The Hierarchical Simulation Language HSL: A Versatile Tool for Process-Oriented Simulation

D. P. SANDERSON, R. SHARMA, R. ROZIN, and S. TREU
University of Pittsburgh

The Hierarchical Simulation Language (HSL) was designed and developed to serve process-oriented simulation of discrete systems. It is interpreter-based and hence offers certain advantages, such as portability (hardware independence) and modifiability (during program execution). An HSL model consists of two major sections. The **Environment** contains the specifications of the model and model control statements. The **Simulator** is a set of functions and processes that carry out the run-time activities of the model. Processes can be hierarchically refined or compressed, to whatever level of model detail desired. This paper describes the design characteristics and programming constructs of HSL. Several issues relevant to simulation languages in general and HSL in particular are then discussed. This is followed by an example HSL program presented to illustrate many of its features.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features; I.6.2 [**Simulation and Modeling**]: Simulation Languages; I.6.8 [**Simulation and Modeling**]: Types of Simulation–*discrete event*

General Terms: Languages

Additional Key Words and Phrases: C + +, hierarchy, HSL, inheritance, interpreter, modularity, process, simulation programming language

1. INTRODUCTION

A new simulation programming language, called Hierarchical Simulation Language (HSL), has been designed, implemented and tested. In appearance, it resembles structured languages like Pascal. In simulation-specific capabilities, it belongs to the category of procedural simulation languages, such as SIMULA [3]. HSL views simulation as attributed entities flowing concurrently through the simulated system each according to its process script, consuming resources along the way. Statistics relevant to entities, processes and resources are automatically collected and reported.

In view of the myriad of simulation programming languages already in existence (e.g., see Banks and Carson [1] Fishman [6] and Unger [39]), it is reasonable for the reader to ask: why yet another language? Analysis of

This research was supported in part by NCR Corporation, Cambridge, Ohio.
Authors' address: Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260.

traditional simulation languages leads to the observation that many of them favor either the modeler (who is not also an expert programmer) at the expense of the expert programmer, or vice versa. For instance, a language that accommodates the modeler's conceptual framework may not be sufficiently expressive for the programmer or support principles of software engineering for modular development. One approach to overcoming these problems is to define a model specification language as an intermediate form between the conceptual model and its programmed implementation [18]. Another approach is to design a programming language that attempts to meet the needs of both the modeler and the programmer.

HSL, which is currently in a prototype implementation state, takes the latter approach. Its design is based on principles of programming language design (e.g., see MacLennan [17] and Pratt [20]) and process oriented simulation [8]. It takes advantage of modern-day software engineering techniques such as structured programming and abstraction. With its process orientation and hierarchy [42], various software engineering principles (e.g. see Golden [10] and Sheppard [34]) can be utilized to support top-down, detailed model construction. On the other hand, its conceptual and structural properties enable modeler-conducive correspondence [28] between a conceptual model and the software model and its manipulation. The object-oriented programming perspective, which we and others (e.g. see Bezivin [2] and Rothenberg [25]) feel is appropriate for discrete system simulation, is supported in HSL by user-defined entity classes. The general HSL language design philosophy can be summarized as providing high-level simulation constructs and full programming power, while keeping the language small but extensible.

In addition, HSL is machine-independent. Its prototype implementation is interpreter-based. The host language for the interpreter was chosen to be C++ [36]. In the interpretive mode of execution, an HSL program is first translated into intermediate form. That intermediate program is used as input to an executor, which executes one intermediate instruction at a time by calling a function to perform the specified action. HSL was implemented through an interpreter rather than a compiler, despite the faster execution speed of the latter. Advantages of doing so include (1) the above-mentioned portability, (2) the ability to modify model parameters during program execution, and (3) easier development and changes of the language itself, especially during its formative stages. Use of the *Lex* [16] and *Yacc* [12] grammar definition tools permitted total freedom to experiment with the language design. The unique syntax was chosen to be concise, in keeping with the language design philosophy.

Documentation of the details of HSL syntax [23] and semantics [24] as well as a user's guide [27] are available in technical report form. In addition, HSL interpreter-based implementation techniques [26] and an HSL overview paper have been published [28]. But, this is our first attempt to provide detailed language statements and examples of interest to a wider audience of simulation programmers and specialists.

This paper first presents an introduction to HSL, in terms of its high-level modularity, language constructs and features. Several specific issues

relevant to HSL language design are then discussed. This is followed by an illustration of HSL use in a simulation program given by means of a machine shop model. A discussion of HSL implementation status and related research follows the example.

2. MODEL DICHOTOMY

A model written in HSL is divided into two modules. One of them encompasses the global environment in which the model exists and the other provides a functional view of the simulated actions undertaken by the entities within the model. The two modules are described below.

2.1 HSL Environment

The **Environment** of an HSL model represents those characteristics of the modeled system, or its configuration, that remain static over the duration of a simulation. Although the Environment normally remains the same during a particular run, it is changed between runs in order to exercise the model under different conditions. For this reason, the Environment of an HSL program is a module separate from the **Simulator**. The HSL Environment is similar to Zeigler's "experimental frame." [41].

A programming language is described in terms of what its programs look like (the language *syntax*) and what its programs mean (the language *semantics*). Syntax is much simpler to describe formally than semantics, and is commonly done so using context-free grammars expressed using Backus-Naur Form (BNF) or an extension of it. For HSL syntax specification, we have chosen an extended BNF (EBNF) similar to that adopted for specifying the Ada language [9]. For the interested reader, an EBNF portrait of the HSL Environment, along with a description of the notation, is listed in Appendix A. It remains difficult, however, to visualize a language through its EBNF specification, so we introduce HSL Environment syntax and semantics informally through an annotated example. The sole purpose of this model is to illustrate a wide range of language features.

The program module shown in Figure 1 exhibits most features of the HSL Environment. The statements are presented and explained in logical groups, with each code segment preceding its explanation. HSL reserved words, functions and variables are displayed in boldface type. Line numbers are included for ease of reference; they are not part of the language.

```
 1:  model Demonstrate
     ...
26:  end Demonstrate;
```

The **model** statement, which is completed by its corresponding **end** clause, encloses and identifies the HSL Environment. Dashes indicate omitted statements.

```
 2:  constant real  InterArrive:= 3.4,
 3:                 ServeMean:= 7.0,
 4:                 ServSD:= 2.7;
 5:  int Arrivals[0..4]:= 1000;
 6:  bool StopFlag, TraceFlag;
```

```
                ! Illustrative HSL Environment Module.

1:      model Demonstrate

                ! Base declarations.

2:              constant real  InterArrive := 3.4 ,
3:                             ServeMean := 7.0 ,
4:                             ServeSD := 2.7 ;
5:              int Arrivals[0..4] := 1000 ;
6:              bool StopFlag, TraceFlag ;

                ! Component declarations.

7:              stat           Duration : ind ;
8:              queue          HoldA, HoldB : fifo ;
9:              resource       Guard : fcfs ;
10:             resource [2] Server[11..20] : fcfs ;

                ! Entity class declarations.

11:             entity Job :
12:                     real EntryTime ;
13:                     bool Success ;
14:             end Job ;

15:             Job entity CompileJob :
16:                     int LinesOfCode ;
17:             end CompileJob ;

                ! Model control statements.

18:             start GenJobs ( 100 ) ;
19:             stop clock > 120.0 or StopFlag ;
20:             trace TraceFlag ;
21:             report rptlapse >= 10.0 ;
22:             report Duration.curr > 22.0
23:                     entity   | true ;
24:                     stat     | HoldA.len.curr > 5.0 ;
25:             end report ;

26:     end Demonstrate ;
```

Fig. 1 Illustrative HSL environment

Four base data types are available for declaring global variables: integer, real, boolean, and string. A declaration can be extended in two ways: by specifying an initial value and by declaring an array of variables. All declarations containing the assignment operator (:=) specify an initial value, which overrides HSL's default initialization of variables. The variable

Arrivals is declared as an array of five integers accessed through subscript values 0 through 4. Only one initial value can be specified for an array; each element of the array receives the specified value. Global constants are declared by prepending the reserved word **constant** to the entire declaration. A constant is distinguished from a variable in that its value may not be changed in any of the Simulator code.

```
 7:  stat            Duration : ind;
 8:  queue           HoldA, HoldB : fifo;
 9:  resource        Guard : fcfs;
10:  resource [2]    Server [11..20] : fcfs;
```

Statistics, queues, and resources are abstract data types which are significant to discrete system simulation. Each has a full complement of data and functional attributes to implement appropriate semantic actions. They define simulation objects which exist for the duration of model execution and are globally accessible. The words specified after the colon (:) represent operation modes.

In this segment, *Duration* is declared as a time-independent statistic for which values are explicitly collected in the Simulator module. Time-dependent statistics are also available. *HoldA* and *HoldB* are declared as two queues for holding entities. Each operates in a first-in-first-out manner. Other available modes are lifo, priority order, and random order. Every queue contains a set of automatically maintained statistics. *Guard* is declared as a resource of capacity 1 which is allocated in a first-come-first-served manner. Other modes are first-fit, and preemptive allocation. Every resource contains a predefined queue, which is accessible but maintained automatically, along with statistics. *Server* is declared as an array of 10 resources (accessed by subscript values 11 through 20), each of which has capacity 2. Capacities may also be specified for queues, as well as arrays of queues and statistics.

```
11:  entity Job:
12:          real EntryTime;
13:          bool Success;
14:  end Job;
15:  Job entity CompileJob;
16:          int LinesOfCode;
17:  end CompileJob;
```

HSL provides a data-typing capability using the **entity** class. It enables the modeler to define an abstract data type and its attributes (component parts). Objects of the entity class may then be dynamically created and destroyed within the Simulator. Each entity object represents a transaction flowing through the model, such as a job through a computer system. The description of an entity's activities in the system is called a process, which forms the basis for model execution.

The declaration of an entity class consists of its name and a list of declarations of its attributes. This is similar to the *record* construct of Pascal. An attribute may be of any of the previously mentioned types, but, in the

current version, may not be of an entity class. Only the data attributes of an entity class are declared here. Its functional attributes are endogenous processes linked to the class through the **accepts** clause of the process definition, as shown in Section 2.2. The HSL Environment contains only the definition of entity classes; instances of a class, called entities, are created during execution in the HSL Simulator.

Refinement or specialization of a class of entities is facilitated through the use of *inheritance*. Inheritance allows a class to extend its attributes automatically with those of another class, called its parent class. Tree-shaped hierarchies of entity classes may be formed by specifying in a class definition the name of its parent entity class. An entity declared in a process to be of a child class contains all the attributes declared for that class plus all those declared for all its ancestor classes. All entities inherit the system-defined attributes **id**, which is assigned an identification number upon entity creation, and **priority**, which is assigned the value 0 (lowest priority) upon entity creation (and can be changed using **setpriority(n)**). In this example, *Job* is a parent class and *CompileJob* is its child class, as specified by prepending the parent class name to the **entity** reserved word.

```
18:  start GenJobs (100);
19:  stop clock > 120.0 or StopFlag;
20:  trace TraceFlag;
21:  report rptlapse >= 10.0;
22:  report Duration.curr > 22.0
23:          entity   | true;
24:          stat     | HoldA.len.curr > 5.0;
25:  end report;
```

Statements to control the execution of the simulation program, from within the Environment module, provide the starting, stopping, tracing and reporting conditions. No restrictions are placed on the order of these statements except that they must be preceded by all of the global declarations of the simulation model. Conditions are specified as boolean expressions and are evaluated with each simulation clock update.

