Report Number: WUCS-86-08

1986-04-01

Toward Comprehensive Specification of Distributed Systems

Gruia-Catalin Roman, Michael E. Ehlers, H. Conrad Cunningham, and R. Howard Lykins

A new approach to modelling distributed systems is presented. It uses sequential processes and event synchronization as building blocks to construct a cohesive picture of the interdependent requirements for the functionality, architecture, scheduling policies, and performance attributes of a distributed system. 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 and leads to the emergence of a uniform incremental strategy for verifying both logical and performance properties of distributed systems. Several small... **Read complete abstract on page 2.**

Toward Comprehensive Specification of Distributed Systems

Gruia-Catalin Roman, Michael E. Ehlers, H. Conrad Cunningham, and R. Howard Lykins

Complete Abstract:

A new approach to modelling distributed systems is presented. It uses sequential processes and event synchronization as building blocks to construct a cohesive picture of the interdependent requirements for the functionality, architecture, scheduling policies, and performance attributes of a distributed system. 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 and leads to the emergence of a uniform incremental strategy for verifying both logical and performance properties of distributed systems. Several small distributed systems have been modelled using this approach. These exercises enabled us to evaluate the notation system and to gain some expertise on how to approach the specification of distributed systems. This paper describes one of the models and the modelling strategy that has emerged from these exercises.

TOWARD COMPREHENSIVE SPECIFICATION OF DISTRIBUTED SYSTEMS

Gruia-Catalin Roman, Michael E. Ehlers,
H. Conrad Cunningham and R. Howard Lykins

WUCS-86-08

April 1986

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

A shorter version of this report appeared in the Proceedings of the 7th International Conference on Distributed Computing Systems, pp. 282-289.

TOWARD COMPREHENSIVE SPECIFICATION OF DISTRIBUTED SYSTEMS

Gruia-Catalin Roman, Michael E. Ehlers,
H. Conrad Cunningham, and R. Howard Lykins

Department of Computer Science
WASHINGTON UNIVERSITY
Saint Louis, Missouri 63130, U.S.A.

ABSTRACT

A new approach to modelling distributed systems is presented. It uses sequential processes and event synchronization as building blocks to construct a cohesive picture of the interdependent requirements for the functionality, architecture, scheduling policies, and performance attributes of a distributed system. 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 and leads to the emergence of a uniform incremental strategy for verifying both logical and performance properties of distributed systems. Several small distributed systems have been modelled using this approach. These exercises enabled us to evaluate the notation system and to gain some expertise on how to approach the specification of distributed systems. This paper describes one of the models and the modelling strategy that has emerged from these exercises.

1. INTRODUCTION

Our research is aimed at developing system level models that enable the designer to formulate and answer questions regarding the system's logical correctness and performance characteristics when the interaction between the hardware and the software is important, i.e., when the impact of faults, failures, communication delay, hardware selection, scheduling policies, etc., must be considered. In the simplest terms, our concern extends beyond the traditional software correctness questions by addressing the issue of *employing logical verification techniques to determine software correctness and performance characteristics when running on a particular distributed hardware architecture and using a particular operating system.*

The models are called *Virtual Systems* and represent either abstractions of existing systems or definitions of proposed systems — a system is a collection of application software modules, allocated over hardware components. The virtual system [1] consists of six components each abstracting some aspect of a distributed system. They are called functionality, architecture, scheduler, allocation, performance specification, and instrumentation.

We are currently experimenting with specifying virtual systems using a language called CSPS *(Communicating Sequential Processes with Synchronization).* An extension of Hoare's CSP [2], CSPS allows synchronization between multiple processes in addition to the I/O primitives of CSP. The functionality, architecture, scheduler, and performance specification are defined as closed communities of *communicating concurrent processes* ; the allocation and instrumentation are defined as sets of *event synchronization rules.* Employing CSP as a base allows modelled systems to be verified using techniques already developed for verifying CSP programs and leads to the emergence of a uniform incremental strategy for verifying both logical and performance properties of distributed systems.

The remainder of this paper starts with an overview of the CSPS extensions. Next, the structure of the virtual system and the motivation for that structure are explained. A simple example illustrates the different components of a virtual system. Finally, the specification strategy is described by means of a sample system and its model.

2. CSPS CONCEPTS AND NOTATION

A full description of CSP is available in Hoare's original paper [2]. CSPS *(Communicating Sequential Processes with Synchronization)*, as used in this paper, is a direct extension of CSP. The new feature, ***n-way synchronization***, is indicated through the use of ***synchronization commands***. The simplest form of a synchronization command consists of a ***synchronization label*** followed by a ***synchronization operator:*** **"a $;"**.

A synchronization command may occur many times in the text of a single process. Any process whose text includes a particular synchronization command must always participate in the corresponding synchronization, i.e., a synchronization takes place only when each participating process is ready to execute the same synchronization command. If two or more synchronization and I/O commands appear together separated by blanks (forming a ***composite synchronization command***), all the commands must occur together — this feature enhances modularity of the models but adds no extra power to CSPS. CSPS extends the CSP distributed termination convention to cover event synchronization commands. The last element in a guard may be an I/O command, a synchronization command, or a composite synchronization command. A guard fails if the boolean part is *false* or if a terminated process is involved in any of the synchronizations or I/O commands appearing on the guard; a guard is passable if the boolean part is *true*, the I/O command may be executed, and all synchronizations may take place.

N-way synchronization, as introduced so far, is unable to simulate the data transfers accomplished by I/O commands. The ***n-way synchronization with pattern match*** is a mechanism for accomplishing what may be seen as data transfer.

Here is an example using synchronization with pattern match to simulate an I/O exchange:

```
P::    ... PtoQ $ (x); ...
Q::    ... PtoQ $ (y'); ...
```

PtoQ is the synchronization label which is used to determine (a priori) the set of processes that must synchronize. The pattern definition appears as a list following the synchronization operator. For a synchronization to be successful, each pattern must contain the same values. A single quote indicates that the value of the particular variable is part of the pattern for the respective synchronization but it is ***indeterminate***, i.e., the variable will assume any value (within the restrictions of the variable type) that renders the pattern match successful.

3. VIRTUAL SYSTEM

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 synchronizations. Communication within each community takes place via CSP I/O commands. All interactions between the application software and the operating system and between the operating system and the hardware are described using synchronization commands. These commands are ignored when the component is discussed in isolation from the rest of the system. In this manner, models are easier to modify and analyze. The components of a virtual system are described below, with the examples drawn from the simple system in Figure 1.

FUNCTIONALITY (a producer/consumer problem)

P:: [x:=0 *[x<3 → x:=x+1; **po $ (x)**; L!x]]

L:: *[**li1 $** P?y → C!y; **di1 $** # **li2 $** P?y → C!y; **di2 $**]

C:: [u:=0 *[L?z → **cd $ (z)**; u:=u+z]]

ARCHITECTURE (unreliable communication)

O:: [up:=true *[up; **po $ (n')** → up:=false *[not(up); I1!n → up:=true
not(up); I2!n → up:=true]]]

I1:: [up1:=true *[up1; **li1 $** O?k1 → D!k1; **di1 $** # up1 → up1:=false]]

I2:: [up2:=true *[up2; **li2 $** O?k2 → D!k2; **di2 $** # up2 → up2:=false]]

D:: *[I1?r → **cd $ (r)** # I2?r → **cd $ (r)**]

SCHEDULER (sequencing the use of I1 and I2)

S:: [*[**li1 $** → **di1 $**] *[**li2 $** → **di2 $**]]

PERFORMANCE SPECIFICATION (flows via I1 and I2)

A:: [f1:=0; f2:=0 *[**li1 $** → f1:=f1+1 # **li2 $** → f2:=f2+1]]

Figure 1: A Virtual System.

