Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

MIDAS: INTEGRATED DESIGN AND SIMULATION OF DISTRIBUTED SYSTEMS

Rajive L. Bagrodia
Chien-Chung Shen

September 1990
CSD-900027

MIDAS: Integrated Design and Simulation of Distributed Systems[1]

Rajive L. Bagrodia

Chien-Chung Shen

3531 Boelter Hall

Computer Science Department

University of California at Los Angeles

Los Angeles, CA 90024.

Tel: (213) 825-0956

bagrodia@cs.ucla.edu

cshen@cs.ucla.edu

[1]This research was supported by a grant from Hughes Aircraft Co.

Abstract

Iterative transformation of performance models into operational systems is a desirable approach for designing performance-critical systems. This paper describes an approach to transform a hybrid model into an operational system that can be shown to satisfy its performance specifications. A hybrid model merges operational modules with abstract, simulation modules. The capabilities supported by the methodology include (a) interrupt-handling, (b) integration of distributed software and hardware components and (c) evaluation of the effect of upgrading existing hardware components. The paper also describes a language to program hybrid models and shows how simulation algorithms may be adapted to execute hybrid models. A small application was designed using this methodology, and the experimental results of the exercise are described.

1 Introduction

Many distributed systems have stringent constraints on their performance. The critical nature of these systems often requires that their performance characteristics be determined rigorously, preferably in the early stages of system design. In order to ensure that a proposed design will meet its performance specifications, the behavior of the system is usually abstracted by an analytic or simulation model. The complex interactions among the software, hardware and human components of many distributed systems typically preclude the construction or solution of detailed analytical models. Even when approximate analytical models can be constructed, lack of sufficient data precludes a comprehensive analysis of the performance issues. The usual alternative to analytical models is to use simulations. For large, complex systems, simulation models are themselves complex and hence expensive to develop. Subsequent modifications in system specifications necessitate a corresponding change in the simulation model. Model validation becomes difficult leading to doubtful consistency between the model and the actual system being developed. Traditional software design methodologies do not lend significant support in the design of such systems.

The MIDAS approach to system design suggests the use of Partially Implemented Performance Specifications (PIPS) as performance models for distributed systems: In this approach, a model is viewed as a hybrid system that consists of operational software and hardware components interspersed with logical abstractions of other hardware and software subsystems. The hybrid model is iteratively transformed into an operational system that can be shown to satisfy its functional and performance specifications. The thrust of this research is not towards system *specifications*; rather it is to predict whether a proposed system design will satisfy performance requirements and to transform the design into an operational system.

What advantages may be derived from the integrated approach to system design? The performance model of a system does not have to be designed and maintained separately. *The evolving design is its own model.* This can result in significant savings in terms of manpower and resources while at the same time ensuring the consistency between the software and its model. This is a realistic alternative to traditional modeling strategies as it allows non critical subsystems to be implemented prior to being modeled, and incorporates their effect in the performance profile for the overall system. Moreover, it allows the performance of existing subsystems to be integrated with their evolving counterparts. In addition to performance evaluation, the hybrid system is also useful for system testing and debugging.

This paper describes the MIDAS approach to the design of distributed systems and discuss theoretical, language design and implementation issues in its use. The next section discusses related work in the area of integrated design. Section 3 describes the MIDAS approach. Section 4 describes

how sequential simulation algorithms may be adapted for the execution hybrid models. Section 5 indicates the relationship between the execution of hybrid models and space-time simulation[5]. This section describes how the sequential algorithm may be adapted for interrupt-handling and integration of high-speed operational modules. Section 6 describes how an existing simulation language can be adapted to program hybrid models. Section 7 discusses implementation issues and presents some experimental results on the design of a simple application using MIDAS. Section 8 is the conclusion.

2 Related Work

Performance modeling in the design stage has frequently been used in the design of hardware systems. The idea of integrating simulation models with system design was used by Zurcher and Randell[18] to develop a methodology for the design of computer systems and also explored by Parnas[12]. Sanguenetti[15] describes a technique for performance prediction by integrating simulation and software system design using PPML[13], a system modeling language. Our work extends the applicability of the integrated approach in significant directions: the PIPS methodology supports interrupt-handling. Further, it supports the execution of hybrid models which include physical components that execute considerably faster than the logical one. Finally, the approach permits an analyst to directly determine the consequence of upgrading existing hardware components.

Other researchers have suggested methodologies and tools to construct performance models *prior* to developing the system. Chandy et al [4] describes a top-down methodology for evaluating the performance of computer and communication systems in the early design stages. Estrin et al[6] suggest a system design methodology to study performance aspects in multiple domains: a control flow graph is constructed to analyze safety and liveness properties. Also, a data flow graph may be used in conjunction with the control flow graph to study performance characteristics by interactive simulation. Roman[14] describes a specification language called CSPS, to study correctness and performance characteristics of distributed systems. CSPS is an extension of CSP[8] and uses verification techniques that have been developed for CSP programs to prove properties of the CSPS programs. It also uses the notion of multiway synchronization to map software modules on a model of the proposed hardware. Although the preceding approaches encourage modular development of models and its iterative refinement into operational systems, they do not support the execution of hybrid models, where performance measurements include both simulation and operational modules.

3 MIDAS Approach

A distributed system consists of a collection of communicating sequential processes that execute concurrently on a number of processors linked by an arbitrary interconnection network. A processor may interleave the execution of multiple processes. Processes communicate exclusively via messages.

In a simulation model, each process (pp), or a collection of processes, in the physical system is abstracted by a logical process (lp). In a hybrid model, each process is either an operational module (also called a physical process or pp) or a logical process. Note that pp is used to refer to a physical process in the real system as well as an operational module in the hybrid model, as the two must be identical. We refer to a *computation step* as the sequence of instructions executed by a pp in response to a message. In contrast, a *simulation step* models or simulates the activities that

would be executed by the corresponding physical process. A pp executes only computation steps, whereas a partially elaborated lp may execute a computation step to respond to some messages and a simulation step in response to others. Consider a file-handler process. On receiving a *read* request for the file, an operational file-handler will read the appropriate record from the file and return it to the requesting entity. If the file-handler is abstracted by a logical process, on receiving a *read* request, the lp estimates t, the time required for the corresponding physical process to read the file and simulates the read action by waiting for t units of simulation time to elapse. The MIDAS approach supports the iterative transformation of a model (where the abstractions are expressed as simulation steps of lps) into an operational system; at every intermediate step, the expected performance of the system can be monitored to ensure that it satisfies system requirements.

Figure 1 displays the MIDAS approach. Given the performance specification for a proposed system and an initial system design, an analyst develops a simulation model of the software that has not yet been implemented (or whose performance is considered to be critical) and a model of the hardware subsystem that is not yet available. The initial model may be refined in a variety of ways: decomposing a module into more modules, elaborating a simulation step into a computation step, elaborating the required processing for a message, or replacing a model of some hardware unit by actual hardware. The above process of refinement and elaboration implies that at some intermediate stage, the model may contain some modules that are (at least partly) operational, and the computation steps of these modules must be included in determining the overall performance characteristics of the evolving system. The intermediate form of the model is referred to as a PIPS program or model. The model is refined iteratively, while its performance is continuously monitored until the (software) model has been transformed into operational code. Note that either software or hardware models may be replaced by operational modules.

When is the integrated approach to software design useful? In order to include both operational and simulation modules in measuring performance, the simulation 'engine' (the architecture on which the model is executed) must be similar to the proposed hardware for the modeled system. If the actual hardware is available, our approach will yield maximum benefit. However, even if some characteristic hardware parameters like clock speed, memory and disk access times, instruction cycle time ... are scalable with respect to the simulation engine, our approach will be of significant help in predicting system performance.

In a hybrid model, modular decomposition of the software should be distinguished from its mapping on some hardware. Multiple processes in the model may in fact be executed sequentially on a common processor in the operational system. To allow for easy experimentation with a variety of mappings for the software components, we introduce the concept of a Logical Element (LE). Each PIPS process is mapped to a specific LE; multiple processes may be mapped to a common LE. The simulation or computation steps of all processes mapped to a common LE are executed sequentially, and those mapped to different LEs are executed (logically) in parallel. Each LE is associated with a unique *virtual* clock, where $clock_i$ refers to the virtual clock for LE_i. Unlike a simulation, the virtual clock of an LE may be advanced by the execution of a computation or simulation step of any process mapped to the LE.

simulation can be made deterministic by

4 Execution of PIPS Models

PIPS models may be executed on a *centralized* simulation engine by adapting a sequential simulation algorithm, as discussed in this section. This algorithm is sufficient to execute PIPS models that