The **start** statement is used to specify which processes of the model begin execution at the start of the simulation run. These are exogenous processes, since their execution is controlled by the Environment and not from within the Simulator. The **stop** statement specifies the conditions under which the simulation run terminates. The HSL reserved identifier **clock** contains the current value of the simulation clock. Default time units and starting time are seconds and 0.0, respectively. The **trace** statement controls when a trace of simulation activity is to be produced.

The **report** statement establishes the conditions under which a simulation report is to be generated. A simulation report by default contains the statistical values for all user-defined statistics, queues, resources, entity classes and processes in the model. All statistics in the report are maintained automatically, except user-defined statistic variables. In this example, reports are generated under two sets of conditions. Line 21 specifies that a full report is to be produced when the elapsed simulation time since the last report (stored in HSL variable **rptlapse**) is at least 10 seconds. If additional

conditions are specified, as they are in the second report statement (lines 22–25), then information about the indicated system component type is included only if its expression evaluates to **true** at the time of report generation. If a **stat** condition is specified and evaluates to **true**, all statistical values in the model are reset.

Much of the power of the model control statements is derived from the modeler's ability to use predefined attributes of statistics, queues resources and processes in specifying conditions. Each variable declared to be of type statistic, for example, has attributes for its mean, standard deviation, number of observations, and minimum, maximum and current value.

2.2 HSL Simulator

The second major module of an HSL program is the **Simulator**. It represents the functional description of a modeled system. The Simulator consists of an unordered collection of process and function definitions, all of which are considered global. The HSL Simulator is conceptually similar to Zeigler's "lumped model" [41] or model frame. Processes simulate the actions undertaken by the entities within the model. A process describes the activities of a model entity through a sequence of HSL statements. This activity sequence may be developed "hierarchically", by introducing **calls** to other processes, much as subroutines or procedures are used in general-purpose languages. Alternatively, it may be developed "laterally" using **schedule** statements, involving permanent transfer of entities to the scheduled processes without any return. This distinction is clarified in the following presentation and in Section 3.1.

Presentation proceeds as with the Environment. The full EBNF grammar of the HSL Simulator is listed in Appendix B, while the flavor of the language is presented through an annotated example. The program module shown in Figure 2 exhibits most features of the HSL Simulator.

The statement sequence within a process or function includes local declarations and programming statements. A process, in addition, contains simulation control instructions. There are no unique symbols for grouping a statement sequence (such as { }); constructs supporting statement sequences enclose them using an **end** statement (e.g. **loop...end loop**). The simple programming instructions include assignment (:=), **read** and **write** statements. The expected arithmetic, logical and relational operations are available. Type conversion, where legal and necessary, is performed implicitly. Specifically, integer and real expressions are compatible, with real being converted to integer through truncation.

```
 1:   process GenJobs (int NumArrival : in)
      ...
14:   end GenJobs;
15:   process Work ( ) accepts CompileJob Cjob
      ...
32:   end Work;
33:   function CompTime (int Amt) returns real
      ...
37:   end CompTime;
```

```
                ! Illustrative HSL Simulator Module.


                ! Exogenous process to generate job entities.

1:      process GenJobs ( int NumArrival : in )
2:              int Count := 0 ;
3:              CompileJob Comp ;
4:              loop
5:                      delay expo ( 1, InterArrive ) ;
6:                      Comp.create ;
7:                      schedule Work ( ) with Comp ;
8:                      if Count < NumArrival then
9:                              Count := Count + 1 ;
10:                     else
11:                             break ;
12:                     end if ;
13:             end loop ;
14:     end GenJobs ;

                ! Process to carry out the work of a CompileJob entity.

15:     process Work ( ) accepts CompileJob Cjob
16:             int WorkAmt, Class ;
17:             Cjob.EntryTime := clock ;
18:             WorkAmt := Cjob.LinesOfCode / 200 ;
19:             case WorkAmt of
20:                     0..5    | Class := 11 ;
21:                     6..8    | Class := 12 ;
22:                     9..12   | Class := 13 ;
23:                     default  Class := 20 ;
24:             end case ;
25:             Server [ Class ] . request ( 1 ) ;
26:                     delay CompileTime ( WorkAmt ) ;
27:             Server [ Class ] . release ( 1 ) ;
28:             call MoreWork ( ) with Cjob ;
29:             write "Job", Cjob.id, "entered at", Cjob.EntryTime,
30:                     "completed at", clock ;
31:             Cjob.destroy ;
32:     end Work ;

                ! Function to calculate required resource time.

33:     function CompTime ( int Amt ) returns real
34:             real result ;
35:             result := Amt * 2.56 + 5.2 ;
36:             return result ;
37:     end CompTime ;
```

Fig 2. Illustrative HSL simulator.

A **process** definition contains the process name, the formal parameter list, an optional entity parameter and a sequence of statements. If the optional entity parameter is specified, the process becomes an attribute of the entity class and activation of the process applies to an instantiated entity. Otherwise, the process is an exogenous process and can only be initiated by the Environment's **start** statement. Thus, process *GenJobs* (lines 1–14) is exogenous, and *Work* (lines 15–32) is associated with entity class *CompileJob*.

The formal parameter list indicates the type and mode of each of the arguments that are passed to the process. Parameter passing modes may be specified as **in**, **out**, or **inout**. **In** means the parameter cannot be modified in the process body, **out** means the parameter can only be modified and **inout** means the parameter can be referenced freely within the process.

A **function** definition contains the function name, the formal parameter list, the function's type and a statement sequence. The execution of a function is terminated with a **return** statement containing an appropriately typed value that indicates the function's result (see line 36 in Figure 2). Parameter-passing modes are not specified, as parameters must be of mode **in**. Functions are provided for computational ease only and may not contain HSL simulation control statements. The actual parameters used in the invocation of a function or process must match the formal parameters in both type and number.

```
 2:  int Count:= 0;
 3:  CompileJob Comp;
     ...
16:  int WorkAmt, Class;
     ...
34:  real result;
```

Variables and constants declared within a process or function are considered local to that process or function. Local constants must be of one of the simple types and local variables must be either of one of the simple types or of an entity class (line 3). Arrays may be declared and all local variables except entity variables may be initialized. It is important to note that declaration of an entity does not create an entity, it merely allows the modeler to refer to a subsequently created entity using the variable name.

```
 1:  process GenJobs (int NumArrival : in)
             ...
 3:          CompileJob Comp;
             ...
 6:                  Comp.create;
 7:                  schedule Work ( ) with Comp;
             ...
14:  end GenJobs;
15:  process Work ( ) accepts CompileJob Cjob
             ...
28:          call MoreWork ( ) with Cjob;
             ...
31:          Cjob.destroy;
32:  end Work;
```

The essence of entity activity is captured in this collection of statements. The *GenJobs* generating process declares a variable (line 3) to represent an entity. An entity is subsequently created and bound to that name (line 6), then scheduled to execute the *Work* process (line 7). Since no future time is specified, *Work* commences execution at the current simulation time. Once the schedule is issued, *GenJobs* continues execution but may no longer access that entity. The scheduled process *Work* then binds the entity to the variable declared to accept it (line 15) and commences execution independently of and in parallel to *GenJobs*.

When the subprocess *MoreWork* (not shown) is called (line 28), execution of *Work* is suspended until its return. In general, either a single process or multiple processes (which work in parallel) may be invoked in a **call** statement. The calling process is suspended until the called process(es) has(have) completed, after which the suspended process continues its execution. It is through this call mechanism that processes may be developed in a hierarchical manner.

Finally, the entity has completed all activity and its memory space is reclaimed (line 31). This is necessary because HSL does not provide a garbage collection facility for automatically reclaiming storage.

```
25:    Server [Class].request (1);
26:          delay CompileTime (WorkAmt);
27:    Server [Class].release (1);
```

Some HSL statements affect the simulation time and hence are specifically important for simulation programming. These statements can only be used in process definitions, not in functions. The **delay** statement is used to suspend a process for a specified period of simulation time. Line 26 illustrates the use of a function invocation to calculate the delay time. Explicit and indefinite process suspension is controlled through the **suspend** and **awaken** operations (not shown).

The **request** and **release** operations, shown in lines 25 and 27, are HSL resource attributes which enable the current entity (the one accepted by the enclosing process) to use system resources in the amount indicated by the parameter. If, at the time of the request, the necessary resource is not available, the process is immediately suspended, and the entity enqueued for the resource. When the request is granted (upon release by another entity), process execution is resumed at the statement following the request. When use of the resource is complete, it is released so that others may gain access. No simulation time passes during resource release.

```
 4:    loop
             ...
 8:          if Count < NumArrival then
 9:                Count:= Count + 1;
10:          else
11:                break;
12:          end if;
13:    end loop;
       ...
19:    case WorkAmt of
20:          0..5      | Class:= 11;
```

```
21:          6..8     | Class:= 12;
22:          9..12    | Class:= 13;
23:          default Class:= 20;
24:    end case;
```

For general-purpose program control, complex statements can be constructed for selection and iteration of a statement sequence. Selection is done with two constructs, the **if** and the **case** statements. The **if** construct is of the *if—then—else—end if* variety, with the **else** clause being optional. The **case** statement provides a selection mechanism for a situation in which alternative actions are available. A **default** clause may optionally be specified. If it is not specified and none of the other choices applies, no action is taken.

Only one iteration mechanism, namely the **loop** construct, is available. This limitation was imposed purposely, to keep the HSL prototype reasonably simple. Loop execution ends when a **break** statement is executed; a loop iteration ends and the next one begins either when the corresponding **end loop** is reached or a **next** statement (not shown) is executed.

```
17:    Cjob.EntryTime:= clock;
18:    WorkAmt:= Cjob.LinesOfCode/200;
       ...
25:    Server [Class].request (1);
       ...
27:    Server [Class].release (1);
       ...
29:    write "Job", Cjob.id, "entered at", Cjob.EntryTime,
30:                "completed at",clock;
31:    Cjob.destroy;
```

Finally, in the Simulator, the modeler can use the predefined attributes of statistics, queues, resources, entities, entity classes and processes along with modeler-defined attributes of entities. Predefined attributes are equivalent to a constant or the result of a function. In any case, they may not be altered by the modeler. Attributes are specified using dot notation, as are Pascal record components. The **request** and **release** operations (lines 25 and 27), explained above, are implemented as attributes of model resources. Modeler-defined entity attributes may be either variables or constants and used accordingly. Line 29 shows an example of both HSL-defined (**id**) and modeler-defined (*EntryTime*) entity attributes.

3. SELECTED LANGUAGE ISSUES

In order to more fully understand the HSL language, selected issues are discussed in greater detail. These are HSL support capabilities for (1) process-oriented simulation, (2) probability and statistics and (3) object-oriented programming and a comparison of HSL with several well-known simulation languages.

3.1 HSL Processes

A brief overview of the process view of simulation [8] facilitates understanding of HSL processes. A process describes the activity sequence undertaken by a model entity. A process is defined once, in the form of a programmed procedure. However, at any given time during the simulation, many copies,

or *activations*, of this process (as well as those of other processes) may be in use by different entities. Conceptually, these process instances are executing in parallel, but in reality they may be executing serially on a single processor system. In this case, they are said to be executing *concurrently*. Programming languages that support concurrency (and there are not many) do so through *coroutines*, which are distinguished from *subroutines* by their role as equal partners as opposed to servants. In general, a process is a coroutine.

In this high-level discussion of concurrency presented, the term "process" refers to a process instance. The simulation executor, an underlying simulation control program, allows a process to run until it must suspend awaiting external action. Suspension can occur under several conditions. A process which delays for a specified period of time cannot continue until the simulation clock is advanced. A process which is blocked awaiting access to a modeled resource cannot continue until the resource becomes available. A process may also choose to deactivate itself, relying on another process to reactivate it. When a process suspends, its execution state is retained and the simulation executor selects a previously suspended process to resume execution from its point of suspension. The process selected must be from the set of suspended processes whose conditions for resumption have been satisfied.