3.1. Functionality

The *functionality* represents the requirements for the application software. As such, it defines the interactions between the system and its operating environment as observed at the system user level. Any proofs that consider the functionality alone (e.g., logical correctness, deadlock freedom, etc.) assume no interference from the operating system and the hardware and the availability of adequate resources (e.g., storage space). Nevertheless, they may be used later in proofs about the system as a whole.

Example: The functionality defined in Figure 1 involves three modules: a producer P which generates several values and passes them, one at a time, to a link L which, in turn, forwards each value to a consumer C. The consumer C receives a value from link L and computes the sum of all the values received so far.

3.2. Architecture

The *architecture* models the physical structure of the system and the behavior of the individual components that make it up. The components are not seen as physical devices but as resources required to support the application software.

Example: In Figure 1, we consider two processors that communicate via failing lines between them. An originator O selects a value to be sent to destination D, and sends it on one of the unreliable interconnections I1 and I2. The unreliable nature of the interconnections has been expressed as a nondeterministic choice in a guarded iteration. Selection of the first guard results in transmission of a value from O to D. Selection of the second guard results in the termination of the process modelling the respective interconnection.

3.3. Scheduler and Allocation

The *scheduler* and the *allocation* describe the interdependency between functionality and architecture. The scheduler consists of a community of processes called schedules. Each schedule, together with the event synchronization rules that are part of the allocation, establishes a mapping between relevant events in one of the modules in the functionality and sequences of events involving one or more processors in the architecture. The allocation may be static or dynamic.

Example: Our illustration (Figure 1) involves the static mapping of P and C to O and D, respectively, and the dynamic allocation of L to the interconnections I1 and I2. L is mapped to I1 until the failure of I1 causes L to be mapped to I2. The synchronization with pattern match is employed with the label **po** in order to make O transmit to D the same value that P sends to C. This is accomplished by having O use an indeterminate variable assignment (**po $ (n');**) which induces n in O to assume the same value as x in P (**po $ (x);**), i.e., the only value for which a match becomes possible. In the case of the label **cd** the pattern serves only as a check that the value z received by C is the same as the value r received by D.

3.4. Performance Specification and Instrumentation

The *performance specification* consists of a community of processes called actors whose interactions with the other components of a virtual system are defined by the synchronization

commands which are part of the *instrumentation*. Actors may be used in a measurement capacity similar to the reporting components of simulation languages or may be used to control nondeterminism in the functionality, architecture, and scheduler by inducing patterns of behavior having particular characteristics.

Example: In Figure 1, the computation of the flows through I1 and I2 is done by the actor A which takes advantage of the synchronization labels li1 and li2 to determine which interconnection is used and to store this information into internal counters f1 and f2.

4. THE SPECIFICATION STRATEGY

This section describes the system specification strategy that emerged from the use of the virtual system and CSPS in the modelling of several simple systems. After first describing the problem to be solved, an overview of the model used in the solution is given. Following this, the functionality, architecture, and scheduler portions of a sample system are discussed in detail. Finally, an alternate architecture for the model is discussed, including its ramifications on the scheduler and functionality.

4.1. Sample Problem

For the purpose of illustrating the specification strategy, we selected a problem from the arena of geographic data processing, that of finding the shortest path for a road between two specified endpoints, given the elevation for the points in a rectangular area containing the endpoints and that the road cannot go between points that differ in elevation by more than some delta. This problem was chosen because there are multiple algorithmic solutions and multiple architectures for these solutions. In the algorithm we selected, the distances to all points in the area from each endpoint are calculated, and then those points such that the sum of their minimum distances to each endpoint equals the minimum distance between the endpoints are selected for the road to go through. The endpoints and area boundary are inputs to the system (from the environment) and the potential road points are outputs from the system.

4.2. Model Overview

In the model developed, the issues of primary concern are storage requirements for data (code space was ignored) and communication among the components of the system. Storage allocation is used to demonstrate how a standard resource of a computer system could be modelled and to determine what sort of information could be obtained on the usage of the resource from the model. Since the systems of interest are distributed, communication between the various components is a necessity, requiring the inclusion of the communication capabilities in the model. Other issues of potential interest, but excluded from the models for simplicity, are time (such as idle time, computation time, and communication time), storage for code, computational processing, etc. Our primary purposes are to develop modelling experience and to evaluate the modelling paradigm. Hence, the number of issues dealt with is kept small enough to enable thorough coverage, but large enough to provide worthwhile modelling experience.

The form our solution took was four different virtual systems, all sharing the same functionality, but with different architectures, schedules and allocation. The first model had a

single processor for the architecture, with all the modules in the functionality allocated to that processor. This version of the solution allowed us to work with the storage allocation and communication between software processes, without the complexity of handling communication at the architecture level. In the second version of the model, which is presented in the following section, we added a second identical processor to the architecture, statically allocating modules to each processor (see Figure 2).

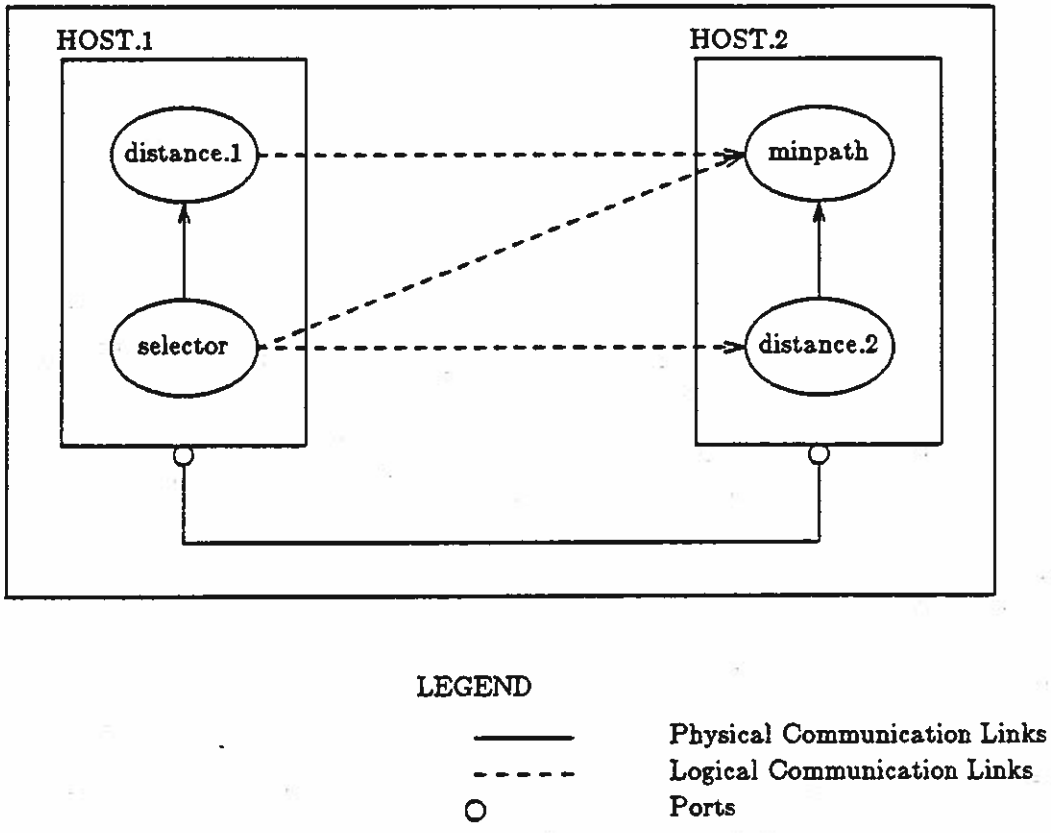

Figure 2: Static allocation, dual-processors, point-to-point communication.