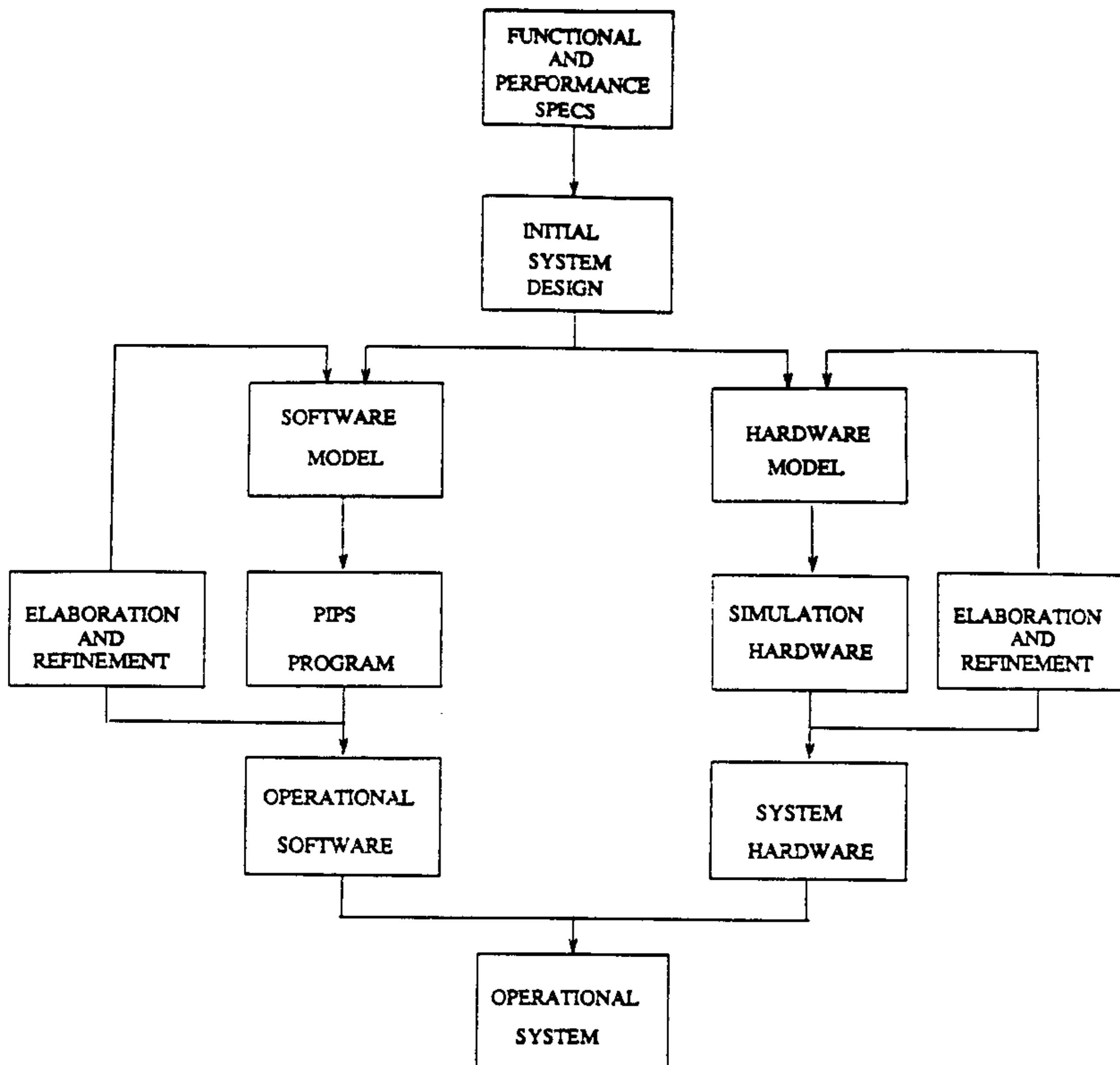

Figure 1: Software Development from a Performance Perspective

consist of a set of interacting, *non-interruptible* processes.

4.1 Simulation Algorithm

In message-based simulation, each physical object is abstracted by an lp, and interactions among the objects (known as events) are represented by message communications among the corresponding lp. Message-based simulation algorithms use two data structures[10]: a *simulation clock* and an *event-list*. The simulation clock gives the time up to which the physical system has been simulated. The event-list is a partial order of tuples; a tuple is represented by (m_i,p_i,s_i,t_i), where m_i represents a message, s_i and p_i the source and destination lps for m_i, and t_i is a timestamp. The partial order is typically based on the timestamp and ensures that events are simulated in the order of their dependencies. At every step of the simulation, the algorithm selects the tuple with the smallest timestamp, say (m_i,p_i,s_i,t_i), and delivers m_i to p_i. The simulation of m_i by lp p_i may generate

additional messages which are added to the event-list.

During the execution of a simulation program, the simulation clock advances in a monotonic non-decreasing manner through the timestamps associated with each tuple. How is the timestamp assigned to a message? When a message is generated, it is timestamped with the current value of the simulation clock — with one exception. We define a special message called a **timeout** message. The timeout message is scheduled by a process for delivery to itself at a future time and is typically used by an lp to simulate the time that would be required by the corresponding physical process to provide the desired service. For example, on receiving a request from a job which requires t units of service, a *server* lp may schedule a timeout message to itself t time-units in the future. As the timestamp on all messages other than the timeout message refers to the current value of the simulation clock, the simulation time advances only when a timeout message is delivered to an lp.

The timeout message in a simulation is a conditional message which is canceled and rescheduled by a process, if it accepts some other message in the interim. For instance, consider a *server* lp that models a preemptible process. While serving a request say *low*, if the *server* receives a request *high*, that has a higher priority, *low* must be preempted. To preempt service of *low*, the *server* cancels its previously scheduled timeout message, and reschedules the message to correspond to completion of service for *high* (service of *low* is resumed subsequently). Many variations of the basic algorithm described above are in use. A common variation is the wait-for simulation algorithm[7] in which a process may delay acceptance of a message based on its state or the message contents. Another variation[10] allows the timestamp on every message to refer to the (future) time at which it is to be delivered. These differences do not concern us here as our emphasis is on establishing the similarities between the execution of simulation and hybrid models.

As a simple example, consider the simulation of a producer consumer program. In the program, a producer process (*p1*) produces data and sends it to a buffer process (*b1*), using a message called *put*. A consumer process (*c1*) requests data from the buffer process via a message called *get*, and is blocked if the buffer is empty. On receiving a *get* message, a non-empty buffer sends a data item to *c1* in a message called *data*. The service times for both *p1* and *c1* are generated in the simulation from a negative exponential distribution. Assume that process *p1* takes 4, 6, 8, and 3 time units respectively to generate successive units of data; *c1* takes 8, 2, 6, and 9 time units respectively to consume successive data units. The simulation model assumes that the time required by *b1* to process the *put* and *get* messages is negligible and can be ignored. Also, the time required to exchange messages between the processes is ignored in the model. Figure 2 shows a time-line diagram for the first few events in the simulation. Each simulation step corresponds to the production or generation of a data item, where d_i and r_i respectively represent the time required to generate or consume the ith data-item.

4.2 PIPS Algorithm

As a partially implemented program contains both simulated components and implemented components, it is required that the run-time environment be able to distinguish the execution of a simulation step from that of a computation step. By definition, if a process receives a *time-out* message it executes a simulation step; otherwise it executes a computation step. In simulations, the clock is advanced only by the duration specified in a timeout message. If the execution time of a computation step is important for performance measurements, its duration must be modeled by means of a timeout message. In contrast, the computation step of an operational or partially implemented module may be directly included in the performance measurement of a PIPS program.

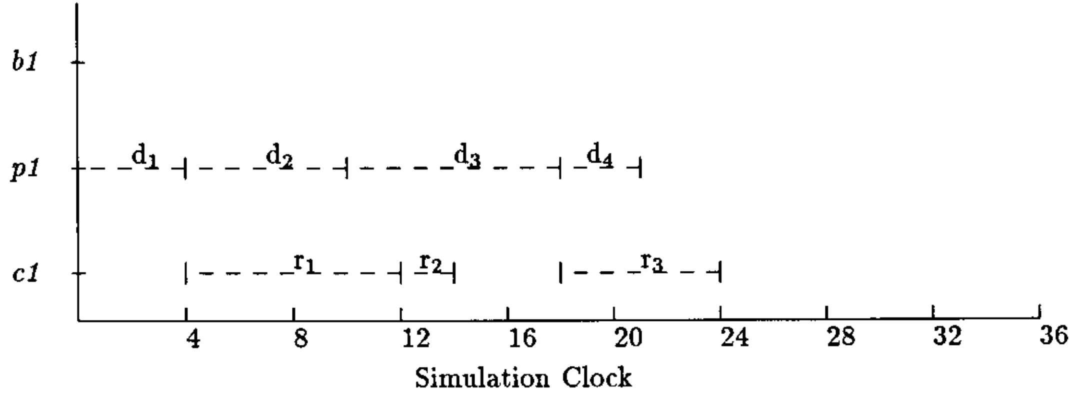

Figure 2: Simulation of a Producer Consumer Program

A PIPS algorithm (*ie* an algorithm to execute PIPS models) also uses two main data structures: a set of virtual clocks, one for each LE in the program, and an event-list. Messages from the event-list are delivered to destination lps in the order determined by their timestamps. The message timestamps are generated on the basis of the virtual clock of the transmitting process. (We use virtual clock of a process to mean the virtual clock associated with the LE to which the process has been mapped.) Unlike the simulation algorithm, the PIPS clock may be incremented by messages other than the timeout message. When a timeout message is delivered to a process, its clock is simply incremented by $\triangle t_s$, the duration of the simulation step specified in the message. For other messages, the clock is advanced by an interval proportional to the duration of the computation step executed by the process (the proportionality factor represents the difference in speed of the simulation engine and the proposed hardware). A centralized algorithm to execute PIPS programs is presented in figure 3. In the figure, *clock* refers to the PIPS virtual clock whose value indicates the time up to which the hybrid PIPS model has been executed. Each LE is associated with a unique scaling factor *f*. As seen from the algorithm, the computation steps of a process mapped to an LE is scaled by *f* before incrementing the clock. Execution of the model terminates when the event-list is empty, or the earliest timestamp in the event-list is greater than T, the maximum time for which the model is to be executed.

In order to prove that the PIPS algorithm is correct, we are required to show that the sequence of messages, say $\mathcal{L}$, generated in the execution of a PIPS program is the same as $\mathcal{P}$, a sequence of messages generated by the physical system being modeled. Under the assumption that the computation step of a process is atomic, the correctness proof for the PIPS algorithm may be derived in a manner similar to the proof of the sequential simulation algorithm in [10] and is omitted.