The basis for process execution in HSL is the entity-process pair. After an HSL entity (an instance of an entity class) is dynamically created, it cannot pursue its activities until bound to a process activation. When a **call** or **schedule** of a process is executed, the process activation is created and bound to the entity. A called process then proceeds immediately; a scheduled process proceeds when conditions allow. When a suspended process (process-entity pair) is scheduled for resumption at a known time, it is placed on a time- and priority-ordered process-resumption ('future-event') list maintained by the simulation executor. Priorities are used to determine order in the case of simultaneous resumptions. Since a process itself has no priority attribute, the executor uses the system-defined priority attribute of its associated entity. All entities are created with equal priority, but higher priorities may be assigned using the entity attribute **setpriority(n)**.

An abstract view of process invocation is shown in Figure 3. Normal flow through a process can be visualized as shown in Figure 3a. An invoked process can operate either synchronously or asynchronously with its invoking process. In the synchronous case, the invoker suspends until the invoked procedure terminates and returns control to it. This represents hierarchical refinement, as through a subroutine, and can be visualized as the vertical control flow shown in Figure 3b. In asynchronous operation, the invoker continues execution while the invoked procedure proceeds independently and in parallel. This represents a lateral model refinement which can be visualized as the horizontal control flow shown in Figure 3c.

Hierarchical refinement is realized in HSL through the **call** statement and lateral refinement through the **schedule** statement. These statements may be used and combined in various ways to support different conceptual views of process flow. Two extreme abstract views are shown in Figure 4. The process flow embodied in Figure 4a is localized in that each operation in a

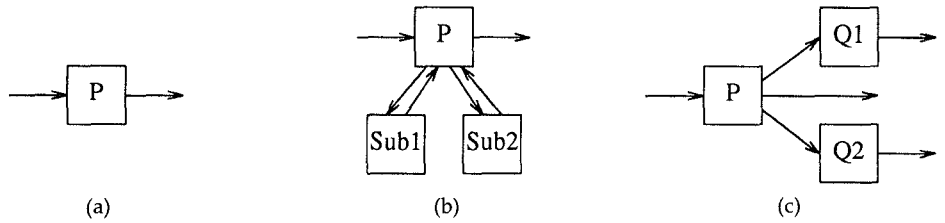

Fig. 3. (a) Normal process flow, (b) hierarchical refinement (vertical orientation); (c) lateral refinement (horizontal orientation).

```
process OperA
    ...
    ...
    schedule OperB
end OperA

process OperB
    ...
    ...
    schedule OperC
end OperB

process OperC
    ...
    ...
end OperC
```

(a)

```
process Overall
    call OperA
    call OperB
    call OperC
end Overall

process OperA
    ...
    ...
end OperA

process OperB
    ...
    ...
end OperB

process OperC
    ...
    ...
end OperC
```

(b)

Fig. 4. Two abstract HSL process flow views.

sequence is encapsulated in a process, which, when completed, schedules its associated entity for the next operation. There is no encapsulated representation of the operation sequence. In this sense, the sequence of operations is "chained" together. An alternative view, summarized abstractly in Figure 4b, represents an operation sequence hierarchically with one process at the top level and sequence implied by the order of its subprocess calls. These views may also be combined to model a system having both sequential and hierarchical aspects, as is seen in the example HSL model presented in Section 4. In general, the method utilized should be that which most closely resembles the system being modeled.

3.2 Probability and Statistics

Systems to be simulated generally contain elements of uncertainty and evolve through time in an unpredictable manner. These are referred to as stochastic systems. The modeling of stochastic systems requires the

application of probability concepts to characterize this variability. The application of statistical methods is required to interpret system variability. HSL supports stochastic modeling through random number generators, probability distributions and both system- and user-defined statistics.

3.2.1 *Probability Functions.* The building block for sampling random variates is the **rand01** function which returns a pseudo-random value selected uniformly from the range 0–1. This function uses a congruence, or residue, method [11] for generating random numbers, which means that the value of any random sample is a function of the previous sample. An initial value, or seed, is required to produce the first sample. This method supports the notion of random number *streams*, sequences of random numbers which can be replicated at will given a known seed value. HSL provides 16 streams with distinct system-supplied seeds accessible by supplying the value 0–15 as the first parameter of **rand01** or any of the probability distribution functions. The **setseed** function not only allows the modeler to supply specific seed values, it also extends the effective number of streams infinitely because when invoked it returns the old seed value. The old seed can be saved in a variable, then restored by using that variable as the new seed parameter in a subsequent invocation of **setseed**.

HSL provides a number of useful probability distribution functions, each of which utilizes **rand01** internally. **Randint** returns an integer uniformly selected from the specified range. **Uniform** returns a real number uniformly selected from the specified range. **Expo** returns a value selected from the exponential distribution having the specified mean. **Normal**, **hyperex** and **erlang** return a value selected from the normal, hyperexponential and erlang distributions, respectively, having the specified mean and standard deviation. The modeler may define additional probability distribution functions as HSL functions which utilize **rand01** as a source of random values.

3.2.2 *Statistics and Reports.* The HSL approach to statistical analysis parallels its approach to other aspects of the language; it provides basic constructs to facilitate initial modeling, plus extensibility to enable specialization. The basic constructs include automatic statistics collection and reporting for user-defined resources, queues, entity classes and processes. Extensibility is provided through user-defined statistical variables and access to all statistical values.

An HSL statistic should be thought of as an object; that is, an instance of a system-defined object class having several numerical and functional attributes. Its numerical attributes are maintained automatically and include the number of observations, the mean value, the standard deviation, the minimum value collected so far, the maximum value collected so far and the last collected value. The sole functional attribute is the **collect** operation. Statistics are discussed in greater detail in Section 4.4 in the presentation of the example program.

HSL defines several statistics to be attributes of all user-defined resources, queues, entity classes and processes. The **collect** operation cannot be used with these statistics, as all collection is performed automatically. Their

numerical attributes are available on a read-only basis using dot notation thereby enabling the programmer to use such values to control execution flow or program customized reports. HSL-defined statistics are referenced through their associated resource, queue, entity class or process identifier. For instance, the mean utilization of resource *CPU* is referenced by *CPU*.**util.mean**. The statistic **util** is an attribute of the resource *CPU*, and the real-valued **mean** is an attribute of the statistic **util**.

A statistics reporting feature is provided by the HSL **report** statement. Report statements are situated in the Environment so that reporting conditions can be altered between simulation runs without disturbing the Simulator, or action, part of the model. A report is output upon satisfaction of the specified boolean expression and a final report is always output upon simulation termination. By default, a report consists of a formatted dump of all HSL- and user-defined statistical values. It is possible to limit the report by specifying output of selected groups of statistics (e.g., only those associated with resources). Statistics may also be reset upon report completion if desired. Section 4.4 contains more information about HSL reports as well as a sample report.

HSL does not perform further statistical analysis methods, such as those to support designed experiments. However, it is possible to define one or more special exogenous processes which serve to oversee the simulation and carry out a controlled experimental sequence, including customized reporting, based on its knowledge of statistical values.

3.3 Object-Oriented Languages

The object-oriented programming perspective, which is considered quite appropriate [2, 25] for discrete system simulation, is supported in HSL by user-defined entity classes. Wegner [40] defines an object-oriented language as one that supports objects, which belongs to classes, which may be defined hierarchically by an inheritance mechanism. In other words,

$$\text{object-oriented} = \text{objects} + \text{classes} + \text{inheritance}.$$

An object is informally defined as consisting of a set of operations and a state that remembers the effect of operations. A class is a template from which objects may be created. All objects of the same class have uniform behavior. Inheritance refers to the ability to define a class as the child of an existing class, thereby inheriting the attributes of the parent class and extending them by defining additional attributes of its own.

HSL meets this definition for an object-oriented language, albeit in a subtle way. The key is to consider the relationship between entities and processes, as formally defined in HSL. An entity class definition specifies only state variables (data members). However, every HSL process (except exogenous processes referenced in the Environment module) is bound to a specific entity class through the required **accepts** clause in the process definition. This is a weaker binding than declaring the process in the class definition but stronger than specifying the entity as a process parameter, because the clause is

syntactically required. A process instance is subsequently bound to an entity class instance (entity) at run-time. The process can therefore be considered an operation of this entity class. An entity class acts as a template for creating entities, each of which contains an identical set of data members and an identical set of processes with which it can be invoked. The object and the class requirements of an object-oriented language have been met.

HSL has furthermore been designed to allow a parent class to be specified in an entity class definition, with the child class inheriting all data members and processes defined for its parent class. For example, given the entity class declarations:

```
entity Base:
  int Battr;
end Base;
Base entity Derived:
  int Dattr;
end Derived;
```

and the process definition:

```
process operate ( ) accepts Base Obj
...
end operate;
```

then the following statements are all valid:

```
Base Boj;
Derived Dobj;
...
write Bobj.Battr, Dobj.Battr, Dobj.Dattr;
call operate ( ) with Bobj;
call operate ( ) with Dobj;
schedule operate ( ) with Bobj in 5.0 secs;
schedule operate ( ) with Dobj;
```

HSL meets the inheritance requirement and therefore also Wegner's definition of an object-oriented language. There is no standard definition of what constitutes an object-oriented language, however, and HSL may not meet other, more stringent, definitions. For example, entity class members are not, and cannot be, protected, so its encapsulation is weak. Also, primitive types such as int and real are treated differently than entity objects. On the other hand, SIMULA also has both these properties and is considered a prototypical object-oriented language. HSL, like SIMULA, allows inheritance from only a single parent. It was decided that the benefits to be derived from multiple inheritance were outweighed by the complexities [35] of its implementation, particularly for a prototype.

The decision to design the language syntax so that processes are bound to entity classes only upon process definition was motivated by the twin desires to enhance the separation of the Environment and Simulator modules and to enhance model readability. The decision to invoke processes through **schedule...with** and **call...with** was also motivated by the desire to enhance

readability. The "object.operation" notation cannot distinguish the synchronous (call) and asynchronous (schedule) methods of process invocation.

3.4 Language Comparisons

A comparison of HSL with well-known simulation languages enables better understanding of where HSL fits into the simulation language landscape. The discussion focuses on languages that support a process view [8] of simulation for modeling discrete systems. The process-oriented (sometimes called process-interaction [1]) approach, has been widely accepted as the best approach for model structuring [5].

For languages that support multiple viewpoints (i.e., processes and events) and continuous as well as discrete simulation, this discussion is limited to language features for process-oriented discrete system simulation. Furthermore, it addresses textual language structure, rather than model development support systems, such as graphical specification tools. We acknowledge that HSL is deficient in the area of model development support, although our current research, discussed in Section 6, includes development of such a system.

The languages with which HSL is compared are divided into two groups: *scenario* languages and *procedural* languages [15]. Scenario languages model by specifying descriptive scenarios, typically in the form of block diagrams, to be carried out by active transactions. Indeed, Banks and Carson [1] distinguish them as "transaction flow" implementations of the process-interaction approach, and Rose [22] distinguishes them as "transaction-oriented." Major scenario languages include SLAM II [21], SIMAN [19] and the older versions of GPSS [31].