The storage allocation remained virtually unchanged, but now the model also had to handle communication between modules on different processors, bringing processor-to-processor communication into the model. The communication used between processors was a point-to-point dedicated communication link. In the next version of the model, we replaced the simple communication link in the architecture by a bus, enabling investigation of a more complicated communication mechanism. The changes in the model that resulted are also included in the next section. In the fourth version of the model, we went back to the point-to-point communication, but also included the dynamic relocation of modules from one processor to another. Although this model is not included, we do discuss the results obtained.

4.3. Functionality

The functionality portion of the model is an operational specification of the processing to be carried out by the system. Structurally, the functionality consists of one or more communicating processes (referred to as modules) that collectively describe the software in the system. Separation of the software into modules is based primarily on logical divisions of work in the computation process desired. In general, the structure of the functionality does not reflect the underlying architecture, and hence could be used in conjunction with several different architectures with little or no modification. This allows for the effects of different architectures on the performance of a functional organization to be studied. The only restrictions the functionality places on the underlying architecture is that the operations needed by each module are provided, and that the communications between the modules can be accomplished by the architecture.

The functionality for this problem is divided into four modules: selector, distance.1, distance.2, and minpath (as seen in Figure 2). Selector obtains the endpoints and the area boundary from the environment (an unspecified module), checks the area for inclusion of the endpoints, and then obtains the elevation data for the area from the geography database (another unspecified module). Finally, it sends the endpoints and the elevation data to distance.1, the endpoints in the opposite order and the elevation data to distance.2, and the area boundary to minpath. Distance.1 and Distance.2 are identical modules which calculate the distance to every point in the area from the first of the pair of endpoints it receives, and then sends the distances to minpath. Minpath uses the area boundary and the two distance arrays to determine the set of points that are on a minimum path, then sends these to the environment. The communication paths required by the modules in the functionality are indicated by the dotted lines in Figure 3. In the actual specification, these communications are represented by CSPS input and output commands. In addition to communication, modules can declare variables and carry out computations. In this particular model, computation was not a concern, so whenever possible the computational process is abstracted into a mathematical function, leaving module specifications that consist primarily of variable declarations, flow of control constructs, and communication commands. As an example, the specification of the selector module is given in Figure 3. The computations required, namely those to check the area specification and endpoints, are abstracted into the functions End_points_ok and Area_spec_ok, leaving only the specification of the operations of interest in this model.

```
MODULE selector ::
[
        a1 : point;
        a2 : point;
        r1 : point;
        r2 : point;

        env1?(a1,a2,r1,r2);

        [       Area_spec_ok(a1,a2)             → skip
        #       not(Area_spec_ok(a1,a2))        → abort
        ];

        [       End_points_ok(a1,a2,r1,r2)      → skip
        #       not(End_points_ok(a1,a2,r1,r2)) → abort
        ];

        distance.1!(a1,a2,r1,r2);
        distance.2!(a1,a2,r1,r2);
        minpath!(a1,a2);
]
```

Figure 3: CSPS Specification of selector module

Although this set of processes specify the functionality, it does not include any ties to the remaining components of the virtual system that indicate how the commands in the functionality correspond to commands in the other levels, i.e. operating system and hardware. In general, each command at the functionality level is viewed as an atomic action, although it may require several commands at other levels to accomplish it. Therefore, the functionality must execute in step with the other levels; a module cannot continue to the next command until all operations at the other levels corresponding to the current command have completed. The mechanism used to do this synchronizing of presumed atomic actions in the functionality with sequences of commands at the lower levels (and hence tie functionality to the architecture) is the synchronization commands. These come in two forms: the labeled synchronization discussed earlier and the second is the resynchronization command, **$$**. The resynchronization command specifies that a synchronization is to be performed with all processes that have synchronized using the **$** operator since their last **$$** command. Their normal use in the functionality is to precede each command with a normal synchronization and to follow it with a resynchronization, effectively bracketing the command and the sequence of steps that correspond to it at the other levels. As an example, consider the assignment command in Figure 4. It is preceded by the labelled synchronization **assign $** and followed by the resynchronization command.

```
assign $;        x := exp;        $$;
```

Figure 4: Simple command with bracketing synchronizations.

In this form, the synchronization, command, and resynchronization can be viewed collectively as an atomic action in the functionality. Since the synchronizations are almost always used in this manner in the functionality, a special (but equivalent) notion is used in which the the $ operator of the synchronization is replaced by $$, and the explicit resynchronization is omitted to combine the synchronization, command, and resynchronization into a single statement. As an example, Figure 5 shows the same statement as in Figure 4 with the alternate, but preferred, representation.

```
assign $$ x := exp;
```

Figure 5: Alternate representation for command bracketing.

In the virtual system we are gradually building in this section, only those commands involving the allocation of memory, deallocation of memory, or communications are of interest, so only these are bracketed in the specification by synchronizations. Since CSPS is a block structured language with explicit variable declarations, storage for a variable must be allocated when it is declared and deallocated when the block containing its declaration is exited. The inclusion of the synchronizations for the allocation is accomplished by bracketing the individual declaration statements, and for the deallocation by placing a bracketed skip (or null) statement at the end of the block corresponding to each declaration statement in the block. Since different size variables are required, the size of the storage allocation (deallocation) is specified through the pattern portion of the labelled synchronization. A determinate pattern is used since the size is known here and is information being given to the lower levels of the virtual system. The synchronization labels used are **declare** and **delete** for the allocation and deallocation, respectively. Figure 6 contains a module p with integer i declared in the outermost block and array A declared in an inner block, with synchronizations included. The preferred notation given before is used here and throughout the remainder of the model.

```
MODULE p ::
[
declare.p $$ (1)        i : INTEGER;
        [
        declare.p $$ (10)       A : ARRAY(10) OF INTEGER;

        delete.p $$ (10);       /* deallocate array A */
        ]
delete.p $$ (1);/* deallocate integer i */
]
```

Figure 6: Example of declare and delete operations.

Included as part of the synchronization label for **declare** (**delete**) is the name of the module in which it occurs. This is necessary so that the synchronization label for **declare** (**delete**) is unique in each module, otherwise all modules would have to synchronize whenever one of them wanted to do an allocation or deallocation of storage. This inclusion of process name in synchronization label to ensure uniqueness occurs throughout the remainder of the model.

The communication between the modules in the functionality is another important concern. Since this model is only concerned with storage requirements, and not the values of the data, only the size of the block of data to be transferred is of interest when a send or receive is done. Hence, the synchronization prior to the send needs only the size in its pattern field to provide all necessary information to the other levels. Should additional information be required (such as values for the data) in a different model, the only change to the functionality would be in the synchronization commands, or more exactly, the synchronization patterns. As an example, Figure 7 shows the sending of array A from module p to q followed by the receiving from q of a value for integer i.

```
send.p.q $$ (10)        q!A;
receive.p.q $$ (1)      q?i;
```

Figure 7 - Sample send and receive commands in module p.

Note in this case, both processes involved are included in the label to differentiate a p to q communication from a q to p one. The communication commands in the functionality are independent of the mechanism used by a particular architecture to accomplish them, even to the extent that the modules may reside on the same or different processors in the architecture. As an example of the complete specification of a module for the storage allocation model, selector's definition is given in Figure 8 with all synchronizations. Another CSPS notation used in the model is the '~' as the initial value of the constant f0. The '~' represents the name of the process (in any community) in which it is referenced, so in this case it is equivalent to "selector".