We use our running example of a producer consumer program to illustrate execution of a PIPS model. Assume that the code for the consumer and buffer processes is operational and executes in physical time, whereas the producer process is only partially implemented and executes in simulation time. (Note that in most simulation languages, the code for the *buffer* lp is almost identical to that used to describe an 'operational' buffer process. This is a simple example of the potential efficiency that may be exploited by integrating system design with its performance evaluation). In the PIPS program, we assume that the execution of the computation step for *c1* takes 5 time units, and that for *b1* is 3 time units. The service time for the producer entity is sampled from an exponential distribution, and the first few values are again assumed to be 4, 6, 8 and 3 respectively. We consider two simple mappings of the PIPS program: one where each process

```
clock:=0;
Initialize the event-list;
while (execution not terminated) do
{ fetch next tuple (m_i,p_i,s_i,t_i) from event-list;
  if (m_i=timeout) then
  { clock_i:=clock_i+Δt_s(delay specified in the timeout message);
    clock:=clock_i;
    p_i simulates processing of m_i;
  }
  else {clock_i:=max(t_i,clock_i);
          clock:=clock_i;
          p_i processes m_i; measure Δt_c (duration of computation step);
          clock_i:=clock_i+Δt_c*f_i;
  }
}
```

Figure 3: Centralized Algorithm to execute PIPS Models

is mapped to a separate LE, and the other where all three processes are mapped to a common LE. Note that in the simulation model, the time required by the buffer to process a message was *not* modeled by lp *b1*, as it was considered incidental to the system performance. We include this time in the PIPS model only to illustrate how the performance of operational modules is included in the measurements. At the end of the section, we indicate how the computation step of a PIPS process may be *excluded* from the measurements.

Figure 4 shows the time line diagram for the execution of the PIPS program where each process is mapped to a separate LE (the processes will execute in parallel in the operational system). The simulation steps are shown by a dashed line and the computation steps by a continuous line. Consider the first few events from figure 4. As in the simulation, the first event is the receipt of a *get* message by *b1*. Unlike the simulation, the computation step executed by *b1* advances its clock by 3 time units. The next event is the delivery of a timeout message to *p1*. This message was scheduled by *p1* when its clock was 0. As each process is mapped to a unique LE, delivery of the timeout message simply advances the the clock of *p1* to 4 units. On receiving this message, *p1* generates a *put* message for *b1* at its current time of 4 units. Delivery of this message to *b1* advances its clock to 4 (the timestamp on the *put* message). The computation step executed by *b1* further advances its clock by 3 time units to 7.

We next consider the situation where the computation step of an entity overlaps with the simulation step of another, as is the case with d_2 and r_1 in figure 4. If the computation step of a process is assumed to be *atomic*, overlapping steps may be processed out of order without affecting correctness. In the above example, even though the simulation step labeled d_2 completes before the computation step labeled r_1, the PIPS algorithm of figure 3 will deliver the timeout message to *p1* *after* *c1* has executed r_1. As any subsequent messages generated by *p1* (or some other process) for delivery to *c1* can only be processed by *c1* after it has completed r_1, the algorithm works correctly. However, if the computation step of *c1* could be interrupted due to the generation of a message by *p1* on completion of its simulation step, the preceding algorithm is inadequate. The integration of

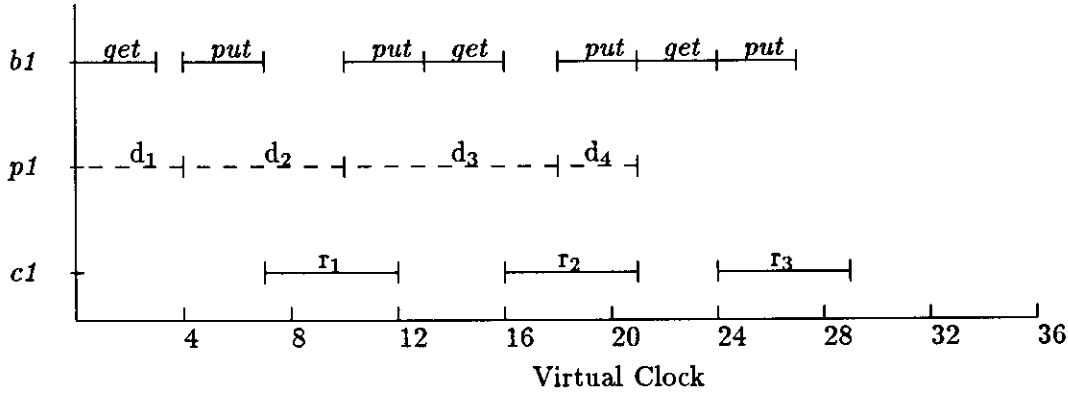

Figure 4: Execution of PIPS Model: Parallel Mapping

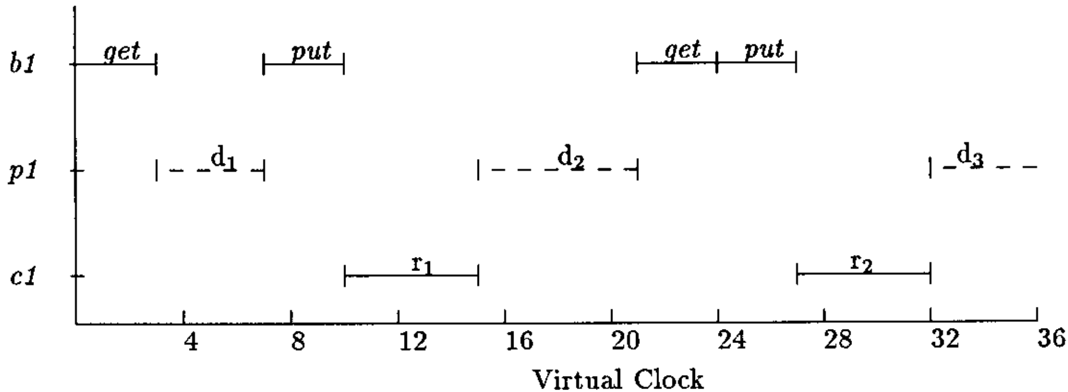

Figure 5: Execution of PIPS Model: Single LE Mapping

an interruptible computation step in a PIPS model is considered in the next section.

We now consider a different mapping of the PIPS model that represents a centralized implementation of the producer consumer program. The new mapping implies that the computation and simulation steps of all three processes must be executed sequentially. This configuration may be tested with minimum modifications to the program; it is sufficient to simply ensure that all processes are mapped to a common LE. The sequential nature of the program is clearly visible in the time line diagram of figure 5. In particular note that the first timeout message, which was delivered at time 4 in the previous mapping, is delivered to *p1* when its clock reads 7 time units. This is due to the prior execution of the computation step by *b1* which advanced the common clock to 3.

The execution time of every computation step in a PIPS program may not be relevant to the performance of a system. Consider, for instance, a statistics collection entity, like the histogram entity. Presumably this entity will not be a part of the eventual system, and the execution time of its computation step should not be included in the performance metrics being collected for the system. We define le_0 to be a null-valued LE. For all entities mapped to le_0, the execution time of the computation steps are ignored and the simulation steps are executed in parallel. A PIPS program may be executed entirely as a simulation, simply by mapping all entities in the program to the logical element le_0.

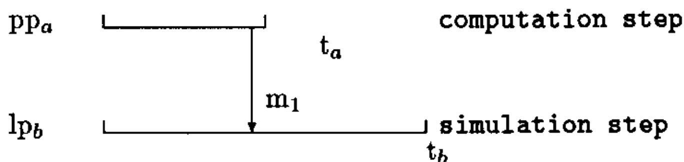

Figure 6: Conditional simulation step

5 PIPS and Space-Time Computation

The previous section described an algorithm to execute PIPS models on a centralized architecture, where the computation step of a process is assumed to be atomic. In this section, we include the execution of models whose computation steps are interruptible. In addition, we also indicate how PIPS models may be executed on distributed hardware.

5.1 Interrupts and PIPS

Atomicity of a computation step makes it infeasible to elaborate a conditional simulation step – a simulation step scheduled by a conditional timeout message. Once again, consider a preemptible program module. An lp simulates such a module by scheduling a conditional timeout message which is rescheduled if the entity accepts another message with a higher priority before completion of the simulation step. To elaborate such models into operational modules, the language in which the operational modules are expressed must allow a message to interrupt the execution of the destination process. We refer to such a message as an *interrupt-message* or simply as an *interrupt* with the understanding that it refers to a message. As illustrated by the following example, elaboration of a conditional simulation step into a computation step is still infeasible, unless the arrival of interrupts can be predetermined. Consider two processes pp_a and pp_b that execute overlapping computation steps of duration t_a and t_b, where $t_a < t_b$. Process pp_a sends an interrupt m1 to pp_b, which must be processed by pp_b before the completion of interval t_b. In the PIPS model of this program, pp_a exists as an operational module, and pp_b is abstracted by the logical module lp_b (figure 6). To model parallel execution, the two processes are mapped to separate LEs in the PIPS program. However, the available simulation engine is a uniprocessor and the execution of the two processes must be interleaved.

When this PIPS model is executed, pp_a executes a computation step and lp_b a simulation step. To execute its simulation step, lp_b schedules a conditional timeout message at time t_b. The PIPS algorithm will execute the computation step of pp_a before delivering the timeout message to lp_b. As the timestamp on message m1 generated by lp_a is less than that of the timeout message scheduled for lp_b, m1 will be delivered to lp_b before the timeout message, as desired.

Now consider a further refinement of the model, where the simulation step of lp_b has also been elaborated into a computation step. In order to correctly process message m1, pp_a and lp_b must be interleaved such that the computation step of pp_a is executed first. However in general, this dependency cannot be known until the computation steps are actually executed! In the above example, if the order of interleavings is reversed, then message m1 has not been generated while lp_b is in execution, and obviously cannot be processed. In addition, the above scenario can be made arbitrarily complex by defining a PIPS model that must interleave the execution of two mutually interruptible computation steps(figure 7).