Procedural languages are distinguished by inclusion of general-purpose programming constructs in addition to simulation-specific constructs [15]. They are considered lower-level, for simulation purposes, than the scenario languages because they usually provide fewer simulation constructs. However, they are applicable in a wider range of situations due to the power and flexibility provided by the general-purpose constructs. Major procedural languages include SIMULA [3], SIMSCRIPT II.5 [14] and CSIM [33]. Also, GPSS/H [32], a more recent version of that popular language, can be viewed as qualifying for this category. We consider HSL to be another procedural simulation language.

3.4.1 *Software Engineering Considerations.* The framework for language comparison is Golden's software engineering considerations for the design of simulation languages [10]. Software engineering for simulation concerns the efficient development, expansion and maintenance of large simulation models. The focus is on efficient use of human resources and the effect of language design. The efficient use of computer resources, which concerns space and time requirements for compilation and execution, is of secondary importance.

The most important concept of structured program design is *modularity*. Large models should be subdivided along clear, functional lines into smaller

components (modules) that are of a more manageable size. This top-down approach eases the development of a software model through the use of stepwise refinement. Model maintenance is facilitated because changes and corrections will tend to fall within modules and are much easier to implement without affecting other software components.

The types of *control structures* used within a program are also important to structured design. It has been proven [4] that any computer program can be written using only control structures for sequence, selection and iteration. One of the software engineering measures, then, of a programming language is its ability to support structured control statements.

The ability to define flexible *data structures* to use in conjunction with control structures is equally important. The data in the model should logically correspond to physical components of the system being modeled. Other software engineering considerations are improved *program readability* through self-documenting code and meaningful variable names and controlled *data communication* between program modules through use of parameters and localized data. Since there is little real variation among the languages regarding data communications, it is not discussed further.

3.4.2 *Scenario Languages.* Pursuant to these software engineering concerns, procedural languages (e.g., SIMULA, SIMSCRIPT II.5, HSL) hold a decided advantage over scenario languages (e.g., SLAM II, SIMAN, older versions of GPSS). First, consider modularity and support for top-down design. Top-down, or hierarchical, design in a process-oriented simulation means building an initial or prototype model from a few abstract processes, then refining those determined to be significant into subprocesses to introduce greater detail. Scenario languages, which represent models as networks of blocks, are designed to model *sequence,* not hierarchy. Model refinement to introduce more detail results in replacement of entire sequences, which complicates model reverification. Scenario languages support process extensibility through general programming language *inserts*, which are typically FORTRAN subroutines. Thus hierarchical refinement results in a model that consists largely of FORTRAN code. Even if all the language statements are available in this subcode, the programmer must interface with them using FORTRAN syntax, which is inconsistent with the top level of the model. SIMAN, however, does provide high-level modularity in the separation of a model into an *experimental frame*, which encapsulates experimental control information and a *system model*, which contains the action statements. This separation, inspired by Zeigler [41] and emulated by HSL, allows the model to be exercised by changing only the experimental frame.

As for control structures, the scenario languages depend on the base language to provide full algorithmic power to the programmer through inserts. The inclusion of language constructs for assignment, selection (if) and iteration (loops), as exemplified by GPSS/H, vastly reduces the need for inserts. The data structures available are typically attributed entities implemented as arrays. They must, if needed in the base language code, be accessed as global arrays. The use and length of user-defined identifiers is limited, thus inhibiting program readability.

In general, the major advantage of scenario languages is that they provide very high-level support for simulation constructs through block diagrams and thus are expected to be more easily understood by the modeler and conducive to graphical specification techniques. They also enable small systems to be specified quickly and easily. Unfortunately, as the model expands in scope and detail, the model becomes more difficult to understand and the programmer is forced to compromise good program structure.

3.4.3 *Procedural Languages.* The procedural languages embody software engineering techniques in different ways and hence cannot be compared with HSL in a uniform manner. We only highlight certain comparable features, on a selective basis. Such languages typically support modular program expansion through one or more procedural constructs. For example, SIMSCRIPT II.5 (hereafter referred to as SIMSCRIPT) supports hierarchical refinement by a **call** to a **routine**, SIMULA by invoking a **procedure**, and HSL by a **call** to a **process**. SIMSCRIPT and SIMULA support lateral refinement by an **activate** of a **process**, and HSL by a **schedule** of a **process**. HSL is conceptually simpler and more flexible in that a process may be invoked in either fashion. Another aspect of modularity is the separation of model control from model action, as exemplified by SIMAN's experimental frame and system model. The closest feature to an experimental frame in SIMSCRIPT is use of the **preamble** section in conjunction with the **main** program, to declare globals and initialize simulation. The main program of a SIMULA model serves a similar function. The HSL Environment module contains declarations of global variables plus simulation control statements and is roughly equivalent to a SIMAN experimental frame. The HSL Simulator is equivalent to a SIMAN system model. The HSL Environment and Simulator modules may also be compiled separately, which facilitates experimentation.

Procedural languages provide a full complement of structured control statements. Compound statements in SIMULA are enclosed in a **begin..end** pair (which reflects its Algol origins), while those in SIMSCRIPT and HSL are not distinguished other than by position within an enveloping structure (such as **loop..end loop**). SIMULA has two assignment operators, one for regular variables and one for reference, or pointer, variables. SIMSCRIPT and HSL do no pointer assignment and thus have only one assignment operator. These three languages, as well as GPSS/H, contain an **if..then..else** construct for selection. HSL additionally offers a **case** statement for more concise selection. SIMSCRIPT and SIMULA have both **for** loops and **while/until** loops, while HSL supports only a generic **loop** construct which relies on execution of an enclosed **break** statement for loop exit. SIMSCRIPT and SIMULA provide a **goto** statement for uncontrolled flow transfer within a procedure, while HSL does not support uncontrolled flow transfer. HSL provides the **next** statement to branch to the next iteration of a loop, the **break** statement to exit a loop, the **exit** statement to prematurely terminate a process and the **return** statement to leave a function. In summary, HSL emphasizes selection over iteration and does not permit uncontrolled transfer of execution flow.

SIMULA and HSL provide more sophisticated data structuring than SIMSCRIPT. Each SIMSCRIPT process can define a set of data attributes which can be referenced once a process instance is created. SIMULA provides the class construct, object templates which contain both data and procedural attributes. Additionally, hierarchies of classes may be defined, with a child class inheriting all attributes of its parent. These properties characterize SIMULA as an object-oriented language, as discussed in Section 3.3. HSL provides a similar feature for defining entity class hierarchies with inheritance. The major difference is that processes defined for use with an entity class are not declared in the class definition, which is contained in the Environment. The association is made in the process definition, which is contained in the Simulator. This support in SIMULA and HSL for extensible and abstract data types enables program data structures to more closely model structures in the system being modeled thereby facilitating model verification.

SIMSCRIPT has adopted the Cobol approach to self-documenting code by allowing long variable names and superfluous words to result in an English-like syntax. This also helps to bridge the gap between the modeler and the programmer; the model is more easily understood by the modeler and the language contains all the programming features needed by the programmer. SIMULA maintains readability through the use of long, meaningful variable names and clear, conceptually simple Algol-like syntax. Unfortunately, SIMULA has never achieved widespread use in America, although popular in Europe. It is clearly, however, a programmer's language. HSL strives for improved readability through long, meaningful variable names, requirement of identifiers on all **end** statements (e.g., **end if; end** GenJobs;), the free use of "white space" (spaces, tabs and carriage returns) within and between statements, and certain sacrifices in syntactic purity for the sake of clarity, such as the use of **call..with** and **schedule..with**. These features help to bridge the gap between the modeler and the programmer, as much as that is possible in a general-purpose simulation language.

3.4.4 *Language Comparison Summary.* In summary, HSL addresses Golden's simulation language software engineering considerations more fully than any of the scenario languages. It also provides more flexible data structures than the procedural language SIMSCRIPT. Distinguishing HSL from SIMULA is more difficult because both provide process and data extensibility as well as structured control statements. HSL supports better simulation control through separate Environment and Simulator modules, allows processes to be invoked either synchronously or asynchronously, provides language primitives to represent simulation constructs (statistic, queue, and resource objects, automatic statistics collection and reporting) and is less intimidating syntactically to the modeler than SIMULA.

4. HSL PROGRAM EXAMPLE

To illustrate the use of HSL, we present a program for a simple "machine shop model." Numerous versions of such manufacturing shop models have

been reported in the literature (e.g., see Ford and Schroer [7] and Ketcham et al. [13]).

4.1 Machine Shop Model

A hypothetical machine shop (or suboperation within a shop) produces metal shafts for large electrical motors. Two types of shafts are produced, lathed shafts and drilled shafts. The former is characterized by a groove which is lathed near one tip of the shaft, the latter by a threaded hole which is drilled in a similar spot.

To save inventory costs, shafts are produced only upon receipt of an order. The interarrival time of orders is exponentially distributed. Orders are given to a cutter, who cuts shafts to the specified length from a continuous (and infinite, for our purpose) strand of metal having a fixed diameter. Shafts to be lathed are placed on one conveyor and those to be drilled are placed on another. The conveyers lead to the lathe area and drill area, respectively, where the shafts are machined. After machining, the shafts are placed on conveyers leading to a single inspection area where each is accepted or rejected. The operation beyond that point is not of interest for this example.

The performance objective of the study is to minimize the number of lathing machines and drilling machines necessary to fill orders in a "timely" fashion. What constitutes "timely" is not critical here, but it clearly implies a need to minimize waiting time in the machining and inspection areas. Because shafts do not arrive at these areas in a uniform fashion, some waiting occurs unless a large number of machines are available. Therefore a tradeoff between waiting time and number of machines is necessary.

A diagrammatic version of the resulting HSL model is shown in Figure 5. This figure has been transcribed from the N-CHIME Graphics-based HSL Editor (to be described in Section 6), so a short explanation of symbols is in order. Each ellipse represents a **process**. Each box represents a **resource**. A nondirected line represents the **request** and **release** of a resource within the connected process. A solid arrow represents hierarchical process invocation through **call**, with the arrow directed toward the subprocess. A dashed arrow represents lateral process invocation through **schedule**, with the arrow directed toward the process being scheduled. The label on an arrow identifies the **entity class** of the entity associated with the invoked process.

Inspection of Figure 5 reveals the simulation structures in the model: processes, resources and entities.

(1) Each process in the model represents an operation, or suboperation, of the shaft manufacturing procedure. Each operation is applied in sequence, in the direction indicated by the arrows in the figure. The flow of entities through the system via processes is as follows:
   (a) The **GenOrder** process drives simulation activity by periodically generating orders for shafts and sending them to the cutting area;
   (b) The **Cutting** process cuts each shaft to length as specified in the order then sends it to the appropriate machining area;

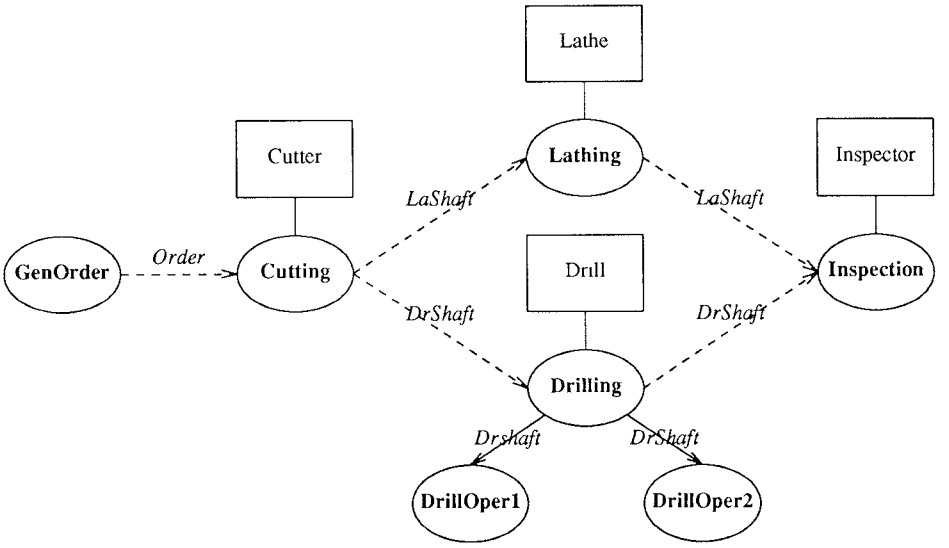

Fig. 5. Machine shop model.