Since it is used to define the constant f0, the use of f0 anywhere in the module is identical to using "selector" at the same place. This notation is most useful when several processes of similar structure are defined using one specification, as could be the case for distance.1 and distance.2, which are identical processes. Also, the built-in function SIZE is used to obtain the storage requirements of variables where that information is necessary, to keep the model as general as possible.

```
MODULE selector ::
[
f0 : MODULE CONSTANT := ~;

declare.f0 $$ (SIZE(a1))      a1 : point;
declare.f0 $$ (SIZE(a2))      a2 : point;
declare.f0 $$ (SIZE(r1))      r1 : point;
declare.f0 $$ (SIZE(r2))      r2 : point;

get.f0.env1 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) env1?(a1,a2,r1,r2);

[      Area_spec_ok(a1,a2)            → skip
#      not(Area_spec_ok(a1,a2))       → abort
];

[      End_points_ok(a1,a2,r1,r2)     → skip
#      not(End_points_ok(a1,a2,r1,r2)) → abort
];

send.f0.distance.1 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) distance.1!(a1,a2,r1,r2);
send.f0.distance.2 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) distance.2!(a1,a2,r1,r2);
send.f0.minpath $$ (SIZE(a1)+SIZE(a2)) minpath!(a1,a2);

delete.f0 $$ (SIZE(a1));
delete.f0 $$ (SIZE(a2));
delete.f0 $$ (SIZE(r1));
delete.f0 $$ (SIZE(r2))
]
```

Figure 8: Selector module with storage allocation model synchronizations.

4.4. Architecture

The architecture portion of the model specifies the processing resources available in the system and the communication mechanism connecting the various pieces together. Each processor is capable of performing various operations, which are used to implement software actions. The specification of the processors is an abstract view of their capabilities, rather than an exact description of low level details of processor design. Capabilities that might be attributed to a processor are allocation/deallocation of storage, performance of a variety of

computations (such as matrix inversion, expression evaluations, etc.), exchange of data with other processors, etc. Since the architecture also involves the physical communication links between processing elements, the communication system itself may be modelled by one or more processes that provide operations to the processing elements (especially if there is any complexity to them at all). The availability of these operations and their cost is the primary distinction between various processors in the architecture.

In the virtual system for the example problem, a dual processor system is used for the architecture, with the processors being identical, general-purpose processors. This selection requires communication between processors to be modelled, as well as the storage operations to be dealt with, without additional and undesirable complexity. The processors used can perform three different operations, namely, allocate storage, deallocate storage, and transfer data.

First, consider the storage allocation model, ignoring the communications for the present. Memory in the processors is modelled by a counter (one in each processor) of available space, which is decremented (incremented) appropriately for an allocation (deallocation) operation. Since each processor must make both the allocate and deallocate operations available on any "cycle", the specification takes on the form of a guarded repetition, where each pass through the repetition represents a cycle, and each guarded command represents one of the possible operations. An as example, Figure 9 contains a specification of a processor h which nondeterministically does an allocate (first guard) or a deallocate (second guard) on every cycle, using an arbitrary (indeterminate) size, represented by the indeterminate form of n (quoted) as the first statement in the guard.

```
PROCESSOR h ::                  /* k is the memory size for the processor */
[
        M : integer := 0;       /* amount of memory allocated */
        n : integer;            /* size of request */

        *[      true → n'       /* allocate operation */
                        [       M+n≤k → M:=M+n   /* successful allocation */
                        #       M+n>k → skip     /* unsuccessful allocation */
                        ]
         #      true → n'       /* deallocate operation */
                        [       M-n≥0 → M:=M+n   /* successful deallocation */
                        #       M+n<0 → skip     /* unsuccessful deallocation */
                        ]
        ]
]
```

Figure 9: Processor capable of allocate and deallocate.

In this form, the choice of what to do next is arbitrary for the individual processors, as is also the value of n, the amount of storage to allocate or deallocate. To do something useful, the architecture must be tied to the levels above in the virtual system, and this is accomplished by synchronizations, in a manner analogous to their use in tying the functionality to the rest of the virtual system. First, a synchronization command is introduced into each guard so that the software can control which operation(s) can be performed next and indicate through the pattern

(which assumes the indeterminate form) the amount of storage to be allocated or deallocated. The processor still makes an arbitrary choice between the operations, but now it is constrained by the operations which the software is willing to have it perform. Secondly, since the success (failure) of the operations is only known in the processor, this information must also be passed back to the software, and for this two different synchronization commands, **failure** and **success**, are used. Following the completion of an operation, the processor must resynchronize with the software (using the resynchronization operator), so that everything may continue. Finally, to distinguish the completion of a processor action from the completion of an application software action, @ (@@) is used in place of $ ($$) as the synchronization operator. This is necessary because, for a single application software action, several hardware operations may be required. Adding these ties to the example above, the processor with allocation and deallocation appears as shown in Figure 10.

```
PROCESSOR h ::
[
        M : integer := 0;          /* amount of memory allocated, maximum of k */
        n : integer;               /* size of request */

        *[      allocate @ (n') →
                        [       M+n≤k  → M:=M+n;  success @
                        #       M+n>k  → skip;    failure @
                        ]; @@
        #       free @ (n') →
                        [       M-n≥0  → M:=M-n;  success @
                        #       M-n<0  → skip;    failure @
                        ]; @@
        ]
]
```

Figure 10: Processor with basic allocate and free operations, and synchronizations.

The communication operations are modelled as input and output CSPS commands. The size of data is used in place of actual data, since only quantities, not values, are of interest here. Two data transfers are required to implement a module level data transfer. The data size is passed first, followed by the data values, which in this model is simulated by a retransmission of the data size. This protocol is enforced by the schedules that support the inter-module communication; it allows the receiver schedule the opportunity to do any setup, such as buffer allocation in this particular scenario, that is required for the actual transfer. This is discussed more in the schedule section, where the relationship between the architecture operations and the functionality actions are developed. In keeping with the method described for the storage operations, the two instances of the send operation will be represented by two guarded commands, the guards of which will be a synchronization with an indeterminate pattern to obtain the size of data to transfer. Since failure of a communication is not in the model, the output statement for the send operation must either succeed or block (because the other processor doesn't have a matching input statement), hence terminating the guarded command by a resynchronization is sufficient. The labels associated with the initialization operation and actual transfer are **s_ready** and **s_pass**, respectively. As an example, Figure 11 contains the initialize and transfer operations (**s_ready1** and **s_pass1** synchronizations) for processor h1

which supports only one-way communication with h2.

```
PROCESSOR h1 ::
[
        n : integer;        /* size of data transfer */

        *[      s_ready1 @ (n') h2!n  →     skip; @@
         #      s_pass1 @ (n')  h2!n  →     skip; @@
         ]
]
```

Figure 11: Processor with single send capability.

The receive operation is quite similar, with the output command to the other processor replaced by an input command from it. Similar synchronization names (**r_ready** and **r_pass**) are also used in the same fashion to indicate initialization or transfer. This is shown in Figure 12, which contains the specification of a processor h2 that is only capable of receiving data from processor h1 described above.

```
PROCESSOR h2 ::
[
        n : integer;        /* size of data transfer */

        *[      r_ready2 @ (n') h1?n  → skip; @@
         #      r_pass2 @ (n')  h1?n  → skip; @@
         ]
]
```

Figure 12: Processor with single receive capability.