The preceding problem may be solved with the selective use of checkpointing and recomputa-

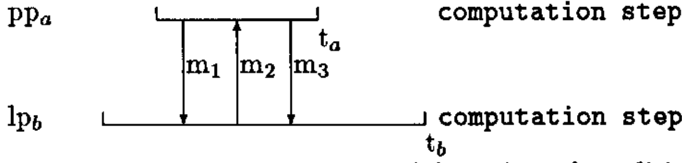

Figure 7: Elaboration of conditional simulation step

tion. For simplicity, we restrict our attention to interruptible PIPS models wherein each process is mapped to a unique LE. We will use c_i to denote some interruptible computation step and m_e to denote an interrupt-message. The state of a process, say p_i, is checkpointed whenever it executes some c_i. Subsequently, if some other entity generates an m_e that must be processed before completion of c_i, p_i is rolled back to its checkpointed state. It is subsequently reexecuted such that m_e is processed in its correct order. This is repeated as many times as required until all messages have been processed in their correct order. Algorithms based on checkpointing and roll back have already been suggested for distributed simulation[5, 9]. These algorithms use recomputation to decrease synchronization delays in a simulation and improve its completion time. In our case however, checkpointing and recomputation is used to unravel dependencies among the (logically parallel) computation steps of multiple lp that must be interleaved on a single processor. Note that unlike in distributed simulation, checkpointing is needed even in the execution of a PIPS program on a centralized simulation engine. However, checkpointing and reexecution overheads may be considerably reduced in a PIPS model by *syntactically* distinguishing between interruptible and non-interruptible computation steps and by *preventing transmission of (potentially) incorrect messages.* This implies that checkpointing is required *only for a process that explicitly indicates that a specific computation step is interruptible.* The appropriate language constructs are described in section 6. In this section, we describe the modifications to the sequential PIPS algorithm to handle interrupts. Note that we are not proposing the use of recomputation to support interrupts in the operational system; a hardware implementation will presumably be used for this purpose. The PIPS interrupt facility allows an analyst to simulate the effect of these interrupts on system performance even though the eventual hardware may not yet be available.

Assume that lp$_b$ executes some c_i of duration (t_b,t_f). Let (m_e,lp$_b$,s_i,t_e), $t_b \leq t_e < t_f$ represent an interrupt that must be handled by lp$_b$ during execution of c_i. We define the following data structures:

- *M*: messages generated by lp$_b$ in its last execution of c_i.
- $M_{b,a}$: the initial subsequence of *M* that consists of messages that precede the interrupt m_e in the event-list. If message timestamps are unique, $M_{b,a}$={$\forall$(m_i,p_i,lp$_b$,t_i) in *M*, such that $t_i < t_e$}. If timestamps are not unique, $M_{b,a}$ includes a message timestamped t_e if it was processed before m_e in the execution of the model.
- $M_{a,f}$: messages from *M* that are not included in $M_{b,a}$; $M = M_{b,a} + M_{a,f}$, where + indicates sequence concatenation.
- *interrupt_q*: interrupts handled in the previous execution of c_i, (these interrupts must precede m_e).

When lp$_b$ is restored to the state checkpointed at t_b, messages that belong to the sequence $M_{a,f}$ are removed from the event-list. Further, messages generated during reexecution of c_i that belong

to $M_{b,a}$ are discarded. The modified PIPS algorithm is described in figure 8. For simplicity, the algorithm assumes that interrupts are disabled during execution of an interrupt-handler. We prove that the modified algorithm correctly executes a PIPS program.

Theorem 1 *Reexecution of an interruptible computation step for a process, say* lp_b*, does not cause any other process to be reexecuted.*

Proof: To prove the result, it is sufficient to show that reexecution of lp_b does not insert into the event-list, any tuple of the form (m_i, p_j, lp_b, t_i), where t_i is smaller than the timestamp on the last message processed by p_j. The tuples in the event-list are processed in (the partial) order of their timestamps. When an interrupt m_e (with timestamp t_e) is scheduled for lp_b, only messages that belong to sequence $M_{b,a}$ have been delivered to other processes in the system. However, during reexecution of c_i, any message generated by lp_b that belongs to $M_{b,a}$ is discarded. This ensures that no other process in the system need be rolled back. *End of proof*

Lemma 1 *If some* c_i *is reexecuted to process interrupt* (m_e, t_e)*, no message timestamped smaller than* t_e *is generated in the reexecution.*

Proof: ¿From the definition of $M_{b,a}$, any message with a timestamp smaller than t_e must belong to $M_{b,a}$, in which case it will be discarded when c_i is reexecuted. *End of proof*

As reexecution of c_i does not change the behavior of any other process in the system, to prove that the modified algorithm correctly executes a PIPS model, we simply need to prove the following properties:

Convergence : Execution of an interruptible computation step will eventually converge.

Safety : The sequence of messages generated by an interruptible computation step that has converged is correct.

The proofs of the safety and convergence properties assume that each component of the physical system has been modeled accurately in the PIPS model. This requirement is referred to as the fidelity property. In addition, we assume that every physical process (and consequently its PIPS model) satisfies the *realizability* and *predictability* properties[10]:

Fidelity Given a sequence of input messages, a physical process and its PIPS model will output the same sequence of messages. The definition is, of course, 'modulo' the statistical and housekeeping messages that are used to execute the model.

Realizability A message sent by a process at time t, is a function of its initial state, t, and the messages received by the process upto and including t.

Predictability The state of a process at time t, cannot depend on the messages sent by it at t.

The convergence and safety properties are proven in theorems 2 and 3 respectively.

Theorem 2 *Execution of an interruptible computation step* (c_i) *will eventually converge.*

Proof: Assume that lp_b executes c_i on receipt of some message m_c. Further, assume that after the kth execution of c_i, one or more interrupt-message for lp_b is present in the event-list.

```
clock:=0;
Initialize the event-list;
while (execution not terminated) do
{ fetch next tuple (m_i,p_i,s_i,t_i) from event-list;
  if (m_i is an interrupt-message) then
  { /*      identify the computation step c_i that is interrupted by m_i,
            its execution interval [t_b,t_f], and the sequences M, M_{b,a} and M_{a,f}.
            Let m_c refer to the message that caused execution of c_i.
    */
    Restore state of p_i checkpointed at t_b;
    add (m_i,p_i,s_i,t_i) to interrupt_q;
    Discard tuples that belong to M_{a,f} from the event-list;
    clock_i:=t_b;
    remove earliest (m_e,p_i,s_i,t_e) from interrupt_q and schedule m_e for clock_i+(t_e-clock_i)*f_i;
    p_i processes m_c and handles interrupts from interrupt_q, measure Δt_c;
    /*      Successive interrupts from the interrupt_q are scheduled and handled
            until the interrupt_q is empty
            Δt_c is the physical time needed to process m_c and the interrupts.
    */
    Messages that belong to M_{b,a} are discarded from the event-list.
    clock_i:=clock_i+Δt_c*f_i;
  }
  else if (m_i=timeout) then
  { clock_i:=clock_i+Δt_s(delay specified in the timeout message);
    clock:=clock_i;
    p_i simulates processing of m_i;
  };
  else {clock_i:=max(t_i,clock_i);
        clock:=clock_i;
        p_i processes m_i; measure Δt_c (duration of computation step);
        clock_i:=clock_i+Δt_c*f_i;
  };
};
```

Figure 8: Modified PIPS Algorithm to handle Interrupts

Without loss of generality, let (m_e,lp$_b$,s_i,t_e) be the interrupt-message with the smallest timestamp in the event-list. (A1)

If we prove that eventually m_e is processed by lp$_b$ and that any interrupt generated subsequently for lp$_b$ must have a timestamp greater than t_e, convergence of c_i can be deduced using straightforward induction.

As m_e is the earliest interrupt, correctness of the PIPS algorithm without interrupts guarantees that eventually all messages preceding m_e will be removed from the event-list and m_e will become the message with the earliest timestamp in the event-list. At this point m_e is removed from the event-list and delivered to lp$_b$. Delivery of m_e initiates the $(k+1)^{th}$ execution of c_i. Assume that subsequently an interrupt message (m_j,lp$_b$,s_j,t_j) exists in the event-list such that $t_j \leq t_e$.

Consider $t_j < t_e$. Due to the realizability property, message m_j must have been generated by p_j on receipt of some m_r timestamped at most t_j. Further, due to assumption A1, m_j must have been generated during or after the $(k+1)^{th}$ execution of c_i. Due to lemma 1, reexecution of c_i does not generate any message timestamped less than t_e. It follows that m_r must have been present in the event-list when m_e was delivered to lp$_b$. As the timestamp of m_r may be at most t_j, and $t_j < t_e$, this violates the assumption that m_e was the interrupt with the smallest timestamp, establishing the necessary contradiction.

Now consider $t_j = t_e$. Assume that interrupts m_e and m_j are generated independently. This implies that reexecution of c_i to handle m_e (or m_j), cannot cause generation of m_j(or m_e). As there can only be a finite number of interrupts timestamped t_e, eventually they will all be processed and the result follows. Assume that m_e and m_j are not independent. Due to the realizability property, prior to generating m_j, s_j must have received some message, say (m_r,t_r) such that $t_r \leq t_j$. As m_e and m_j are not independent, m_r must itself have been generated as a consequence of lp$_b$ handling interrupt m_e which implies that $t_r \geq t_e$. The preceding statements together with the assumption that $t_j = t_e$ imply that $t_r = t_e$. This implies that the message received by lp$_b$ at t_e depends on the messages it sends at t_e, which violates the predictability property. *End of proof*

Assume that execution of some computation step c_i was completed at t_f. The following two conditions are sufficient to deduce that its execution has converged:

- the event-list does not contain any m_e with a timestamp smaller than t_f for lp$_b$.
- *clock*$>t_f$; this ensures that no further interrupts for c_i can be generated in the model.

Theorem 3 *The sequence of messages generated by some c_i that has converged, is correct.*

Proof: We use induction on the length of the message sequence. Let c_i be the first interruptible computation step executed in a PIPS model. Assume execution of c_i was initiated at t_b and completed at t_f. The centralized PIPS algorithm guarantees that execution of the model prior to t_b is correct. At some point after convergence of c_i, let (m_i,lp$_b$,s_i,t_i), $t_b \leq t_i \leq t_f$, be the earliest message in the event-list. Assume that execution of the PIPS model is correct prior to delivery of m_i. In other words, if $\mathcal{L}$ denotes the sequence of messages generated in the execution of a PIPS program and $\mathcal{P}$, a sequence of messages generated by the physical system being modeled.

$\forall t < t_i$, $\mathcal{P}$ and $\mathcal{L}$ are identical. (A2)

Let (m_j,t_j), $t_j \leq t_i$, be the last message received by lp$_b$.