(c) The **Lathing** process machines only the lathed shafts, then sends them to the inspection area.
(d) The **Drilling** process machines only the drilled shafts, then sends them to the inspection area. The drilling operation consists of two sequential suboperations, which are modeled as subprocesses. They are the following:
  (1) the **DrillOper1** process, which positions the shaft and drills the threaded hole to the specified depth, and
  (2) the **DrillOper2** process, which repositions the shaft under a different drill bit and drills a concentric smooth hole for countersinking.
(e) The **Inspection** process inspects all shafts after machining and accepts or rejects each.

(2) Beyond order receipt, each step of the operation requires the use of consumable system resources. The resources for this system are the following:
  (a) The **Cutter**, which requires a certain amount of time to cut each shaft to length. The system includes one, but this number can easily be modified, as will become apparent when the program is presented.
  (b) The set of lathing machines, **Lathe**, which performs the lathing operation. The set is modeled as a specified number of identical machines operating in parallel and drawing from a common queue.
  (c) The set of drilling machines, **Drill**, which performs both drilling operations. As above, the set is modeled as a specified number of identical machines operating in parallel and drawing from a common queue.

(d) The **Inspector**, which requires a certain amount of time to inspect each finished shaft.

(3) The model entities represent the objects that move through the operation sequences. Individual entities are created, activated and deleted dynamically. The classes of entities in the model are the following.
   (a) an **Order**, which represents the purchase order for a specified number of each shaft type,
   (b) a **LaShaft**, which represents a lathed shaft, and
   (c) a **DrShaft**, which represents a drilled shaft.

4.2 Machine Shop Model Environment

The Environment module of the model's HSL program is listed in Figure 6. It includes declarations of the resources and entity classes required by the model, declarations of other global identifiers and model control specifications. The four required resources are declared first. Each resource has a built-in queue, which can be accessed if needed, and all queueing is handled automatically. The capacity for each is enclosed in brackets. A resource request specifies the number of units requested, which must be no larger than the resource capacity. For this model, each request is for one unit and each unit of capacity represents one physical machine. All units of the capacity are served by the same queue. Therefore the number of machines to be used for lathing or for drilling is modified simply by changing the capacity.

There are two main classes of entities to represent purchase orders and shafts. A purchase order, which is vastly simplified for this example, contains the quantity of lathed shafts and drilled shafts, plus specification of the width of lathed sections and depth of holes. Although the two types of shafts are processed differently, both are shafts, so a generic entity class called **Shaft** is defined to contain characteristics common to both. The declarations of the **LaShaft** and **DrShaft** entity classes confirm their common heritage. Every child class inherits the attributes of its parent; every entity instantiated during simulation contains all defined and inherited attributes. The **Shaft** class is used here as an abstract class, which means that it exists only as an ancestor; all entities are either **LaShafts** or **DrShafts**.

An extra statistical variable is declared to confirm the acceptance rate of inspected shafts. The acceptance rate is specified below, but acceptance or rejection of a specific shaft is determined by a random variate. The decision to accept or reject is verified by collecting a value for this time-independent statistic, the reported average of which reflects the percentage of accepted shafts.

Several global identifiers are declared to represent model parameters. Since they are declared as constants, they can be modified only between simulation runs. The more such parameters a model contains, the more flexible it becomes in terms of the range of factors that can be easily considered in an experimental study. If they were declared as variable identifiers, their values could be modified, for example, by a "metaprocess," an exogenous process designed to control an entire experimental study by

```
!                    Environment module for Machine Shop model

!  This is a simulation of a simple milling operation for a shop which
!  produces shafts (only on demand) for large electrical generators.
!  The supply of raw material used to make the shafts is limitless.

model MachineShop

    !  Declaration of system resources, with capacity of each in brackets.

    resource [1] Cutter      fcfs ;
    resource [3] Lathe     : fcfs ;
    resource [2] Drill     : fcfs ;
    resource [1] Inspector : fcfs ,

    !  There are two main entity classes: to represent order sheets, and
    !  shafts.  Subclasses are defined to represent the two types of shafts.

    entity Order :
        int  LaQuantity ,
             DrQuantity ,
        real TotalLength ,
             LatheWidth ,
             DrillDepth ;
    end Order ;

    entity Shaft .
        int  Serial ,
        real Length ;
    end Shaft ,

    Shaft entity LaShaft .
        real LaWidth ,
    end LaShaft ;

    Shaft entity DrShaft
        real DrDepth ,
    end DrShaft ;

    stat PctGood . ind ;  ! Statistical variable to confirm acceptance rate.

    !  Global identifiers, to change experimental parameters between runs.

    int SerialNumber  = 0 ;                  ! ALL TIMES HERE ARE IN SECONDS.
    constant real ArrivalMean :=  60.0  ,    ! Order interarrival mean
                  CutTime     :=   7.0  ,    ! Time needed to cut shaft
                  InspectMean :=   7.0  ,    ! Avg amount of time to inspect
                  InspectSD   :=   2.0  ,    ! Std Deviation of time to inspect
                  AcceptProb  :=   0 95 ;    ! Prob of acceptance by inspector

    !  Model Control statements.  The exogenous process GenOrder will kick
    !  off the simulation.  A full report will be generated every hour of
    !  simulated time.  After 3 simulated hours, the simulation will halt.

    start GenOrder( ) ,
    report rptlapse >= 1 0 hrs ;
    stop clock >= 3 0 hrs ;

end MachineShop ;
```

Fig. 6. HSL Environment module for machine shop model.

awakening periodically to check the state of the simulation and adjust parameters accordingly.

The model control specifications for this example are few and simple. Simulation is to commence by executing the exogenous process **GenOrder**. A full, automatically generated, statistical report is to be output every hour of simulation time (**rptlapse**, the time lapse since the previous report, is maintained by HSL). Execution, which commences by default at time 0.0, is to be halted at simulation time 3.0 hours.

4.3 Machine Shop Model Simulator

The Simulator module of the machine shop model is listed in Figure 7. It is observed to be a collection of processes, one for each ellipse in Figure 5, and functions. The model includes one function for illustrative purposes; it does not appear in Figure 5 because functions cannot include simulation constructs.

The exogenous process, **GenOrder**, must be **started** in the Environment. All other processes are endogenous; each **accepts** an entity of a particular class from within the simulated model. This entity is then used within the context of the process invoked on its behalf. Thus, for example, the **Cutting** process uses the information from the **Order** entity (identified by **Sheet**) to determine how many of each type of shaft to create. After each individual shaft entity is created, it is scheduled for the next operation. **Cutting** continues after issuing the schedule, looping to create more shafts as required. The **Inspecting** process is special in that it accepts either **LaShaft** or **DrShaft** entities. This can be seen in Figure 5. This is accomplished by specifying the name of their common ancestor class, **Shaft**, in the process **accepts** clause. Thus in effect, entity subclasses inherit processes from their ancestors as well as data attributes.

The machine shop model as written contains examples of both process views discussed in Section 3.1: the localized, or sequential, view is reflected by the scheduling of shafts for inspection after being machined, and the hierarchical view is reflected by the hierarchical breakdown of the Drilling process into two subprocesses: DrillOper1 and DrillOper2. The model can be revised to support either view exclusively, but at a loss of correspondence to the real system.

As previously described, the **schedule** and **call** statements are two options for interprocess communication in HSL. While a **schedule** invokes another process in an asynchronous manner (i.e., independently continuing with its own execution), a **call** initiates synchronous behavior. That is, the calling process suspends itself until the called process(es) return(s) control. The example program illustrates this kind of hierarchical control flow in the **Drilling** process, where two different subprocesses are called in sequence. A fork-join capability is effected by naming several processes in the call statement; all subprocesses are invoked at once and the calling process is suspended until all have returned.

However, the hierarchical nature of HSL can be exploited much more extensively in the example program and in HSL programs in general. Any

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Generate orders forever  Simulation termination conditions must be specified
! in the Environment, because there is no provision here for halting arrivals
! This exogenous process must be initiated by a 'start' in the Environment.

process GenOrder ( )
    Order Sheet ,
    loop
        delay expo ( 1, ArrivalMean ) ;
        Sheet.create ,
        Sheet LaQuantity   = randint ( 2, 1, 10 ) ,   ! flat shaft quantity
        Sheet DrQuantity  := randint ( 3, 2, 4 ) ,    ! drilled shaft quantity
        Sheet.TotalLength  = 26 0 ;                   ! shaft length
        Sheet LatheWidth   =  2 5 ,                   ! width of lathed groove
        Sheet DrillDepth   =  1 2 ;                   ! Depth of threaded hole
        schedule Cutting ( ) with Sheet ,
    end loop ,
end GenOrder ,

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  The cutting process  Orders received are handled first-in-first-out  The
!  supply of raw material is limitless  The cutter cuts all to-be-lathed shafts
!  then to-be-drilled shafts  At each cut, a shaft entity is created, assigned
!  a serial number, and scheduled for machining after 10 seconds on conveyer

process Cutting ( ) accepts Order Sheet
    int     count ,
    LaShaft LatheShaft  ,
    DrShaft DrillShaft ,
    Cutter request ( 1 ) ,
        count  = 0 ,
        loop
            if count = Sheet LaQuantity then break , end if ;
            SerialNumber  = SNIncrement(SerialNumber) ,
            delay CutTime ,        ! cut off a shaft
            LatheShaft create ,
            LatheShaft Serial   := SerialNumber ,
            LatheShaft Length   := Sheet TotalLength ,
            LatheShaft LaWidth := Sheet LatheWidth ,
            schedule Lathing ( ) with LatheShaft in 10 0 secs ,
            count   = count + 1 ,
        end loop ,
        count  = 0 ,
        loop
            if count = Sheet DrQuantity then break , end if ,
            SerialNumber  = SNIncrement(SerialNumber) ,
            delay CutTime ,        ! cut off a shaft
            DrillShaft.create ,
            DrillShaft Serial     = SerialNumber ,
            DrillShaft Length     = Sheet TotalLength ,
            DrillShaft DrDepth    = Sheet DrillDepth ,
            schedule Drilling ( ) with DrillShaft in 10 0 secs ,
            count   = count + 1 ,
        end loop ,
    Cutter release ( 1 ) ,
    Sheet destroy ,      !  For initial version, order sheet is finished
end Cutting ,

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This represents the "lathing" operation, in which a groove is lathed near
! one tip of the shaft  This operation is simple, so no subprocesses are
! called  The shaft then spends 15 seconds being conveyed to the inspector
```

(a)

Fig. 7 HSL Simulator module for machine shop model.