As was the case in creating the synchronization labels for the functionality, uniqueness of the labels will be a problem, but even more serious. Since each operation on each processor is unique, the label for each operation on each processor must also be unique, so the processor name must be included as part of the label. In addition, any schedule process can potentially request any operation from any processor (since the binding of schedules and modules to processors is generally unknown), and these requests must likewise be unique, hence the schedule name needs to be in the synchronization label. Further, on the **ready** and **pass** synchronizations, the schedule and processor with which the communication is taking place also need to be included in the label, along with the direction of transfer to completely specify all the information that is needed to do the exact communication that is desired. Even for a fairly small system, this can lead to a very large specification, most of which is repetitive. This is exemplified by the processor in Figure 13, which provides only the allocate operation, but it must provide it to four schedules, s1, s2, s3, and s4.

```
PROCESSOR h1 ::
[
        M : INTEGER := 0;  /* amount of memory allocated, maximum of k */
        n : INTEGER;       /* size of request */

        *[      allocate.s1.h1 @ (n') →
                        [       M+n≤k→ M:=M+n;    success.s1.h1 @
                        #       M+n>k→ skip;      failure.s1.h1 @
                        ]; @@
         #      allocate.s2.h1 @ (n') →
                        [       M+n≤k→ M:=M+n;    success.s2.h1 @
                        #       M+n>k→ skip;      failure.s2.h1 @
                        ]; @@
         #      allocate.s3.h1 @ (n') →
                        [       M+n≤k→ M:=M+n;    success.s3.h1 @
                        #       M+n>k→ skip;      failure.s3.h1 @
                        ]; @@
         #      allocate.s4.h1 @ (n') →
                        [       M+n≤k→ M:=M+n;    success.s4.h1 @
                        #       M+n>k→ skip;      failure.s4.h1 @
                        ]; @@
        ]
]
```

Figure 13: Processor supporting storage allocation for multiple schedules.

Even this small example is lengthy, and if this processor also provided the free and communication operations for all possibilities, which it must do in general, the specification quickly grows to several pages, despite the small number of actual operations. In a system with many schedules and processors, the problem becomes very burdensome, resulting in the specification losing its readability and clarity. In response, a new notation is used that condenses all operations of a given type (such as allocate) into a single guarded command. The notation consists of a new unary, postfix operator, represented by the double quote, and the introduction of variables of enumeration type into synchronization labels. The operator, when applied to a variable of some enumeration type, results in the entire statement being replicated into n statements, each one with a different value in the enumeration substituted for the variable throughout the statement. Its most common and powerful use is when the synchronization appears in the guard of some guarded command, in which the single guarded command is expanded into several guarded commands, one for each element in the enumeration. This is only a syntactic transformation, and the fully expanded specification can be regenerated from the condensed form at any time. As an example, the condensed version of the processor in Figure 13 is given in Figure 14, with the assumption that schedule is enumerated over (s1,s2,s3,s4).

```
PROCESSOR h1 ::
[
        M : INTEGER := 0;     /* amount of memory allocated, maximum of k */
        n : INTEGER;  /* size of request */
        s: SCHEDULE; /* enumeration variable for all possible schedules */

        *[      allocate.s".h1 @ (n') →
                        [       M+n≤k→ M:=M+n;    success.s.h1 @
                        #       M+n>k→ skip;      failure.s.h1 @
                        ]; @@
         ]
]
```

Figure 14: Concise description for processor described in Figure 13.

This notation is used throughout the model to shorten as well as clarify the specifications.

Finally, to specify a complete processor for this example, it is necessary to compose the individual pieces that have been described. There are six types of operations, allocate, deallocate, initialize and transfer for send, and initialize and transfer for receive. This is reflected in the complete specification of a processor (see Host.1 specification in Figure 15) by the six guarded commands of the repetition. Each guarded command uses the concise notation described above to include all possible operations.

Before concluding this section, there are several issues that should be touched on. One is that data passage via the synchronizations is predominantly from modules to processors in the examples (e.g. size). This is an artifact of the problem under consideration, not of the modelling strategy, and we do envision systems where significantly more data originates within the processors (e.g., a real time sensor). Another point is that processors exhibit little or no independent behavior, but rather are directed by the schedules in the sample system. This is to be expected because the software execution determines the hardware behavior. More detailed processor models, however, will reveal automatic, independent activities triggered by the software, e.g., access protection and page management.

```
PROCESSOR host.1 ::
[
        M : size := 0;    n : size;
        p,q : MODULE;
        h : PROCESSOR;        h0 : PROCESSOR CONSTANT := " ;

        *[
                allocate.h0.p" @ (n') →
                        [       M+n≤k → M:=M+n;    success.h0.p @
                        #       M+n>k → skip;      failure.h0.p @
                        ]; @@

        #       free.h0.p" @ (n') →
                        [       M-n≥0 → M:=M-n;    success.h0.p @
                        #       M-n<0 → skip;      failure.h0.p @
                        ]; @@

        #       s_ready.h0.h".p".q" @ (n') h!(n)        → skip; @@

        #       r_ready.h".h0.p".q" @ (n') h?(n)        → skip; @@

        #       s_pass.h0.h".p".q" @ (n') h!(n)         → skip; @@

        #       r_pass.h".h0.p".q" @ (n') h?(n)         → skip; @@
        ]
]
```

Figure 15. - Full specification of processor host.1

4.5. Scheduling and Allocation

The scheduler portion of the model has the responsibility of tying the modules of the functionality to the processors in the architecture. In other words, it fills the role normally associated with a conventional operating system, providing the application software access to the operations available on the physical hardware through a variety of system services. The scheduler is a collection of one or more processes, called schedules, very much analogous to the communities of processes that make up the functionality and architecture. In general, the scheduler consists of two groups of schedules, the first, a collection of "shadow" schedules, called servers, one for each module in the functionality, and the second, a collection of special purpose schedules for various system functions such as bus daemons, queue managers, special device controllers, etc. Having a schedule for each module allows the modules to view the underlying architecture as a dedicated system, much like in a conventional time-sharing system. In addition, it provides flexibility and simplicity in the allocation process by having modules implicitly allocated to the same processor that their server resides on, hence by moving the servers, the application modules are reallocated. Often, properties of the allocation scheme, whether it be a static allocation or dynamic allocation of modules to processors, can be

analyzed entirely inside the scheduler, and hence completely independent of the functionality. By creating separate schedules for each special service in the scheduler, such as daemons, etc., the particulars of certain services are encapsulated thus making composition and analysis of the system simpler.

The purpose of each server is to provide the capabilities required by its module through use of the operations available on the processor to which it is allocated. This implies that the processor provides sufficient operations to the modules allocated to it, which is one property of a proper allocation, namely that modules are only allocated to processors capable of supporting them. The schedules also maintain information about the state of the system, such as the amount of storage allocated to each module on a given processor. Such information may be kept in the individual servers or in a central schedule for the processor, depending on whether it is strictly local information or global across the schedules on a processor. The amount and type of information maintained depends on what properties are being modelled and the level of detail of the model.

To perform its task, a server schedule must be able to direct the processor through the sequence of operations necessary to accomplish each of the commands used by its module. In doing this, it interfaces with the module through an identical synchronization/resynchronization pair that brackets each command in the module, but instead of a single command between them, the server contains the sequence of operations that the processor must perform to accomplish the command for the module. Since the operations of the processor are controlled through another set synchronizations (@), most of the statements in this sequence are such synchronizations with the processor. Finally, since the schedule does not know what order the commands will be executed by its module, it must be capable of servicing commands in any order, resulting in a structure much like that of the processor. Hence, the schedule is a repetition, where each guard is a synchronization corresponding to one of the module's commands, and the statement portion of the guarded command is the sequence of operations needed to implement the command, followed by a resynchronization with the module, if the sequence is successful. Since the operation in the sequence can fail, these failures must be handled in the scheduler. In the current model, failures are handled by aborting the schedule process, thereby blocking the module from further execution.

