form of the language will be determined primarily by human factors.

REFERENCES

[APT80]   Apt, K. R., Francez, N., and DeRoever, W. P., "A Proof System
           for Communicating Sequential Processes," ACM Trans. Prog. Lang.
           Sys. 2, 3 (July 1980), pp. 359-385.

[BELL77]   Bell, T. E., Bixler, D. C. and Dyer, M. E., "An Extendable
           Approach to Computer-Aided Software Requirements Engineering,"
           IEEE·Trans. on Soft. Eng. SE-3, No. 1, pp. 49-60, January 1977.

[BIRT73]   Birtwistle, G. M. et. al., Simula Begin, New York: Petrocelli,
           1973.

[ESTR78]   Estrin, G., "A Methodology for Design of Digital Systems,"
           1978 NCC Proc., pp. 313-324, 1978.

[HOAR78]   Hoare, C. A. R., "Communicating Sequential Processes," Comm. ACM
           21, 8 (August 1978), pp. 666-677.

[LEVI81]   Levin, G. M. and Gries, D., "A Proof Technique for Communicating
           Sequential Processes," Acta Informatica 15, 3 (July 1981), pp.
           281-302.

[MARI79]   Mariani, P. M. and Palmer, D. F., Distributed System Design,
           IEEE Computer Society Press, 1979.