¿From the inductive hypothesis, $\mathcal{L}$ and $\mathcal{P}$ must be identical prior to delivery of m_i in the PIPS model; thus the sequence of input messages to lp$_b$ in the interval [0, t_j] is identical to that of the

corresponding physical process (if $t_j = t_i$, we only include messages upto and including m_j in the subsequences). Suppose (m_i,lp_b,s_i,t_i) is not transmitted in the physical system. This implies that the physical process must have received an interrupt (m_e,t_e), where $t_e \leq t_i$. However, as c_i has converged, it follows that lp_b cannot receive this interrupt. Together with A2, we conclude that such an interrupt cannot be received by the physical process. From the fidelity property it follows that (m_i,t_i) must also be generated in the physical system. *End of proof*

5.2 Distributed Execution

A PIPS model is executed in a distributed architecture by mapping each LE in the model to a specific physical processor (referred to as a PE) in the distributed architecture. This implies that although processes mapped to different LEs may execute on different PEs, processes mapped to a common LE are executed on the same processor.

Consider a system that consists of two interacting, concurrent physical components pp_a and pp_b. Assume that the hardware for the operational system is available and consists of a network of two PEs. In the PIPS model of this system, pp_a is an operational module, whereas pp_b exists as simulation module lp_b. The PIPS model is executed on the available hardware by executing pp_a and lp_b on different PEs. An immediate benefit of the integrated approach is apparent in that the available communication network may directly be used and need not be *modeled* in the PIPS program. If pp_a and lp_b are not cyclically dependent on each other, messages for pp_a may be generated (with appropriate timestamps) in an off-line mode by lp_b and then be input by pp_a as desired. However, the presence of cyclical dependencies makes the execution non-trivial. We show that the dependencies may be unraveled by using an iterative computation method suggested by the space-time paradigm for distributed simulation[5].

In the space-time paradigm, multiple logical processes may be used to simultaneously compute the state of a physical process at different points in time. Let T be the upper bound on the time for which the system is to be modeled. Let $p_i^{1,2}$ refer to the lp responsible for the computation of pp_i in the interval $[t_1, t_2)$, $t_1 < t_2$; exactly one process computes the behavior of the system for every t in [0,T]. A *precedence* relation, symbolized by $\rightsquigarrow$, is defined between two processes, where $p_i^{1,2} \rightsquigarrow p_j^{3,4}$ if and only if the state of $p_j^{3,4}$ depends on the state of $p_i^{1,2}$ or on some message received from $p_i^{1,2}$. If $p_i^{1,2} \rightsquigarrow p_j^{3,4}$, we say that $p_i^{1,2}$ is a *predecessor* of $p_j^{3,4}$ and $p_j^{3,4}$ is a *successor* of $p_i^{1,2}$. Note that although the exact predecessor or successor set for a process cannot be determined a priori, a loose upper bound on these sets can typically be determined (a trivial bound is the entire set of processes in the system).

Given that the preceding set of processes are executed on a distributed architecture, the correct state of each process is computed by using the following iterative strategy: given some state for its predecessor processes, a process p_i computes an *estimate* of its final state. During this computation, it generates a (possibly empty) sequence of messages for each of its successors. The message sequence is sent to each successor after a process has computed its final *estimated* state. When a process gets a message sequence from one of its predecessors that is different from the one it received in its previous iteration, the process recomputes its behavior. This procedure is repeated until eventually the computation reaches a fixed-point where further execution of any process does not change its state, and the computation is said to have converged. A complete description of the algorithm and sufficient conditions for the convergence of the computation may be found in [5].

The previous algorithm is used to unravel the cyclic dependencies between concurrently executing PIPS modules like pp_a and lp_b described above. We use the following notation:

- S_a^i: sequence of messages generated by pp_a (or lp_a) in its ith iteration.
- R_a^i: sequences of messages received by pp_a (or lp_a) after executing its ith iteration.

On receiving R_a^i (from all its predecessors), lp_a executes its $(i+1)^{th}$ iteration and sends the suffix of S_a^{i+1} that is different from S_a^i to its successor processes. Assume that the sequence S_a^i was correct upto some time t_j. The predictability property can be used to deduce that S_a^{i+1} must be correct upto some time t_i, where $t_i=t_j+\Delta$, for $\Delta > 0$. Using straightforward induction, it is possible to show that eventually the correct message sequence can be generated for each module in the PIPS program over any time-interval. We emphasize that the multiple executions of a PIPS process are generated transparently by the distributed PIPS run-time system and need not be created explicitly by the programmer.

6 Language Support

A language called Hybrid Maisie has been designed to develop hybrid programs. This language extends the Maisie simulation language[3] in two ways: it incorporates software interrupts and provides constructs to integrate simulations with physical modules. The discussion in this section emphasizes those features of Hybrid Maisie that are pertinent to developing hybrid programs. A complete description of the Maisie language may be found in [2].

Both physical and logical processes of a hybrid model are represented as entities. An entity-type models objects of a given type. An entity-instance, henceforth referred to simply as an entity, represents a specific object and may be created and destroyed dynamically. An entity can refer to itself using the default keyword **myid**. The language provides two new types, **e_name** and **le_name**, which are used to store entity- and LE-identifiers respectively. When created, an entity may be mapped to a specific LE (by default its LE is the same as that of its creator). All entities mapped to the same LE are executed sequentially. The following code fragment illustrates the creation of a new *server* entity s_1 which is mapped to LE le_1.

```
entity driver{ }
{   e_name s1; le_name le1;
        ...
    s1 = new server on le1;
}
```

Entities communicate with each other using buffered message-passing. An entity-type must declare the types of messages that may be *received* by it. Every entity has a unique message-buffer. A message is deposited in the message-buffer of an entity on the execution of an **invoke** statement. For instance, the following fragment will deposit a *request* message in the message-buffer of entity s_1.

```
invoke s1 with request;
```

Each message carries a timestamp, *tstamp*, which corresponds to the time (as measured by the clock of the sending entity) at which the corresponding **invoke** statement was executed. The message is deposited in the destination buffer at the time given by its *tstamp*. In logical modules, the message transmission times can be modeled explicitly by defining a separate *channel* entity. In a hybrid model, the physical time required to transmit a message may be relevant for an operational module.

If the keyword **invoke** in the send statement is replaced by **pinvoke**, the physical time required to transmit the message is measured (or estimated) and included in the message timestamp.

An entity accepts messages from its message-buffer by executing a **wait** statement. The **wait** statement has two components: an integer value called wait-time (t_c) and a Maisie statement called a resume-block – a (non-empty) sequence of resume-statements. The wait-statement has the following form:

wait t_c **until**
{ [declarations;]
r_1;
or r_2;
⋮
or r_n;}

Each r_i is a resume statement. A resume statement has the following form:

mtype(m_t) [**st** b_i] : *statement*;

where m_t is a message-type, b_i a boolean expression referred to as a *guard* and *statement* is any C or Maisie statement. The guard is a side-effect free boolean expression that may reference local variables or message parameters. If omitted, the guard is assumed to be the constant *true*. The message-type and guard are together referred to as a resume condition. A resume condition that includes a message-type m_t and a guard b_i is said to be *enabled* if the message-buffer contains a message of type m_t, which if delivered to the entity would cause b_i to evaluate to *true*; the corresponding message is called an *enabling* message. A resume condition that is not enabled is said to be *disabled*. If exactly one resume condition is enabled, and the message-buffer contains exactly one enabling message, the message is removed from the buffer and delivered to the entity which resumes its execution. If the buffer contains more than one enabling message, the message with the smallest timestamp is delivered to the entity. If two or more resume conditions are *enabled*, the timestamps on the corresponding *enabling* messages are compared and the message with the earliest timestamp is removed and delivered to the entity.

If all resume conditions in the wait statement are *disabled*, the entity is suspended for a *maximum* duration equal to its wait-time t_c; if omitted, t_c is set to an arbitrarily large value. A suspended entity resumes execution *prior* to expiration of t_c, if it receives an *enabling* message. If no enabling message is received in the interval t_c, the entity is sent a special message called a timeout message. An entity must accept a timeout message that is sent to it. Note that a non-blocking form of receive may be implemented by specifying t_c=0. By selecting r_cs appropriately, the **wait** statement may be used to ensure that an entity accepts a message from its input buffer only when it is ready to process the message.

The **wait** statement may also model a simulation step. For instance, the following statement executes a simulation step of duration 10 time units.

wait 10 **until**
mtype(timeout): *counter* = *counter* + 1;

Another construct called the **hold** statement is provided to express simulation steps. The **hold** statement is an abbreviated form of the **wait** statement and can only be used to program non-interruptible simulation steps. The following fragment rewrites the preceding simulation step with a hold statement. The timeout message is delivered implicitly.

hold(10);
counter = *counter* + 1;

The **wait** statement can also be used to model an interruptible action. For this purpose, any resume statement in a wait statement can be augmented with the **interrupt** phrase to indicate specific message(s) that may interrupt the resume statement. The general syntax for the modified resume statement is as follows:

mtype(m_i) [**st** b_i] : *statement*$_i$
    interrupt {
        mtype($m_{i,1}$) : $stat_{i,1}$;
        ...
    or **mtype**($m_{i,k}$) : $stat_{i,k}$; }

While executing an interruptible step (say *statement*$_i$), receipt of any of the specified interrupt message (say $m_{i,1}$) causes interruption of the corresponding simulation or computation step; the corresponding 'interrupt-handling' code is executed, after which the interrupted step resumes execution.

Implementation of the **interrupt** phrase in the simulation mode is straightforward: the original time-out message is simply rescheduled. The implementation is much harder for a computation step, because the latter is assumed to be atomic, and the PIPS run-time system cannot gain control until after the computation step has completed. As discussed in section 5.1, a restricted checkpointing and recomputation facility is required to implement interrupts in an operational module.

7 Examples

In this section, three different variations of the classical producer consumer problem are used to illustrate the use of MIDAS approach to the design of distributed systems and Hybrid Maisie to program hybrid models. This problem and the variations discussed here, provide a high level abstraction of many distributed systems. In the next section, we present measurements taken on the prototype MIDAS environment to design each of the three applications.