As an example, consider a schedule s, allocated to processor h0, that supports the variable declaration in its module f. The operations necessary in s are to allocate the memory on the processor and, if that succeeds, terminate normally, otherwise terminate in failure by aborting the server schedule, leaving the module waiting forever on the resynchronization. Such a variable declaration in the functionality, the corresponding sequence in the schedule, and the allocate operation in the architecture are shown in Figure 16.

Functionality:

```
declare.f $$ (SIZE(A))A : ARRAY(10) of INTEGER;
```

Schedule:

```
declare.f $ (n') →
        allocate.h0.f @ (n);
        [       success.h0.f @ → skip; @@; $$
        #       failure.h0.f @ → abort; @@
        ]
```

Processor:

```
allocate.h0.f @ (n) →
        [       M+n≤k → M:=M+n;   success.h0.f @
        #       M+n>k →           failure.h0.f @
        ]; @@
```

Figure 16: Processing a variable declaration.

In an actual server schedule, the above guarded command would be one of many in the repetition. For each of the other commands, an analogous guarded command is created, and in the case of the delete command, it is the same as the declare, except the free synchronization is used in place of the allocate synchronization. The communication operations are more complicated. They are broken down into two cases (for either a send or a get), one for when the other module involved is allocated to the same processor (local transfer), and one for when it is allocated to a different processor (distant transfer).

In the case of the local transfer described in Figure 17, the basic scheme is to use a temporary, common buffer to pass the data. It is the responsibility of the sender to allocate the buffer, place the data in it, and then signal the receiver. Once the receiver acknowledges the signal, the sender is finished, and the destination must deallocate the buffer after it has retrieved the data. In this model, the data values are not modelled, only the allocation and deallocation of the buffer space are important. In servicing a send command from its module, the server first allocates the buffer and then attempts a **ready** synchronization with the destination module's server, using the size of the transfer as a determinate pattern to indicate the size of buffer. On the destination side, the server will not perform the **ready** synchronization with sending server unless it can simultaneously perform the appropriate **get** synchronization with its module, and vice versa, which is represented in the specification by a composite synchronization in the guard. Once the composite synchronization has taken place, the destination schedule S2 deallocates the buffer, and resynchronizes with its module, indicating successful completion. The originating module M1 and its schedule S1 are also involved in this synchronization because the transfer is not complete unless the destination M2 has received the data. The only possible failures in this model are in the allocation and deallocation of the buffer. If the allocation fails, the sending server aborts before the **ready**, and its module is blocked forever by the resynchronization. The destination server prevents its module from performing the receive command since the **ready** in the composite synchronization prevents the **get** portion of the composite synchronization from ever taking place. This is illustrated through

the specifications for all levels of a local transfer on processor P0 for an array of size 10 from module M1 (the sender) with server S1 to module M2 (the receiver) with server S2, as shown in Figure 17.

```
M1::

   send.M1.M2 $$ (10)  M2!A;

S1::

   send.M1.M2 $ (n') →
      allocate.P0.M1 @ (n);
      [   success.P0.M1 @ → skip; @@
      #   failure.P0.M1 @ → @@; abort
      ];
      ready.P0.M1.M2 $ (n);
      $$

M2::

   get.M2.M1 $$ (10)  M1?B;

S2::

   get.M2.M1 $ (n') ready.P0.M1.M2 $ (n) →
      free.P0.M2 @ (n);
      [   success.P0.M2 @ → skip; @@
      #   failure.P0.M2 @ → @@; abort
      ];
      $$

P0::

[  allocate.P0.M1 @ (n') →
       [   M+n≤k → M:=M+n; success.P0.M1 @
       #   M+n>k → skip;       failure.P0.M1 @
       ]; @@
#  free.P0.M2 @ (n') →
       [   M+n≤k → M:=M+n; success.P0.M2 @
       #   M+n>k → skip;       failure.P0.M2 @
       ]; @@
]
```

Figure 17: Comprehensive specification for local transfer from M1 to M2.

For a distant transfer, the scheme is slightly more complicated. First, the sending server must allocate a buffer of the appropriate size for use in the transfer, after which it must direct its processor to initiate a transfer of the given size. Once this has been accomplished, it then directs its processor to send the data (just the size in this model) to the receiver. Finally, after this is done, the sender deallocates the buffer space and resynchronizes with the sender module indicating completion of the send command. On the destination side, the server accepts the initialize from sender and the receive command from its module simultaneously, thereby obtaining the data size of the following transfer. Following this it allocates a temporary buffer of the designated size and then allows the actual transfer to take place. Lastly, it deallocates the buffer space and then resynchronizes with its module to mark the completion of the receive command.

The specification of this operation is similar to that for the local transmission and is shown in Figure 18. The sending server S1 first obtains the data size from the sending module M1 through the synchronization (**send**) indicating the start of a send command. It then uses

the processor P1 to allocate the buffer (through the **allocate** synchronization) and to send the initialization message to the destination (**ready** synchronization). After the processor resynchronizes indicating that destination has gotten the initialization, the transfer is commenced (**pass** synchronization). When this completes, the sender deallocates the buffer (**free**) and then resynchronizes with its module to mark end of send command for the module. On the destination side, the server S2 must perform the **ready** synchronization, accepting the initialization from the sending server simultaneously with performing the appropriate **get** synchronization with its module M2 to begin the receive command, represented in the specification by the composite synchronization of **ready** and **get** in the guard. Once this is accomplished, it allocates the buffer (the size of which it obtained from **get**), directs the transfer to take place (**pass**), deallocates the buffer (**free**), and then resynchronizes with its module indicating completion of the receive command. This is illustrated by the specifications for all levels of a distant transfer for an array of size 10 from module M1 (the sender) with server S1 on processor P1 to module M2 (the receiver) with server S2 on processor P1, as shown in Figure 18.

Formally, a pair of matching application software I/O commands may either accomplish a successful synchronous data transfer or may block. The model described in Figure 18 is consistent with this definition, but goes one step beyond. By revealing the sequence of actions required to accomplish the apparently synchronous communication between the modules, it also shows the reasons why blocking may occur: because a matching pair of hardware I/O commands is not executed, because of lack of buffer storage, because of a mismatch in data sizes, etc.

```
M1::

   send.M1.M2 $$ (10)  M2!A;

S1::

send.M1.M2 $ (n') →
   allocate.P1.M1 @ (n);
   [  success.P1.M1 @ → skip; @@
   #  failure.P1.M1 @ → @@; abort
   ];
   s_ready.P1.P2.M1.M2 @ (n); @@;
   s_pass.P1.P2.M1.M2 @ (n); @@;
   free.P1.M1 @ (n);
   [  success.P1.M1 @ → skip; @@
   #  failure.P1.M1 @ → @@; abort
   ];
   $$

P1::

[  allocate.P1.M1 @ (n') →
      [   M+n≤k → M:=M+n; success.P1.M1 @
      #   M+n>k → skip;       failure.P1.M1 @
      ]; @@
#  free.P1.M1 @ (n') →
      [   M+n≤k → M:=M+n; success.P1.M2 @
      #   M+n>k → skip;       failure.P1.M2 @
      ]; @@
#  s_ready.P1.P2.M1.M2 @ (n') P2!n →
      skip; @@
#  s_pass.P1.P2.M1.M2 @ (n') P2!n →
      skip; @@
]
```

```
M2::

   get.M2.M1 $$ (10)  M1?B;

S2::

get.M2.M1 $ (n') r_ready.P1.P2.M1.M2 $ (n) →
   allocate.P2.M2 @ (n);
   [  success.P2.M2 @ → skip; @@
   #  failure.P2.M2 @ → @@; abort
   ];
   r_pass.P1.P2.M1.M2 @ (n); @@;
   free.P2.M2 @ (n);
   [  success.P2.M2 @ → skip; @@
   #  failure.P2.M2 @ → @@; abort
   ];
   $$

P2::

[  allocate.P2.M2 @ (n') →
      [   M+n≤k → M:=M+n;  success.P2.M2 @
      #   M+n>k → skip;       failure.P2.M2 @
      ]; @@
#  free.P2.M2 @ (n') →
      [   M+n≤k → M:=M+n; success.P2.M2 @
      #   M+n>k → skip;       failure.P1.M2 @
      ]; @@
#  r_ready.P1.P2.M1.M2 @ (n') P1?n →
      skip; @@
#  r_pass.P1.P2.M1.M2 @ (n') P1?n →
      skip; @@
]
```

Figure 18: Comprehensive specification for distant transfer from M1 to M2.