```
process Lathing ( ) accepts LaShaft Thing
    Lathe.request ( 1 ) ;
        delay 5.0 ;                    ! position the piece
        delay 8.5 * Thing.LaWidth ;    ! Work shaft edge at 8.5 secs/inch
    Lathe.release ( 1 ) ;
    schedule Inspection ( ) with Thing in 15.0 secs ;
end Lathing ;

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  Two part "drilling" operation.  First, a threaded hole is drilled, then a
!  concentric countersink hole (same machine).  Then 15 seconds being conveyed.

process Drilling ( ) accepts DrShaft Thing
    Drill.request ( 1 ) ,
        call DrillOperA ( ) with Thing ;
        call DrillOperB ( 0.35 ) with Thing ;
    Drill.release ( 1 ) ;
    schedule Inspection ( ) with Thing in 15.0 secs ;
end Drilling ;

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  This process represents drilling the threaded hole.  For brevity of
!  presentation, the process is not refined.

process DrillOperA ( ) accepts DrShaft Thing
    delay 5 0 ;                    ! position the shaft
    delay 10.0 * Thing.DrDepth ;   ! Drill threaded hole 10 secs/inch
end DrillOperA ;

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  This process represents drilling the smooth countersink hole.  The depth is
!  a parameter representing percent.  Rate of drilling is 7 seconds per inch.

process DrillOperB ( real PctDepth : in ) accepts DrShaft Thing
    delay 3.0 ;                          ! Reposition shaft for the other bit
    delay 7.0 * Thing.DrDepth * PctDepth ; ! Drill countersink hole.
end DrillOperB ;

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Shaft inspection process   The inspector handles either kind of shaft the
! same way, so this process accepts an entity of their common parent.
! The user-defined statistic will just verify the acceptance probability.

process Inspection ( ) accepts Shaft Product
    Inspector.request ( 1 ) ;
        delay normal ( 4, InspectMean, InspectSD ) ;
    Inspector.release ( 1 ) ;
    if rand01 ( 5 ) <= AcceptProb then
        PctGood.collect( 100.0 ) ;
    else
        PctGood.collect(   0 0 ) ;
    end if ;
    Product destroy ;          ! Finished with entity, so reclaim its storage
end Inspection ;

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Useful (but unsophisticated) function to increment Serial Number counter.

function SNIncrement (int Number ) returns int
    return Number + 1 ;
end SNIncrement ;
```

(b)

Fig. 7. (*Continued*).

process including a **delay** statement, each simulating some constant or variable time delay in the process' action sequence at that point, can be refined by the modeler into a more detailed representation of that delay. The modeler can simply do the following:

(1) replace a **delay** statement with a **call** statement to invoke a new subprocess at that point;
(2) program that subprocess to model the time delay at a finer level of granularity; add the resulting subprocess to the Simulator's collection of processes;
(3) update the model Environment to reflect any requirements (e.g., resources) imposed by the new process.

Such hierarchical *refinement* of an HSL model is one of the significant features of the language. The opposite of refinement, namely *compression*, can also be carried out easily by replacing unnecessarily detailed subprocesses with simple delay statements in the calling processes. As a result, a simulationist has the flexibility of zeroing in on the troublesome parts of a model, in great detail (to whatever hierarchical level desired), while simplifying (and minimizing simulation effort) those parts of the system that exhibit predictably stable and noncritical behavior.

4.4 Machine Shop Model Output

The Machine Shop model has been written such that all output is generated automatically by HSL pursuant to the conditions specified in the model control section of the Environment. The modeler is not limited to using the provided fixed report formats; customized reports may be written within any process using write statements in conjunction with the identifiers of HSL and user-defined statistical values and variables. An automatically generated full HSL report is, however, always output upon termination of the simulation run. The Machine Shop model has been written to output a complete report after each hour of the three-hour simulation.

The experimental conditions for this model are also contained in the Environment. One simulation run is to be made using the parameter values specified in the Environment. HSL does not include a feature for designing and conducting experimental studies, but it does provide the programming tools for implementing these capabilities. An experiment control mechanism can be included by careful programming of one or more exogenous processes which act as "metaprocesses" to monitor the state of the simulation run and react if necessary. Experiments can also be controlled externally by providing for interactive user entry of parameter values in a startup process. This is accomplished by defining an exogenous process to request values using read and write statements before any further simulation activity takes place. As a result, a sequence of simulation runs can be performed without recompilation of the model. This feature has proven very useful, when used in conjunction with file redirection, for exercising simulation models in a "batch" mode during nonpeak computing hours. File redirection is an operat-

ing system feature which allows keyboard input and display output to be redirected from/to disk files.

Figure 8 shows an abbreviated default HSL report produced by the Machine Shop model run under the conditions specified in the program listing. Statistical reports are generated for all user-defined statistics, queues, resources, entity classes and processes. The abbreviated report contains one or more examples of each. No user-defined queues were included in the model, but a queue report is included within each resource report for its built-in queue.

Each line of numerical output represents one statistic. The values output for each statistic are the following: number of observations (OBS), mean value (MEAN), standard deviation (STDEV), minimum value collected (MIN), maximum value collected (MAX) and the most recent value collected (CURRENT). If the number of observations is output as a real number (as opposed to an integer), then the statistic is time-dependent and the OBS value represents the length in time units of the observation period. Examples of time-dependent statistics are queue length and resource utilization, in which each collected value must be weighted by the time period for which it applies.

The statistic report for user-defined statistic *PctGood* serves to confirm, through its mean value of 94.8966, that the 95% acceptance ratio specified in the Environment was met. A total of 1450 observations was collected, representing the number of shafts completing inspection (the collection point for this statistic).

A resource report contains several statistics. *Utilization* is the fraction of time that the resource was not idle, i.e., that at least one unit was in use. *Occupancy* is the fraction of total resource capacity in use over the time period. For example, if a resource of capacity two had both units busy for one second and one unit busy for the next second, then the utilization for that time period is 1.0 and the occupancy is ((2 busy * 1 sec) + (1 busy * 1 sec))/(2 units * 2 sec) = .75. If the resource is of capacity one, utilization and occupancy are identical. Other resource statistics are number of *users*, *capacity* (which can be varied under program control using the **chrcap** resource attribute) and periods of *idle time*.

Since every resource has a single built-in queue (resource attribute **q**), a queue report is embedded within the resource report. Queueing statistics are entity *wait* time, queue *length* and queue *capacity* (which can be varied under program control using the **chqcap** queue attribute). An array of resources has one queue per array element.

An entity class report includes statistics gathered from the individual entities generated from that class. Two statistics are collected: *population*, which tracks the number of "alive" entities (those that have been created but not yet destroyed) in the system at any given time and *lifetime*, which is collected when an entity is destroyed to identify the length of time since its creation.

Finally, two statistics are collected for each process in the model, including exogenous processes. These statistics are gathered from individual activations of processes, much like entity class statistics are gathered from

FINAL REPORT simulation time: 10800.65

pctgood STATISTIC REPORT 10800.65

	OBS	MEAN	STDEV	MIN	MAX	CURRENT
	1450	94.8966	22.0144	0.0000	100.0000	100.0000

cutter RESOURCE REPORT 10800.65

	OBS	MEAN	STDEV	MIN	MAX	CURRENT
UTILIZATION	10800.65	0.9481	0.2217	0.0000	1.0000	1.0000
OCCUPANCY	10800.65	0.9481	0.2217	0.0000	1.0000	1.0000
USERS	10800 65	0.9481	0.2217	0.0000	1.0000	1.0000
CAPACITY	10800.65	1.0000	0.0000	0.0000	1.0000	1.0000
IDLE TIME	11	50.9227	40.1177	4.8591	108 3496	105.2888
queue·						
WAIT	151	195 1099	139.6290	3.0325	531.3901	28.8274
LENGTH	10800.65	2.7387	2.2013	0.0000	9.0000	2.0000
CAPACITY	10800.65	65536.0000	0.0000	0.0000	65536.0000	65536.0000

order ENTITY CLASS REPORT 10800.65

	OBS	MEAN	STDEV	MIN	MAX	CURRENT
population	10800.65	3 6868	2 2757	0.0000	10 0000	3.0000
lifetime	161	245.9862	146 2255	31 0325	622.3901	310.5030

shaft ENTITY CLASS REPORT 10800.65

	OBS	MEAN	STDEV	MIN	MAX	CURRENT
population	10800.65	10.7982	2.6832	0.0000	17 0000	12.0000
lifetime	1450	80.0558	12.7784	52.2763	122.9799	79.2536

cutting PROCESS REPORT 10800.65

	OBS	MEAN	STDEV	MIN	MAX	CURRENT
calls	10800 65	3.6868	2.2757	0 0000	10.0000	3.0000
thru time	161	245.9862	146.2255	31.0325	622 3901	310 5030

lathing PROCESS REPORT 10800.65

	OBS	MEAN	STDEV	MIN	MAX	CURRENT
calls	10800.65	3.6202	1.5110	0.0000	7.0000	3.0000
thru time	959	30 6861	4.5579	26 2500	47.2500	31 5000

NORMAL SIMULATION TERMINATION

Fig. 8. Abbreviated HSL report for machine shop model

individual entities. The *calls* statistic tracks the number of activations of the process that exist at any given time. The *thru time* statistic is gathered at process termination to measure the length of time since its activation. Both are directly analogous to the entity class statistics.

4.5 Results Summary

Simulation results obtained by running the example Machine Shop model are summarized in Table I for resources and Table II for processes. For resources, the occupancy statistic is more useful than utilization because both Lathe and Drill are multicapacity resources where each unit of capacity represents one machine and occupancy represents the fraction of the total machine capacity in use over the course of the simulation.

Table II contains information relevant to system throughput in the **thru time** statistic. The number of observations for each process represent the number of entities, of the class it accepts, passing through that process. The 161 observations for *Cutting* represent the number of orders that were completed by the cutter. Likewise, the observations for *Lathing* and *Drilling* represent the number of shafts drilled and lathed, respectively. Both shaft types are included in the numbers for *Inspection*. Notice that the sum of observations for *Lathing* and *Drilling* is greater than the observations for *Inspection*; this is because several shafts were awaiting inspection when the simulation ended.

5. HSL STATUS

The HSL language has been implemented in an interpreter-based prototype. Language processors are classified as being either interpreters or compilers, although most exhibit characteristics of both to some degree. A pure interpreter is characterized by its complete processing of one program statement at a time. Each statement is recognized and its actions carried out before the next statement is considered. A pure compiler is characterized by its translating of the entire program into machine code. The compiled program is then executed with no further compiler involvement.

The HSL interpreter is a hybrid, combining methods of both interpreters and compilers into a two-phase process. Specific structures and methods are presented by Rozin and Treu [26]. The key to this two-phase approach is the definition of a machine language for an abstract (nonexisting) machine. This simple language acts as an intermediary between the two phases. The first phase translates an HSL program into abstract machine code, much like a pure compiler. The second phase, which is completely separate from the first, recognizes and executes each abstract machine statement in turn, like a pure interpreter.

The HSL interpreter includes a separate translation phase for the Environment and the Simulator modules. This allows modifications local to one or the other to be made without necessitating recompilation of both. This is quite useful for modifying global parameters (in the Environment) between experiments. The prototype, however, has no mechanism for determining

Table I. Selected Statistics for Machine Shop Resources

Resource	occupancy		queue wait				queue length		
	mean	stdev	obs	mean	stdev	max	mean	stdev	max
Cutter	0.948	0.222	151	195.110	139.629	531.390	2.739	2.201	9
Lathe	0.778	0.314	575	7.435	3.530	21.000	0.396	0.586	3
Drill	0.529	0.445	224	8.956	3.297	21.760	0.186	0.435	2
Inspector	0.942	0.233	1375	19.655	12.232	61.040	2.508	1.875	8

Table II. Selected Statistics for Machine Shop Processes

Process	calls			thru time			
	mean	stdev	max	obs	mean	stdev	max
Cutting	3.687	2.276	10	161	245.986	146.226	622.390
Lathing	3.620	1.511	7	959	30.686	4.558	47.250
Drilling	1.706	1.340	6	498	26.968	4.977	44.700
Inspection	5.472	2.154	12	1450	25.638	12.800	64.431

dependencies between the modules, so this feature has not realized its full potential.

The decision to implement HSL as an interpreter was a natural consequence of the decision to first develop a prototype. The execution of an interpreted model is certainly slower than that of a compiled model, but the programming effort to implement an interpreter is much less. This is because the code to carry out language semantics, which in a simulation language can be very complex, is written in the high-level language of the interpreter not in the machine code of the target computer. In addition, the intermediate code produced by the translation phase of the HSL interpreter, unlike the machine code produced by a compiler, is machine-independent, thus enhancing portability. Another pleasant side-effect is that an interpreter retains source program information, such as symbol tables, throughout model execution. This facilitates the development of interactive run-time tools such as debuggers.

The HSL interpreter itself was written using the C++ language [36]. This language was chosen after a deliberate selection process. Our criteria were based on the desires to develop the interpreter on a Unix-based platform and to explore techniques of object-oriented programming. Given the languages available at the University at that time (early 1988), the choices were narrowed to C++ and Modula-2. A study was conducted and C++ selected due to its superior execution performance (both in terms of time and space) and its anticipated wider popularity and availability.

The interpreter was originally developed on an NCR Tower running the Unix operating system. It has since been ported to a Unix-based VAX system, and to a DOS-based NCR personal computer. Table III summarizes

Table III. Characteristics of Two Versions of the HSL Interpreter

Attribute	NCR Tower	NCR PC
Processor	Motorola 68020	Intel 80386
Operating System	Unix	DOS
Interpreter size:		
-- lines of C++ code	17602	18967
-- bytes	413676	419117
Translation phase speed		
-- lines of HSL code per second	85	53
Interpretation phase speed		
-- lines of HSL code per second	1180	210

the storage space and execution speed characteristics of the Tower and PC versions. The PC version is somewhat larger because it includes the Real-Time Model Interrupter software (see Section 6). The prototype interpreter is limited in the size of HSL model it can process, depending on memory availability. The limitation is more severe for the PC version. The largest HSL models written to date have been about 1200 lines in length. The execution speed figures are approximations based on the Machine Shop model and vary widely with model complexity.

HSL has been used for the most part within the University community. In addition to NCR-sponsored research, HSL has been used in the graduate-level simulation course for individual term projects, for the simulation of an emergency response system written as part of a Master's project and for the performance evaluation of a retail transaction processing system written for NCR as an independent project.

6. HSL-ORIENTED INTERFACE SYSTEM

Comprehensive, user-oriented design of software tools for support of any interactive, computer-based application demands that the designers attend to both the *language* software to enable effective representation of the application of interest and the *interface* software to bridge any remaining gaps between the language software and the user's ability to utilize it effectively. The focus of this paper is on the HSL language software. Our current research efforts focus on the design of the interface software.

N-CHIME is the NCR-sponsored, Cohesive, HSL-oriented Interactive Modeling Environment. It is a prototype interface software system designed for use with the HSL language interpreter and implemented on a PC-based platform [29]. The primary mission of N-CHIME is to assist the HSL user in carrying out various modeling and simulation tasks. This is accomplished by following user-oriented software design principles. It is important for the interface to *facilitate* learning to use the interaction techniques and software

tools and to *provide context* by displaying carefully structured and supportive visual information and directions. The current prototype supports these principles and is being extended to also *adapt* to the needs and preferences of different users. The capability to automatically adapt interface behavior to individual users requires the effective application of expert systems technology [30].

The visible features of N-CHIME were determined as part of a three-pronged design approach which encompassed principles and factors of user-oriented design, interface design and application software engineering [38]. One aspect of the design analysis addressed the needs of users for guidance and direction toward completing specific tasks. An existing method for structuring interactive simulation sessions [37] was refined for this purpose. The use of structures to represent the various phases of an interactive session allows the interface to give the user a sense of orientation (or context) with respect to where they are in the task domain, where they are going or can go next and what they can do there.

The ability to provide guidance and direction to the user is enhanced by superimposing a task plan, or directed graph, over the session structure. Each such plan is designed to reflect the required steps that must be taken to carry out a specific simulation task. The session and task structures enable us to define for the interface a series of menus which present the user with the most appropriate choices of tools and other options in a given context.

Each of the session phase structures is supported by one or more interface features, which are realized in the visible interface through software tools. An example should clarify these relationships. Suppose the goal of an interactive simulation session is to begin building a new simulation model. The initial phase of that session is *conceptualization* of the desired model. Two operations which support conceptualization are the browsing of an existing library of models in search of a similar model and visually-aided model creation. The browsing feature is supported by an interactive Browser tool and the creation feature by a visually-oriented, graphics-based HSL editor.

Pursuant to the analysis, we identified a number of appropriate tools. They include the following:

(1) Browser Search and Retrieval Tool,
(2) Graphics-based HSL Editor,
(3) Text Editor,
(4) HSL Interpreter Front-End (translation phase),
(5) HSL Interpreter Back-End (interpretation phase),
(6) Real-Time Model Interrupter and Manipulator,
(7) Graphic Data Display Tool, and
(8) Statistical Model Comparison Tool.

With the exception of the Statistical Model Comparison Tool, these capabilities now exist in the N-CHIME prototype. Two of them are of particular interest.

The Graphics-based HSL Editor supports model conceptualization, development and refinement through inclusion of specific visual structures to represent major HSL language constructs: processes, call statements, schedule statements, entity classes, resources and resource requests. These structures are created, deleted and moved around the screen using mouse-based direct manipulation techniques. The figure that illustrates the example model in Section 4 is a reproduction of a Graphics-based HSL Editor screen.

The Real-Time Model Interrupter and Manipulator tool allows the user to interrupt an executing model, make certain changes and then resume execution. Its purpose is to enable the user to view and optionally modify the values of global parameters during the simulation run, based on visual inspection of intermediate reports or other output. The development of this debugging tool was facilitated by the presence of symbol table information in the interpreter. However, because the interpreted code is pretranslated, this tool cannot be easily extended to allow HSL program text to be viewed/modified.

7. OBSERVATIONS

HSL is a new, process-oriented language designed to serve the needs of both modeler and simulation programmer. It seems conducive to the way modelers conceptualize systems and it adheres to principles important to software engineers. It is especially well suited for hierarchical refinement and compression of its processes or process branches. That is, it supports heterogeneous representations, or levels of modeled details, in the various parts of a model depending on which are deemed to be most critical or unpredictable.

In addition, being interpreter-based, HSL not only provides portability with its machine independence; it also affords the flexibility of easier access to an executing simulation program, to check values of variables and to make selected parameter changes. Further, since the language is not yet locked into a compiler, it remains more malleable for any desired changes in the language itself.

Finally, above-indicated design features of HSL have enhanced the potential of providing effective, interactive support by means of an interface system, such as N-CHIME. We are presently extending that prototype interface to include other important capabilities, such as adaptation to different kinds of HSL users.

APPENDIX A: EBNF NOTATION AND HSL ENVIRONMENT GRAMMAR

EBNF Syntax Notation

The syntax of an HSL module is described by means of a context-free grammar. The notation for expressing the grammar is an extension of the Backus-Naur Form and is similar to that used to specify the Ada language [9]. The notation consists of metalinguistic variables, metalinguistic symbols, HSL reserved words and HSL symbols. The grammar itself is presented as a

sequence of rules, where each rule consists of a left-hand side (LHS) and a right-hand side (RHS) connected by the meta-symbol ::= . The left-hand side of the rule is a metavariable. The right-hand side of the rule is an expression consisting of metavariables, metasymbols, HSL reserved words and HSL symbols. Wherever a metavariable appears in the right-hand side of a rule, the right-hand side of any rule having that variable as its left-hand side can be substituted for the variable. This is the distinguishing characteristic of context-free grammars.

A metavariable is a character string consisting of letters and underscores (_). The variable name is chosen to be meaningful in its context. The metasymbols to be used are the following:

::=	Connects LHS and RHS of a rule.
|	Separates alternative RHS items, as an "or" operator.
[ ]	Encloses an optional RHS item.
{ }	Encloses a repeated RHS item that can appear 0 or more times.

HSL reserved words and functions are distinguished by their **boldface** type. All other symbols are HSL symbols (which are also emboldened). Unfortunately, the |, [and] symbols are used both as metasymbols and as HSL symbols. In this case, the HSL symbols are enclosed in single quotes ('|', '[', ']'), which are not valid language symbols.

The grammar is not perfect; certain context-sensitive characteristics cannot be expressed. Examples are the following: type-checking both for assignment and for matching of formal and actual parameters, prohibition of a **return** statement in a process and prohibition of an **exit** statement or any simulation statement in a function. A small degree of rigor in grammar presentation has been sacrificed for improved readability.

Some of the metavariables referenced in the RHS of rules are defined in the HSL Simulator grammar. Any metavariable name which ends in 'identifier' stands for a user-defined identifier consisting of a string of letters, digits and underscores (_). Any metavariable name which ends in 'integer' stands for a string of digits.

HSL Environment Grammar