7.1 Bounded-Buffer Producer/Consumer Problem

We begin with a simple configuration with one producer and one consumer that communicate via a buffer of bounded size. The simulation model of this problem is shown in figure 9. The buffer process sends a *free* message to the producer, whenever it has an empty slot. The producer entity executes a **hold** statement to simulate data generation. The wait-time in the **hold** statement is sampled from a random exponential distribution.

After receiving a *free* message from the buffer, the producer entity generates a data item and sends a *put* message containing the data to the buffer. The consumer entity sends a *get* message to the buffer whenever it is idle and waits to receive the next data item in a *data* message. The driver entity is used to create a single instance of the the producer, consumer and buffer entities, where each entity is mapped to a unique LE.

To program this model into an operational program, it is sufficient to replace the **hold** statements executed in the producer and consumer entities by appropriate code that generates the required data item. The message-type *put* and *get* may also need to be modified to correspond to

the structure of the data communicated between the producer and the consumer. For instance, in figure 10 the consumer entity is elaborated into operational code by replacing the **hold** statement by a call to function *process_data* which is presumably used to process incoming data. Note that a hybrid model with a simulated producer and implemented consumer may be reexecuted to determine its performance characteristics.

7.2 Multiple Producers/Multiple Consumers Problem

One of the most important issues in system design is the *task allocation*, where a collection of processes are allocated to finite number of processors according to some resource and/or performance criteria. As we mentioned previously, the concept of Logical Element (LE) is introduced for easy experimentation with the allocation of software processes to hardware processors. Here we introduce a multiple producers/multiple consumers problem to demonstrate the applicability of the LE concept.

Two entities, fighter and bomber respectively, generate different messages, fighter_object and bomber_object, at an arrival rate determined by a negative exponential distribution. The messages represent the arrivals of fighter and bomber objects that need to be tracked by radar_fighter and radar_bomber entities, respectively. We first consider the configuration where both radar_fighter and radar_bomber entities are allocated to the same processor. The **new** statements in driver entity create instances of radar_fighter and radar_bomber entities, which are mapped to LE 3 (le3) as specified in the **at** phrases shown in figure 17 of appendix A. radar_fighter and radar_bomber entities, as mapped to the same LE, are executed sequentially according to the arriving order of fighter_object and bomber_object messages. To improve response time, we now allocate radar_fighter and radar_bomber entities to its own dedicated processor. All we have to do is to modify the arguments of the **at** phrases in the driver entity as shown in figure 18, where radar_fighter mapped to LE 4 (le4) and radar_bomber mapped to LE 3 (le3); the rest of the program remains unchanged. Since they are allocated to different LEs, the tracking computations for different objects may now execute in parallel.

7.3 Priority-Based Producer/Consumer Problem

To demonstrate the design of interruptible systems, we propose a priority-based producer/consumer problem, where an interruptible computation is specified by the **interrupt** construct together with its interrupt service computation. A producer process generates data items with two different priorities: high and low, and sends the data to the consumer for processing. The data item with the higher priority may preempt (interrupt) the processing of lower priority data item. Figure 11 presents the simulation code for the producer and consumer entities. In the simulation, negative exponential distributions with different means are used to generate simulation time for producing and consuming data items.

The producer entity executes a **hold** statement to simulate the generation of a data item and sends either a *HIGH* or a *LOW* message to the consumer entity according to some specified probability. In the consumer entity, the processing of a low priority data item is simulated by scheduling a *conditional simulation step*: the simulation step completes if the entity does not receive a high priority message in the interim; otherwise the simulation step is rescheduled after the higher priority item has been processed. Figure 12 shows a possible timing diagram for the interrupt processing, where data item *l0* is interrupted twice.

```
entity driver{ }
{   e_name p, c, b; le_name le1, le2, le3;
    p = new producer{pmean} at le1;
    b = new buffer{p} at le2;
    c = new consumer{b, cmean} at le3;
}

entity producer{pmean}
int pmean;
{   message free{e_name hisid; };
    while (true) do {
        wait until mtype(free);
        hold(exp(pmean));
        invoke msg.free.hisid with put; }
}

entity consumer{bufid, cmean}
e_name bufid; int cmean;
{   message data;
    while (true) do {
        invoke bufid with get{myid};
        wait until mtype(data);
        hold(exp(cmean)); }
}

entity buffer{pid}
e_name pid;
{   int count; message put; message get{e_name hisid; };
    for (count=0; count<BUFFER_SIZE; count++) invoke pid with free{myid};
    count= 0;
    while (true) do {
        wait until {
            mtype(get) st (count> 0) {
                count--; invoke msg.get.hisid with data;
                invoke pid with free{myid}; }
        or mtype(put) st (count< BUFFER_SIZE) count++;
        }
    }
}
```

Figure 9: Simulation Program for Bounded-Buffer

```
entity consumer{bufid, cmean}
e_name bufid; int cmean;
{   message data;
    while (true) do {
        invoke bufid with get{myid};
        wait until mtype(data);
        process_data(data); }
}
```

Figure 10: PIPS Program for Bounded-Buffer

```
entity producer{cons, pmeanh, prob_hi, xmit}
e_name cons; int pmeanh, xmit; float prob_hi;
{   int data;
    while (true) do
    {   hold(exp(pmeanh));    /* simulate data generation */
        if (random() ≤ prob_hi)
          invoke cons with HIGH{data};
        else
          invoke cons with LOW{data};
    }
}

entity consumer{cmeanh,cmeanl}
int cmeanh,cmeanl;
{   int hcnt = 0, lcnt = 0;
    message HIGH{int data};
    message LOW{int data};
    while (true) do
        wait until {
            mtype(HIGH) {
                hold(exp(cmeanh)); /* simulate consumption of high priority data*/
                hcnt++; }
        or  mtype(LOW) {
                hold(exp(cmeanl)); /* simulate consumption of low priority data*/
                lcnt++;
            } interrupt mtype(HIGH) { /* simulate interruption by high priority data*/
                hold(exp(cmeanh));
                hcnt++; }
        }
}
```

Figure 11: Priority-based Producer/Consumer Simulation Program

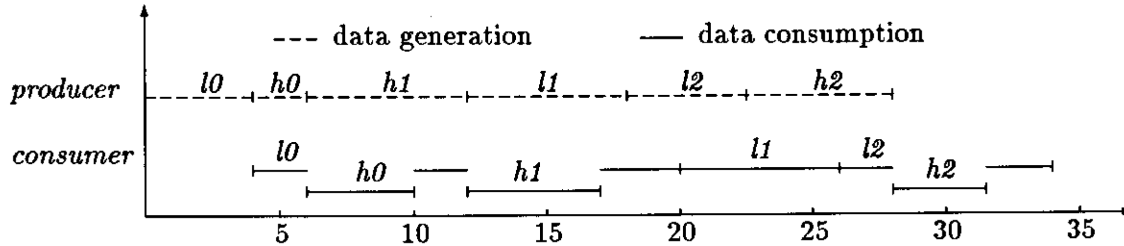

Figure 12: Time-Line for Interruptible Consumer Model

```
entity consumer_entity{cmeanh, cmeanl}
int cmeanh, cmeanl;
{
    int hcnt = 0, lcnt = 0;
    message HIGH{int data};
    message LOW{int data};
    while (true) do
        wait until {
             mtype(HIGH) { hcnt = hcnt + 1;
                           process_high(data); }
          or mtype(LOW) {lcnt = lcnt + 1;
                           process_low(data);
                         } interrupt
                             mtype(HIGH): {hcnt = hcnt + 1;
                               process_high(data) }
    }
}
```

Figure 13: Priority-based Producer/Consumer PIPS Program

We now consider the PIPS program for the preceding problem, where the consumer entity has been elaborated into operational code as shown in figure 13. In particular, note that the code for the consumer entity changes only to the extent that actual statements, functions *process_high* and *process_low*, to process the messages must be included in the entity.

8 Implementation and Results

A centralized environment to support the use of MIDAS in system design has been implemented at UCLA. The environment was used to design the three simple applications described in the previous section. MIDAS is also being used to develop a software subsystem of the Multiple Target Tracking Electron Beam Radar System in collaboration with Hughes Aircraft Co[11].

The MIDAS environment consists of three main components: the Hybrid Maisie compiler, a run-time system to execute hybrid programs, and a library of simulation entities and measurement facilities. The run-time system consists of the following major sub-systems:

- the PIPS algorithm

- an IPC kernel
- management of state-saving and recomputation to handle interrupts, and
- instrumentation routines to measure computation steps

The IPC kernel in a centralized implementation is straightforward and the PIPS algorithm has been described in section 4.2. The implementation of interruptible computation steps requires a restricted checkpointing and recomputation facility. (Note that an interruptible simulation step can be implemented simply by rescheduling the corresponding timeout message.)

Suppose that entity p_i executes a computation step c_i, that can be interrupted by the arrival of a message m_i. Assume that execution of c_i began at time t_b and was completed at time t_f. The state of p_i is saved prior to execution of c_i (that is at t_b). Suppose that an interrupt message m_i with timestamp t_i, $t_b < t_i < t_f$, is present. The run-time system restores entity p_i to its state at t_b, and schedules an operating system **alarm signal** which interrupts the computation c_i at t_i; the computation specified in the **interrupt** phrase is then executed as the interrupt-service code, after which execution of step c_i is resumed.

The fact that interrupts cannot cause propagating rollbacks ensures that a reasonably small amount of additional storage is required to implement interrupts in PIPS models. In addition, the overhead to detect if a rollback is needed is also small and restricted to simply checking for the presence of an interrupt message in the event-list prior to delivering a subsequent message to the entity following the execution of an interruptible computation step.

To measure physical time used by computation steps, the run-time system uses the **clock** routine provided with UNIX. The resolution of the clock prevents accurate measurements of the smaller time-intervals (including the physical message-transmission times). For such situations, we use an approximation deduced from the number of instructions executed and the average time to execute an instruction.