The specification of the allocation in the static case consists entirely of defining the function HOST_OF, which is referenced in the schedules whenever the host of a schedule must be known. One use of this in the actual schedules is to determine whether a send is local or distant by comparing the hosts of the two modules (schedules) involved in the transfer. In the sample system, this definition involves the allocation of the four modules to the two identical processors, which are capable of supporting all commands required by any module. Since distance.1 and distance.2 are the only modules that are totally independent of each other and can run concurrently, they are logical choices for allocation to separate processors. This leaves selector and minpath to allocate, and this is done by arbitrarily placing them on host.1 and host.2, respectively. The specification of this allocation for the sample problem is given in

Figure 19. To check if some server S is local or remote, all that is required is a boolean expression of the form

$$HOST\_OF(S) = HOST\_OF(\tilde{}).$$

HOST_OF : ARRAY(SCHEDULE) OF PROCESSOR CONSTANT :=
(server.selector : host.1, server.distance.1 : host.1,
server.minpath : host.2, server.distance.2 : host.2);

Figure 19: Specification of static allocation for sample problem.

If dynamic reallocation is desired, the schedule is the only level that is affected. Modelling the reallocation is accomplished by changing the HOST_OF function mentioned above to reflect the new processor to which a schedule (and hence its module) is allocated. The decision as to when and where to relocate is, in general, specific to the model being used. In addition, it may be done globally by a centralized resource, modelled as another schedule, or locally by a distributed algorithm, modelled as a part of each server schedule or a new schedule for each processor to handle all relocations on that processor. This choice impacts both the complexity of the schedule and the robustness of the system as a whole. In the sample model, when an allocate fails, reallocation could be used to move the schedule to a processor with enough free memory to serve the request, so that the schedule and its module could continue execution. Since there are only two processors, a local decision could be made — there is only one other possible processor for any schedule to be allocated to. If the relocation is to occur, then the memory in use on the current processor must be freed after its contents are copied to the memory allocated for this schedule on the new processor , before the schedule can be moved to its new location. Throughout the transfer process, no communications with the moving process may be initiated. The module is effectively suspended during the relocation process.

4.6. Alternate Architecture

One of the benefits of the virtual system model is the ability to assess the impact of hardware choice on the software by trying different architectures with the same functionality. As an example of how this is done, consider replacing the point-to-point communication scheme used in the sample problem by a contention bus (i.e., local area network) to which all (both) of the processors are attached. In this situation, several changes are in order at the architecture level of the specification. First, a model of the bus operation must be developed and included in the specification, and secondly, the communication operations of the processors need to be amended to reflect the changed communication mechanism.

In keeping with the modelling philosophy, the bus is included in the architecture by the specification of a processor describing its operation. One difference is that this processor can not have any schedules or modules allocated to it, since it executes no software. The model for the bus is quite simple, it receives a packet and a destination address from one of the processors attached to it and then attempts to send the packet to the destination. If the destination never does a receive of the packet, the bus process is blocked, and no further communication on it

may take place. There is no concept of time in the model, so no way to model timeout or failure of the communication, resulting in blocking of the bus process (where failure may be more appropriate). Since failure detection and resolution were not of concern in this model, allowing blocking in such cases was sufficient. The bus process can be modelled by a single guarded repetition, with each guard being a receive from one of the processors attached to the bus. As an example, Figure 20 contains the specification of a processor P1 performing a send to processor P2 via a bus that can only perform transfers from P1 to other processors.

```
P1::                                        P2::

    bus!(P2,x) → skip                           bus?(y) → skip


    PROCESSOR bus ::
    [
            b : BOOLEAN := true;
            x : PACKET;
            h : PROCESSOR;

            *[
                    P1?(h,x) →
                            *[ b; h!(x) → b := false]
                            b := true;
             ]
    ]
```

Figure 20: Data transfer from P1 to P2 using the bus.

In the general case, the bus can accept transfers from any of the processors, so the guarded command in the outer repetition of the bus processor would be replicated for each processor connected to it. Using the concise notation described earlier (in the architecture section), the complete bus can be specified as shown in Figure 21.

```
PROCESSOR bus ::
[
        b : BOOLEAN := true;
        x : PACKET;
        source, destination : PROCESSOR;

        *[
                source"?(destination,x) →
                        *[ b; destination!(x) → b := false]
                        b := true;
         ]
]
```

Figure 21: Complete specification of bus processor.

The send to the destination is placed in the repetition structure instead of a selection so that, if the destination process is terminated, the repetition terminates normally, and the bus process continues, where failure of the guard in selection would cause abortion of the bus process. Hence, if the destination is no longer there, the bus just tosses the packet away.

Using this communication structure, the sending processor obtains no information about the arrival of the data at the destination, since the bus may toss it away. As a result, all data sent is in the same form, namely a packet, and so there is no longer any benefit from using two hardware operations for a send (receive) as was done previously, and it is left to the schedules to implement a different protocol to handle the alterations. This simplifies the processor from the earlier example, in that now it need only provide a send operation, which is specified by the **transfer** synchronization, and a receive operation, which is specified by the **receive** synchronization. To illustrate this, the specification of a processor P0 that provides send and receive operations to all other processors is given in Figure 22.

```
PROCESSOR P0 ::
[
        d : PROCESSOR;

        *[      transfer.h.d" @ (x') bus!(d,x) → skip; @@
         #      bus?(x) receive.h.d" @ (x) → skip; @@
         ]
]
```

Figure 22: Processor communication operations using bus.

Both processors involved in the transfer are referenced in the synchronization label that provides the operation to the schedule to ensure uniqueness, as before.

The primary differences between the point-to-point communication and the bus are the introduction of the bus process and the change in operations that the processors can provide. Since the processors are no longer in direct contact, a more complicated mechanism is necessary than the pass and ready commands to perform a transfer. It requires state information to be maintained by the scheduler, but simplifies the communication operations that the hardware must provide. The result is the replacement of the **ready/pass** pair for a send by **transfer** and the replacement of the **ready/pass** pair for a receive by **receive** in the complete processor shown earlier. This solution is more general and the first example could now be retrofitted to follow the same paradigm. We should not have taken advantage of the proximity of the two processors — the advantage of scheduler simplicity was paid for with lack of generality.

4.7. Resultant Changes In Schedule

To use the new communication mechanism provided in the architecture, the scheduler must also change, in particular the way it implements the communication commands it provides to the functionality. Since it is desirable that these commands remain the same from the functionality's point of view, the only changes allowed are in the implementation of the commands. This implementation consists of two parts, the introduction of a schedule process called bus_daemon that interfaces to the hardware communication operation provided by the architecture, and modification of the server schedules to use the bus_daemon to accomplish their communication obligations.