```
environment              ::= model model_identifier
                               {global_declarations}
                             model_control_statements
                           end model_identifier;
global-declarations      ::= base_declaration
                           | abstract_declaration
                           | entity_class_declaration
base-declaration         ::= [constant] base_type var_list;
base_type                ::= int | real | bool | str
var_list                 ::= var_spec {, var_spec}
var_spec                 ::= identifier [array_spec] [:= initial_value]
array_spec               ::= '['dimension {, dimension} ']'
dimension                ::= lower_bound_integer..upper_bound_integer
abstract_declaration     ::= stat abs_list : stat_mode;
                           | queue [capacity] abs_list : queue_mode;
                           | resource [capacity] abs_list : resource_mode;
```

```
capacity                ::= '[' integer ']'
abs_list                ::= abs_spec {, abs_spec}
abs_spec                ::= identifier [array_spec]
stat_mode               ::= dep | ind
queue_mode              ::= fifo | lifo | prio | siro
resource_mode           ::= fcfs | ffit | preempt
entity_class_declaration ::= [parent_class_identifier] entity class_identi-
                             fier:
                               {attribute_declarations}
                               end class_identifier;
attribute_declarations  ::= base_attr_declaration
                          | abstract_declaration
base_attr_declaration   ::= [constant] base_type base_attr_list;
base_attr_list          ::= base_attr_spec {, base_attr_spec}
base_attr_spec          ::= identifier [array_spec]
model_control_statements ::= model_control_statement {model_control_
                             statement}
model_control_statement ::= start exog_process {, exog_process};
                          | stop boolean_expression;
                          | trace boolean_expression;
                          | report boolean_expression;
                          | report boolean/expression
                            {report_clause}
                            end report;
exog_process            ::= process_identifier ([actual_parm_list])
actual_parm_list        ::= expression {, expression}
report_clause           ::= queue '|' boolean_expression;
                          | entity '|' boolean_expression;
                          | process '|' boolean_expression;
                          | resource '|' boolean_expression;
                          | stat '|' boolean_expression;
```

APPENDIX B: HSL SIMULATOR GRAMMAR

The syntax of the HSL Simulator module is presented as a context-free grammar using an extended Backus-Naur Form (EBNF). An explanation of the EBNF is found in Appendix A, along with the HSL Environment grammar. Again, as stated there, a small degree of rigor in grammar presentation has been sacrificed for improved readability. In particular, the grammar makes no distinctions of operator precedence (which is context free) because this vastly simplifies the expression rules. Some of the metavariables referenced in the RHS of rules are defined in the HSL Environment grammar, as presented in Appendix A.

HSL Simulator Grammar