We first present the results of designing the simple bounded buffer application described in section 7.1. Performance metrics are measured at three different stages of model refinement. In the initial model, the producer, consumer and buffer entities all execute only simulation steps; subsequently the simulation steps of the consumer and producer entities were respectively elaborated to yield an operational system. The buffer entity from the simulation model was incorporated directly into the operational system. The time required to produce or consume a data item in the operational process was generated using a **for** loop. To validate the PIPS algorithm, the time consumed by the **for** loop was kept equal to the same random values of the exponential distribution used to generate processing time in the simulation model. For each data item, let t_0 be the time at which the producer entity begins to produce the data item, t_1 be the time the data item is inserted in the buffer, t_2 be the time it is removed by the consumer entity from the buffer and t_3 be the time at which the consumer entity finished processing the data item. At each stage of refinement, three performance metrics were measured:

- t_q: the average queueing time $= t_2 - t_1$
- t_s: the average system time $= t_3 - t_1$
- t_t: average total time $= t_3 - t_0$

The preceding metrics for each of the three different stages of system design are displayed in figures 14. The three stages respectively refer to the initial simulation model, a hybrid model with

an operational consumer and the eventual operational system. As seen from the figure, t_q, $t_s$and t_t are in reasonable agreement for each of the three stages.

A similar exercise was subsequently repeated for the priority-based producer/consumer example, as described in section 7.1, where *consumer utilization* is measured at the following three different stages of model refinement: in stage one, both producer and consumer entities execute only simulation steps; in the second stage, consumer entity executes an operational step to consume *LOW* messages, while a *HIGH* message is still processed by executing a simulation step. In the last stage, the processing of *HIGH* messages is also elaborated into an operational step. Once again, for easier validation, the time consumed by operational steps is sampled from the same exponential distribution used in the simulation model. The consumer utilizations measured at three different stages of system design are displayed in figures 15.

The multiple producers/multiple consumers problem was used to illustrate the ease with which different mappings of the software could be tested for their impact on the overall system performance. Two different configurations were executed: one with both consumer processes, radar_f and radar_b executing sequently, and the other where they were executed in parallel. Once again, the performance metrics were measured at three different stages of system design: the initial simulation model, a hybrid model with operational consumers and the eventual operational system. The measurements were used to determine the average *system time*, the sum of queueing time and service time. To simplify the experiment, both fighter and bomber entities use the same mean producing time and the same mean consuming time. Figure 16 shows the performance result.

9 Conclusion

We have described a methodology to integrate the performance evaluation of distributed systems with its design. The methodology may be used to determine that a system meet its performance requirements from its initial stage of inception until it is operational. In this section, we examine a few restrictions of our approach and briefly describe the implementation efforts in progress.

The central contribution of this methodology is in its use of a hybrid model of a distributed program, where each process may execute in either *simulation* or *computation* mode. A process may also execute partially in each mode. A process executes in the simulation mode to *model* the processing of a message, and executes in the computation mode to actually process the messages. In the computation mode, the physical time taken by the entity to process the message is measured by a physical processor clock. In the simulation mode, the actual time taken by the processor is ignored; instead the relevant time is the duration of the simulation step measured by the simulation clock. Both the simulation and computation time-periods must be included to predict system performance. As discussed in the previous section, the PIPS methodology can handle interruptible processes and also include the execution of hybrid models in which the physical module executes faster than the logical module.

The discussion in this paper has assumed a message-communicating model of a distributed system. We would like to emphasize that this does not exclude shared memory architectures from being used in this methodology. The methodology only assumes that the *programming model* is message-based, and may actually be implemented on a shared memory architecture. It is also possible to use a shared memory programming model by restricting the ways in which shared variables are updated. The effect of these restrictions is to ensure that every update to a shared variable is effectively treated as a synchronization point among the concerned processes. The restrictions are similar to those placed on Ada programs that use shared variables[17][section 9.11].