A modified communication protocol is necessary for the servers to use the new communication mechanism. This new protocol is a two phase scheme, with the first phase being a transmission of the data from the sender to the receiver and the second being a transmission of an acknowledgement from the receiver back to the sender. In both the sender and the receiver this only affects distant transmissions, so the local ones remain unchanged. For a distant send, the sender must still allocate a temporary buffer for the transfer, and free it after the transfer is complete, so this much remains the same. Instead of using the **ready** and **pass** synchronizations to direct the processor to do a data transfer, the schedule now gives the data to the bus daemon (on its processor) along with the name of the receiver module and processor, and lets it perform the transfer. The daemon accomplishes this by building a packet containing the sender module, the receiver module, and the data, and then uses the appropriate **transfer** synchronization, with the packet as the determinate pattern, to instruct the processor to send the packet to the receiver's processor. Once the hardware transfer operation is complete, the daemon is free to service another request. The sending schedule now must wait for an acknowledgement from the bus daemon that the data has been successfully delivered to the receiver. Once the receiver obtains the data, the daemon on that processor will send a acknowledgement packet back to the sender, and the receiving module is free to continue work, although the completion will not propagate to the sender until sometime later. In our model, this is not a problem since the acknowledge is guaranteed to reach the sender by the fact that the hardware (including communication system) is failure free. When the daemon on the sender obtains a packet from the processor, it checks to see if it is an acknowledgement, and if it is, it does an output command of the acknowledgement to the appropriate server schedule. After this occurs, the communication is complete, and the server can free the buffer and resynchronize with its module to indicate successful completion. The failure of a send can be caused by (1) the buffer allocation failing (as before), (2) the bus_daemon never accepting the data (i.e. not performing an input from the schedule), or (3) an acknowledgement never coming back. In cases (2) and (3) the schedule is blocked by the input (output) statement with the daemon never

taking place, causing its module to block also. Case (1) is identical to the allocation failure in previous model, the schedule aborts causing its module to block. The distant send portion of a schedule S1 is illustrated in Figure 23 for a send from its module M1 to module M2. S1 is assumed to be allocated to processor P1.

```
S1::
[
        P2 : PROCESSOR;

        *[  send.M1.M2 $ (n') →
                h := HOST_OF(p);
                allocate.h0.f @ (n);
                [   success.h0.f @ → skip; @@
                # failure.h0.f @ → @@; abort
                ];
                bus_daemon.P1!(P2,M1,M2,n);
                bus_daemon.P1?(ack);
                free.P1.M1 @ (n);
                [   success.h0.f @ → skip; @@
                # failure.h0.f @ → @@; abort
                ];
                $$
        ]
]
```

Figure 23: Distant send for schedule assuming bus architecture.

For the receive operation from distant processor, the server schedule is again similar in structure to the previous case. In order for the server to perform a **get** synchronization with its module, appropriate data must already have been received from the designated sender, and be residing in the daemon's queue (described shortly). This is checked by a synchronization (**have_data**) between the server schedule and the daemon that forms a composite with the **get** synchronization. Once this is accomplished, the server allocates a buffer, performs a receive of the data from the daemon, then deallocates the buffer and resynchronizes with its module to indicate completion of the software receive command. This is exemplified by the portion of schedule S2 in figure 24. This portion performs the receive command for its module (M2) from a distant sender M1, where S2 resides on processor P2.

```
S2::
[
        n : INTEGER;
        P1 : PROCESSOR CONSTANT := HOST_OF(M1);

        *[   get.M2.M1 $ (n') have_data.P1.P2.M1.M2 $ (n) →
             allocate.h0.f @ (n);
             [   success.h0.f @ → skip; @@
             # failure.h0.f @ → @@; abort
             ];
             bus_daemon.h0?(n);
             free.h0.f @ (n);
             [   success.h0.f @ → skip; @@
             # failure.h0.f @ → @@; abort
             ];
             $$
        ]
]
```

Figure 24: Distant receive operation in schedule using bus communication.

In delivering data sent from other processors to schedules on its processor, the daemon's operations are broken into two parts. First, it obtains data packets sent to its processor over the bus via the **receive** synchronization with the processor. Once it obtains such a packet, it decodes it to determine who the sender was, who the destination is, and the data. If the data is an acknowledgement, it performs a send of the acknowledgement to the designated schedule, and discards the packet. Otherwise it places the data in the data queue for the destination, overwriting anything that was there (the daemon maintains a separate one deep queue of data corresponding to each possible sender, for each schedule allocated to its processor). It also notes that new data has arrived, so that it can later synchronize (**have_data**) with the appropriate schedule when that schedule is ready for the data. The second part is the delivery of the data from the queue to the schedule when requested. If new data is available, and the **have_data** synchronization takes place, indicating that the schedule wants the data, the daemon does a send of the data to the schedule. Following this, it marks the data as old and sends an acknowledgement message back to the sender of the data, telling it that the data has been picked up by the receiver schedule. Hence, the daemon performs three basic operations, sending a packet of data for a local schedule, receiving packets destined for local schedules from the processor and storing them until they are requested, and finally delivering stored packets to local schedules when requested. The daemon can be specified by a guarded repetition containing three generic guarded commands for the three operations.

In summary, there are several key items of note. First, the specific details of the bus communication are localized to the daemon. This allows maximum flexibility for change with minimum impact on the rest of the scheduler, as can be seen in the example. Should another communication mechanism be desired, its properties could be encapsulated in a similar type process, with little or possibly no change to the scheduler. Secondly, in the altered version, the entire communication protocol is encapsulated in the scheduler, where in the first version, it was

partly in the hardware as two send operation and two receive operations. Also, despite the changes in the architecture and resultant changes in the way commands are executed by the scheduler, the functionality remained completely intact, including the semantics of the application software communication commands. It is this ability that allows many architectures to be mated with the same functionality for analytical comparison of the different systems for solving the same problem. Finally, it demonstrates how changes in the architecture are absorbed by changes in the scheduler to give an equivalent system, very much analogous to the changes necessary in a conventional operating system to move it from one processor to another or to include a new device in the system.

5. CONCLUDING REMARKS

The motivation for the work presented in this paper is rooted in the concept of a *Total System Design (TSD) Framework* [3]. The TSD Framework treats distributed systems as complex hardware/software aggregates whose design often requires careful consideration of the relationship between functionality and architecture in order to meet highly demanding constraints. The *virtual system* is a model that attempts to capture precisely the essence of this relationship.

The definition of the virtual system is motivated by traditional software engineering considerations such as separation of concerns and complexity control. 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 communities of communicating processes used to model the components capture design decisions that may be evaluated (up to a certain point) independently from the overall design or in a limited context. Dependencies between these components are expressed via synchronization rules that tie them together; e.g., functionality, architecture, and scheduler (all having independent existence) are brought together by the allocation.

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.

The first formal definition of the virtual system [4] was developed from a system theoretical perspective and lacked practicality, analyzability, and parsimony. The use of processes and event synchronization has remedied the situation and brought us closer to what we envision a distributed system design language ought to provide. Most recent efforts have been directed primarily toward the development of a powerful, compact, unobtrusive, and easy to modify notation and the definition of its semantics. This paper illustrates some of the choices we have made.

The language design work has been driven by the case study described here.

6. REFERENCES

[1] Roman, G.-C. and Day, M. S., "Multifaceted Distributed Systems Specification Using Processes and Event Synchronization," *Proc. of 7'th Int'l Conf. on Soft. Eng.*, pp. 44-55, 1984.

[2] Hoare, C. A. R., "Communicating Sequential Processes," *CACM* 21, No. 8, pp. 666-677.

[3] Roman, G.-C. et al., "A Total System Design Framework," *Computer* 17, No. 5, pp. 15-26, May 1984.

[4] Roman, G.-C. and Israel, R.K., "A Formal Treatment of Distributed Systems Design," *Requirements Engineering Environments,* Ohno, Y. (editor), pp. 3-12, OHM/North-Holland Pub. Co., 1982.