```
simulator               ::= {process_definition | function_definition}
process_definition      ::= process process_identifier ([process_parm_list])
                              [accepts class_identifier entity_identifier]
                              {base_declaration | entity_declaration}
                              {general_statement | simulation_statement}
                            end process_identifier;
function_definition     ::= function function_identifier ([func_parm_list]) re-
                            turns base_type
                              {base_declaration}
                              {general_statement}
                            end function_identifier;
```

```
process_parm_list       ::= process_parm {;process_parm}
process_parm            ::= base_type identifier {,identifier}: process_parm_
                            mode
process_parm_mode       ::= in | out | inout
func_parm_list          ::= func_parm {;func_parm}
func_parm               ::= base_type identifier {,identifier}
entity_declaration      ::= class_identifier entity_spec {,entity_spec};
entity_spec             ::= entity_identifier [array_spec]
statement_sequence      ::= statement {statement}
statement               ::= general_statement | simulation_statement
general_statement       ::= variable := expression;
                          | read variable {,variable};
                          | write expresion {,expression};
                          | if boolean_expression
                              then statement_sequence
                              [else statement_sequence]
                              end if;
                          | case expression of
                                {choice_list '|' statement_sequence}
                                [default statement_sequence]
                              end case;
                          | loop
                              statement_sequence
                              end loop;
                          | break; | next; | exit; | return expression;
simulation_statement    ::= schedule process_identifier ([actual_parm_list])
                            with entity_reference
                              [in expression];
                          | call process_identifier ([actual_parm_list]) with
                            entity_reference
                              {,process_identifier ([actual_parm_list]) with en-
                            tity_reference};
                          | delay expression;
                          | suspend (queue_reference);
                          | awaken (queue_reference);
                          | entity_reference.create;
                          | entity_reference.destroy;
                          | entity_reference.setpriority (expression);
                          | stat_reference.collect (expression);
                          | queue_reference.enqueue (entity_reference);
                          | queue_reference.chqcap (expression);
                          | resource_reference.request (expression);
                          | resource_reference.release (expression);
                          | resource_reference.chrcap (expression);
expression              ::= identifier
                          | integer
                          | real_number
                          | "string_of_characters"
                          | (expression)
                          | expression arithmetic_operator expression
                          | function_identifier ([actual_parm_list])
                          | random_variate_function
choice_list             ::= choice {,choice}
choice                  ::= integer
                          | lower_bound_integer .. upper_bound_integer
real_number             ::= integer . integer
```

```
arithmetic_operator     ::= + | - |*| / | ^ | %
boolean_expression      ::= identifier
                          | true
                          | false
                          | ( boolean_expression )
                          | not boolean_expression
                          | boolean_expression and boolean_expression
                          | boolean_expression or boolean_expression
                          | expression relational_operator expression
relational_operator     ::= = | # | < | > | <= | >=
variable                ::= identifier [ array_reference ]
                          | entity_reference . attribute
                          | stat_reference . attribute
                          | queue_reference . attribute
                          | resource_reference . attribute
                          | process_identifier . stat_reference
                          | entity_class_identifier . stat_reference
attribute               ::= variable
entity_reference        ::= entity_identifier [ array_reference ]
stat_reference          ::= stat_identifier [ array_reference ]
queue_reference         ::= queue_identifier [ array_reference ]
resource_reference      ::= resource_identifier [ array_reference ]
array_reference         ::= '[' expression {,expression } ']'
random_variate_function ::= rand01 ( stream )
                          | randint (stream , low , high )
                          | uniform ( stream , low , high )      Note: These
                          | expo ( stream , mean )               parameters are
                          | normal ( stream , mean , stdev )     expressions.
                          | hyperex ( stream , mean , stdev )
                          | erlang ( stream , mean , stdev )
                          | setseed ( stream , newvalue )
```

ACKNOWLEDGMENT

Special thanks are due to Larry Rose who was a principal investigator on this project during the HSL design stage, until he left the University of Pittsburgh in mid-1988. We also appreciate greatly the support provided by NCR, Cambridge, OH. The conscientious help and guidance from Tom Donnelly and the early encouragement for the HSL-style design from Don Finfrock were invaluable assets to our group.

REFERENCES

1. Banks, J., and Carson, J. S. Process-interaction simulation languages. *Simulation 44*, 5 (May 1985), 225-235.
2. Bezivin, J. Timelock: A concurrent simulation technique and its descripton in smalltalk-80. In *Proceedings of the 1987 Winter Simulation Conference* (Atlanta GA, Dec. 14-16, 1987), A. Thesen, H. Grant, W. D. Kelton, Eds., pp. 503-506.
3. Birtwhistle, G. M., Dahl, O. J., Myhrhaug, B., and Nygaard, K. *SIMULA begin*. Petrocelli/Charter, New York, 1975.
4. Bohm, C., and Jacopini, G. Flow diagrams, turning machines, and languages with only two formulation rules. *Commun. ACM 9*, 5 (May 1966), 366-371.
5. Buxton, J. N., Ed. Simulation programming languages. In *Proceedings of the IFIP Working Conference on Simulation Programming Languages* (Oslo, Norway, May 1987). North-Holland, New York, 1968.

6. FISHMAN, G. S. *Principles of Discrete Event Simulation.* Wiley, New York, 1978.
7. FORD, D. R , AND SCHROER, B. J. An expert manufacturing system. *Simulation 48*, 5 (1987), 193-200.
8. FRANTA, W R. *The Process View of Simulation.* North-Holland, New York, 1977.
9 GAHANI, N *Ada, An Advanced Introduction: Reference Manual for the Ada Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1984.
10. GOLDEN, D. G. Software engineering considerations for the design of simulation languages, *Simulation 45*, 4 (Oct. 1985), 169-177
11. GORDON, G. *System Simulation, 2nd Ed.* Prentice-Hall, Englewood Cliffs, NJ, 1978.
12. JOHNSON, S. C. Yacc: Yet another compiler-compiler. CSTR 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
13. KETCHAM, M. G., SHANNON, R. E., AND HOGG, G. L. Information structures for simulation modeling of manufacturing systems. *Simulation 52*, 2 (Feb 1989), 59-67.
14. KIVIAT, P. J., MARKOWITZ, H. M , AND VILLANUEVA, R. *SIMSCRIPT II.5 Programming Language.* CACI, Los Angeles, 1983.
15. KREUTZER, W *System Simulation Programming Styles and Languages.* Addison-Wesley, Reading, MA., 1986.
16. LESK, M. E., AND SCHMIDT, E. Lex—A lexical analyzer generator, CSTR 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975
17. MACLENNAN, B. J. *Principles of Programming Languages: Design, Evaluation, and Implementation, Second Edition.* Holt, Rinehart and Winston, New York, 1987
18 OVERSTREET, C. M., AND NANCE, R. E A specification language to assist in analysis of discrete event simulation models. *Commun. ACM 28* 2 (Feb.1985), 190-201.
19. PEGDEN, C D. Introduction to SIMAN In *Proceedings of the 1986 Winter Simulation Conference* (Washington, DC, Dec. 8-10 1986), J. Wilson, J. Henriksen, S. Roberts, Eds., pp. 95-103.
20. PRATT, T W. *Programming Languages: Design and Implementations, Second Edition* Prentice-Hall, Englewood Cliffs, NJ, 1984.
21. PRITSKER, A. A. B. *Introduction to Simulation and SLAM II.* Systems Publishing Corp., West Lafayette, IN, 1986
22. ROSE, L L. Modeling and simulation of computer systems. Tech. Rep. 86-8, Dept. of Computer Science, Univ. of Pittsburgh, 1986.
23 ROSE, L. L , SANDERSON, D P , SHARMA, R , AND ROZIN, R. Syntax of a programming language for process-oriented discrete systems simulation. Tech. Rep. 87-6, Dept. of Computer Science, Univ. of Pittsburgh, 1987.
24. ROSE, L L., SANDERSON, D. P , SHARMA, R., AND ROZIN, R. Semantics of a programming language for process-oriented discrete systems simulation. Tech. Rep. 87-7, Dept. of Computer Science, Univ. of Pittsburgh, 1987.
25 ROTHENBERG, J. Object-oriented simulation: Where do we go from here?, In *Proceedings of the 1986 Winter Simulation Conference* (Washington, DC, Dec. 8-10, 1986), J. Wilson, J. Henriksen, S. Roberts, Eds., pp. 464-469.
26. ROZIN, R., AND TREU, S. A hybrid implementation of a process-oriented programming language for system simulation *Softw Pract Exper. 21*, 2 (June 1991), 557-579.
27. SANDERSON, D. P., AND SHARMA, R.. User guide to HSL and its prototype interpreter. Tech. Rep 88-8, Dept. of Computer Science, Univ. of Pittsburgh, 1988.
28 SANDERSON, D P., TREU, S , SHARMA, R., AND ROZIN, R. Simulation language design based on modeler-conducive structural features. In *Modeling and Simulation. 20*, 3, Part 3: Computers, Computer Architectures and Networks (W. G Vogt and M. N. Mickle, Eds., In *Proceedings of 20th Annual Pittsburgh Conference on Modeling and Simulation* (Pittsburgh, PA, May 4-5, 1989). pp. 1295-1300.
29. SANDERSON, D P., ROZIN, R., AND TREU, S N-CHIME user's guide. Tech. Rep. 90-1, Dept. of Computer Science, Univ. of Pittsburgh, 1990.
30. SANDERSON, D. P., AND TREU, S. Designing the intelligent component of a user interface for modeling and simulation. In *Proceedings of the Simulation Multiconference on Artificial Intelligence and Simulation.* (New Orleans LA, April 1-5, 1991), R. Uttamsingh and A. M. Wildberger, Eds., pp. 47-52.

31. SCHRIBER, T. J. *Simulation Using GPSS*. Wiley, New York, 1974.
32. SCHRIBER, T. J. *An Introduction to Simulation Using GPSS/H*. Wiley, New York, 1990.
33. SCHWETMAN, H. CSIM: A C-based, process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*. (Washington, DC, Dec. 8-10, 1986), J. Wilson, J. Henriksen, S. Roberts, Eds., pp. 387-396.
34. SHEPPARD, S. Applying software engineering to simulation. *Simulation 40*, 1 (Jan. 1983), 13-19.
35. SNYDER, A. Encapsulation and inheritance in object-oriented programming. In *Proceedings of the 1986 Conference on Object-Oriented Programming Systems, Languages and Applications* (Portland, OR, Sept. 29-Oct. 2, 1986), N. Meyrowitz, Ed., pp. 38-45.
36. STROUSTRUP, B. *The C++ Programming Language*. Addison-Wesley, Reading MA, 1986.
37. TREU, S. Designing a 'Cognizant Interface' between the user and the simulation software. *Simulation 51*, 6 (Dec. 1988), 227-234.
38. TREU, S., SANDERSON, D. P., ROZIN, R., AND SHARMA, R. High-level, three-pronged design methodology for the N-CHIME interface system software. *Inf. Softw. Technol. 33*, 5 (June 1991), 306-320.
39. UNGER, B. W. Programming languages for computer system simulation *Simulation 30*, 4 (1978), 101-110.
40. WEGNER, P. Dimensions of object-based language design. In *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages and Applications* (Orlando FL, Oct. 4-8, 1987), N. Meyrowitz, Ed., pp. 168-182.
41. ZEIGLER, B. P. *Theory of Modelling and Simulation*. Wiley, New York, 1976.
42. ZEIGLER, B. P. Systems hierarchy as a basis for simulation model description. *SIMULETTER 15*, 1 (March 1984), 8-13.