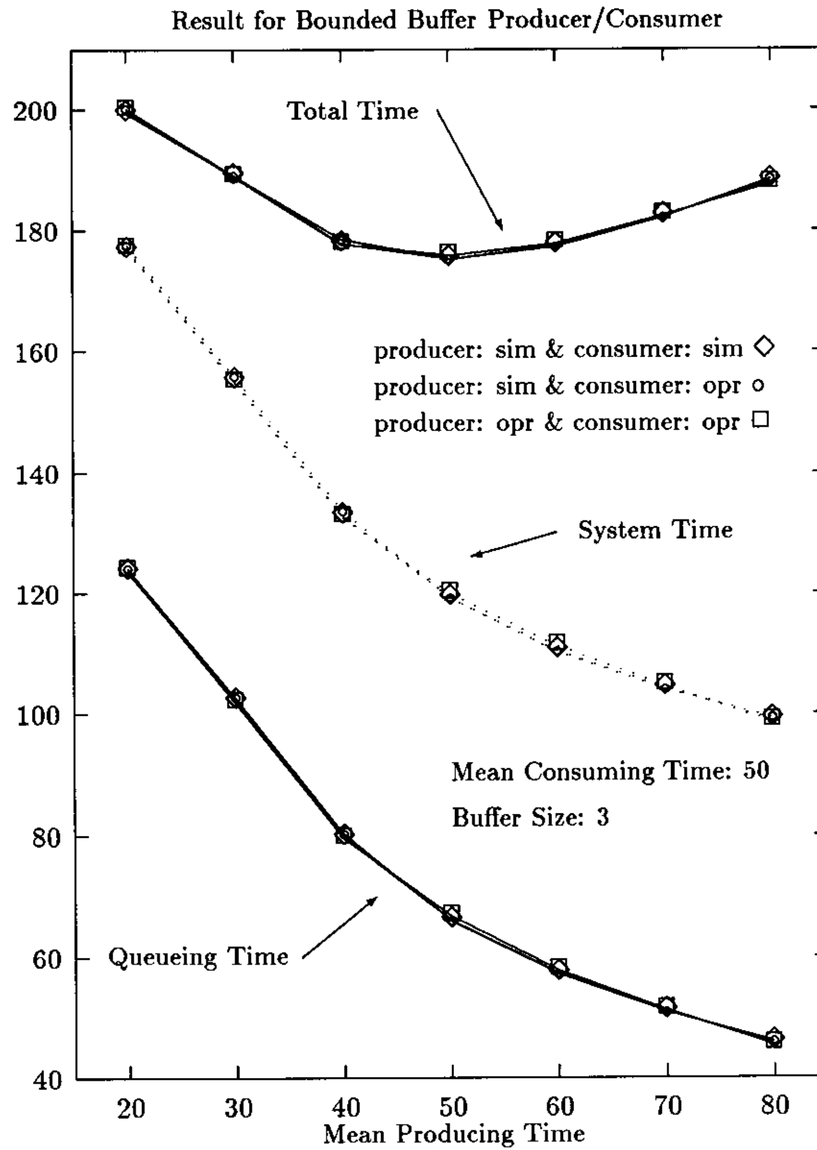

Figure 14: Performance Metrics for Bounded-Buffer Producer/Consumer

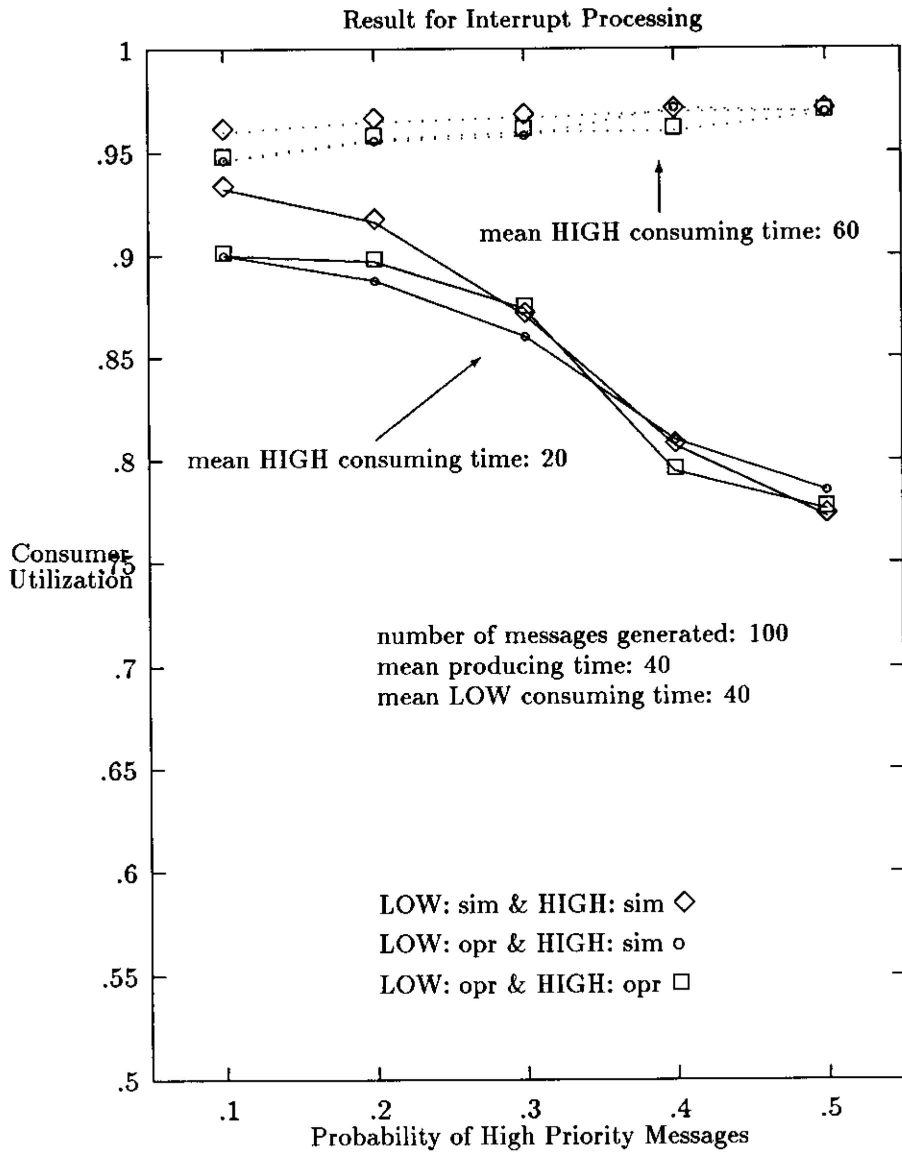

Figure 15: Performance Metrics for Priority-Based Producer/Consumer

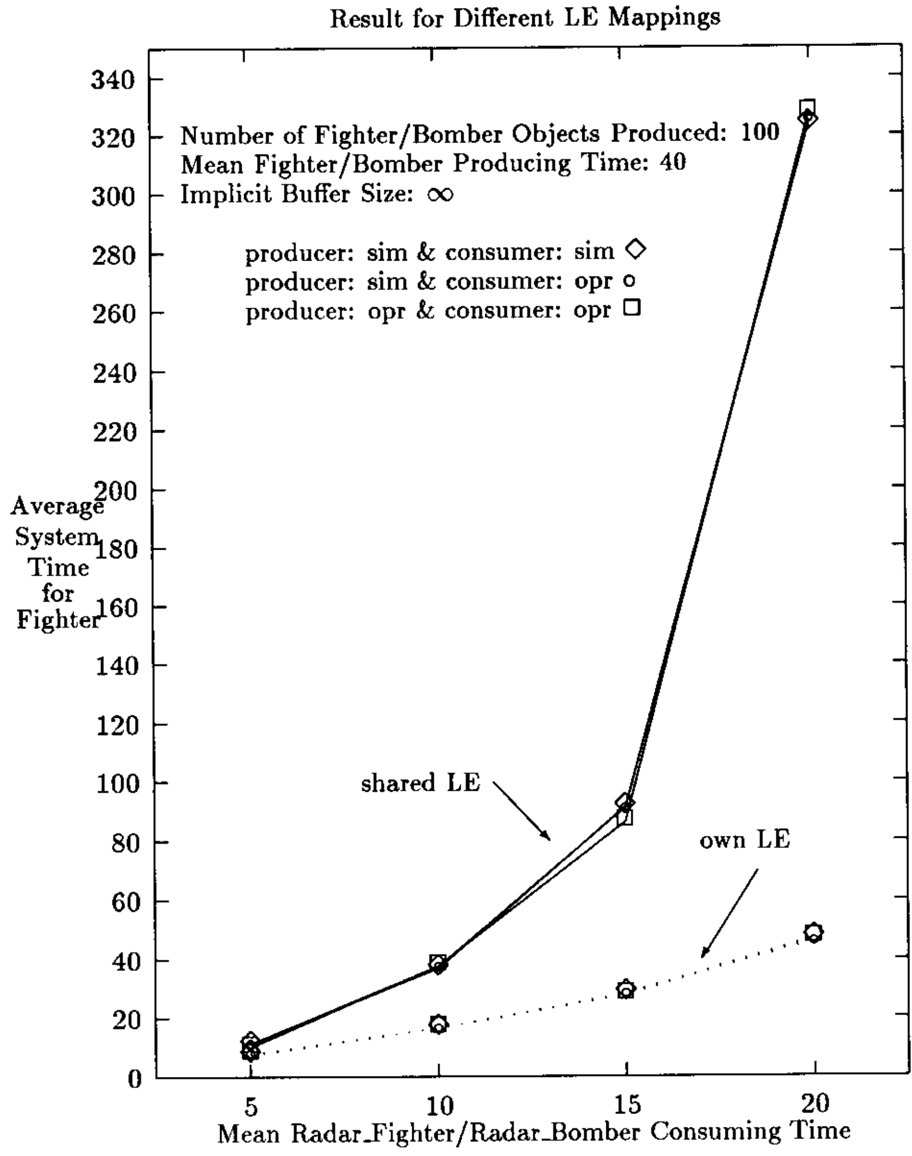

Figure 16: Performance Metrics for Multiple Producer/Multiple Consumer

Other restrictions have to do with the resolution of the processor clock on the simulation engine, which may limit the accuracy of the measurement of a computation step.

In order to use the integrated approach to performance prediction, it must be the case that the simulation hardware either be the same as, or be scalable to the hardware on which the proposed system will eventually be executed. As discussed in section 4.2 for centralized architectures, and section 5 for distributed architectures, the *virtual* clock of a process may possibly be advanced by a duration that is proportional, rather than equal, to the execution time of a computation step. This extension will allow analysts to directly examine the consequence of upgrading some existing hardware by a component that is, for instance, 50% faster. If the simulation hardware and the proposed hardware are radically different, measurements of the computation steps on the simulation hardware are not meaningful. In such situations, all processes in the PIPS program must be mapped to the *null element.* The integrated approach to system design is still useful; however the nature and purpose of the iterative refinements must be modified. As refinements are progressively introduced in the design, performance metrics must also be refined such that they relate only to the portion of the design that is as yet abstract[1].

We have not addressed the problem of workload characterization, or the related problem that arises when enhancements have to be made to an existing system: how do we incorporate the performance of the existing system into the performance prediction for future systems? This problem is relatively simple, because the approach presented in this papper includes operational modules in performance predictions. As such it is sufficient to build scaffolding around the separately designed, operational modules and treat it as a monolithic process which interacts with the rest of the system via messages.

A prototype implementation of the PIPS environment is under development. The system consists of a language called RTM and a run-time environment. RTM may be used to write real-time programs and their message-based simulation models. The run-time support environment provides facilities for inter-process communication and implements the PIPS algorithm to integrate operations in physical time with those in logical time. Remote communication facilities (communication between processes resident on different nodes) are provided by the Cosmic C environment[16]. Experience with the prototype in the design of real-time programs will be reported in a forthcoming paper.

Acknowledgements

The authors are grateful to Dick Muntz for many useful discussions and to Mani Chandy for discussions on the applicability of the space-time ideas to the execution of PIPS programs.

A Multiple Producers/Multiple Consumers Problem

```
entity driver{ }
{  e_name f, b, rf, rb
   le_name le1, le2, le3;
   rf = new radar_fighter{ rfmean }    at le3;  /* LE 3 */
   rb = new radar_bomber { rbmean }    at le3;  /* LE 3 */
   f  = new fighter      { rf, fmean } at le1;
   b  = new bomber       { rb, bmean } at le2;
}

entity fighter{ radar_f, rfmean }
e_name radar_f; int rfmean;
{  for ( ; ; ) {
        hold(expon(rfmean));
        invoke radra_f with fighter_object; }
}

entity bomber{ radar_b, rbmean }
e_name radar_b; int rbmean;
{  for ( ; ; ) {
        hold(expon(rbmean));
        invoke cb with bomber_object; }
}

entity radar_fighter{ fmean }
int fmean;
{  message fighter_object;
   for ( ; ; ) {
        wait until mtyp(fighter_object);
        hold(expon(fmean)); }
}

entity radar_bomber{ bmean }
int bmean;
{  message bomber_object;
   for ( ; ; ) {
        wait until mtyp(bomber_object);
        hold(expon(bmean)); }
}
```

Figure 17: Multiple Producers/Multiple Consumers Single LE Mapping

```
entity driver{ }
{  e_name f, b, rf, rb
   le_name le1, le2, le3, le4;
   rf = new radar_fighter{ rfmean }    at le4;  /* LE 4 */
   rb = new radar_bomber { rbmean }    at le3;  /* LE 3 */
   f  = new fighter      { rf, fmean } at le1;
   b  = new bomber       { rb, bmean } at le2;
}
```

Figure 18: Multiple Producers/Multiple Consumers Multiple LE Mapping

References

[1] R. Bagrodia. *An Environment For the Design and Performance Analysis of Distributed Systems.* PhD thesis, Dept. of Computer Sciences, University of Texas, Austin, Tx 78712., May 1987.

[2] R. Bagrodia and W.-T. Liao. *Maisie User Manual.* Computer Science Dept., UCLA, 1990.

[3] R.L. Bagrodia and W.-T. Liao. Maisie: A language and optimizing environment for distributed simulation. In *1990 Simulation Multiconference: Distributed Simulation*, San Diego, California, January 1990.

[4] K.M. Chandy, J. Misra, R. Berry, and D. Neuse. The use of performance models in systematic design. In *AFIPS, Procedings of the National Computer Conference*, volume 52, 1982.

[5] K.M. Chandy and R. Sherman. Space-time and simulation. In *Distributed Simulation Conference*, Miami, 1989.

[6] G. Estrin, R. Fenchel, R. Razouk, and M. Vernon. Sara (system architects apprentice): Modeling, analysis and simulation support for design of concurrent systems. *IEEE Transactions on Software Engineering*, SE-12(2):293–311, February 1986.

[7] W.R. Franta. *The Process View Of Simulation.* Elsevier North-Hollabd Inc., New York, 1977.

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

[9] D. Jefferson and H. Sowizral. Fast concurrent simulation using the time-warp mechanism. In *Distributed Simulation 1985, Society for Computer Simulation Multiconference*, San Diego, 1985.

[10] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1), March 1986.

[11] D. Noal. The design of the multiple target tracking electron beam radar system using midas approach. Master's thesis, Computer Science Dept., UCLA, 1990.

[12] D. Parnas. Sodas and a methodology for system design. In *AFIPS Conference Proceedings*, 1967.

[13] W.E. Riddle. The modeling and analysis of supervisory systems. Ph.D. thesis, Stanford University, March 1972.

[14] G.C. Roman. Specifying software/hardware interactions in distributed systems. In *Proceedings of the International Conference on Software Engineering*, pages 126–139, May 1987.

[15] J. Sanguinetti. A technique for integrating simulation and system design. In *Conference on Simulation, Measurement and Modelling of Computer Systems*, Boulder, Colorado, August 1979.

[16] C.L. Seitz. The cosmic cube. *CACM*, 28(1):22–33, January 1985.

[17] United States Department Of Defense. *Reference Manual for the Ada Programming Language*, 1983.

[18] F. Zurcher and B. Randell. Iterative, multi-level modelling - a methodology for computer system design. In *Proceedings IFIP Congress 68*, 1968.