SOFTWARE—PRACTICE AND EXPERIENCE. VOL. 21(6). 557–579 (JUNE 1991)

A Hybrid Implementation of a Process-oriented Programming Language or System Simulation

ROMAN ROZIN AND SIEGFRIED TREU
Department of Computer Science, 324 Alumni Hall, University of.Pittsburgh, Pittsburgh, PA 15260, U.S.A.

SUMMARY

The implementation of a hierarchical, process-oriented programming language for simulation (HSL) is described. It features a hybrid approach, involving the front end of a compiler and the back end of an interpreter. An HSL program is dichotomous in structure. Source statements from each part are translated into three-address code for an abstract machine, and the resulting code is then interpreted. The algorithms and the supportive data structures that effect the translation and interpretation of HSL are detailed. The host language for HSL is C++. HSL is machine independent and can be ported to any machine on which the host language is available. Its initial implementation was carried out on an NCR Tower. More recently, it was transferred to an NCR PC916.

KEY WORDS Simulation programming language Design Implementation Compiler Interpreter C++

INTRODUCTION

Frequently used methods for implementing new programming languages [1-5] tend to be very general and apply to practically all languages, whether imperative or functional in nature. The design criteria for a particular language of course influence the choice of implementation method. [6,7] One approach is to write a full interpreter to carry out the meanings of individual source statements. But, that is usually quite slow. In general, no simple mapping exists between the statements and operations of the target language and those of the source language. Also, the translator has to perform all the functions of lexical and syntactic analysis as it is translating. It must do so every time it passes over a statement. Looping is very slow in such implementations.

Another method is to write a full compiler, either in another high-level language or in the target language itself. If it is written in the target language, one usually implements in it a subset of the full language, and hand translates the compiler into assembly language, assembles the resulting program, and writes the compiler for the full language, using the subset of the language for which the compiler exists. The advantage of a full compiler is the speed with which the resulting programs run. But, there are several disadvantages. First, because it generates machine code, it fixes the machine on which the compiled programs can run. Thus, it makes the language non-portable. Secondly, it fixes the definition of the target language. Ideally, the definition of a language should be somewhat malleable during its initial usage and experimentation. Writing a compiler that generates decent machine code,

0038–0644/91/060557–23$11.50

Received 7 August 1989
Revised 13 December 1990

with efficient register allocation,[1] is also an order of magnitude more work.

Yet another method for implementing a new language combines the advantages and, to a certain extent, also the disadvantages of the compiler with those of the interpreter. This hybrid approach uses the front end of a compiler and the back end of the interpreter. It can also just be called an interpreter.[8] Thus, the term is used in two senses: one standing for the complete language processor of programs, and the other for a specific part of it, namely the back end. The hybrid implementation option was chosen for the Hierarchical Simulation Language, or HSL.[9,10]

HSL is designed to support process-oriented simulation. As outlined more fully in the next section, it includes the well-known simulation primitives such as start, create, schedule, call, delay, suspend, awaken and stop. The simulated system activities are modelled by HSL program modules called processes. The HSL interpreter works as follows: HSL program statements, including those that constitute processes modelling the system activities and time delays, are translated into intermediate code, called *quads*. These quads are grouped into process-specific modules in an appropriate data structure, the *intermediate code table* (IC table). That IC table is then the basis for interpretation and for simulation of the modelled system.

How does the HSL interpreter compare with previous interpreter-based implementation efforts? Among the many interpreted languages are Lisp[11] and Icon,[12] which support list and string processing, respectively. Algol-like languages, such as Pascal-P, have also been implemented in an interpretive manner. However, the Pascal-P compiler generates code for a hypothetical stack machine, which is subsequently run by the stack computer simulator.[13,14] The reason for doing that is program portability. Because the Pascal-P compiler is itself compiled into the stack computer code, one only needs to write the stack computer simulator on a new computer, in order to port the Pascal-P system to it.

The HSL implementation offers portability also, but it differs from the other languages in an important respect, as suggested above: its code is generated for a virtual, quad-based machine, including many high-level operations essential to simulation. These operations have no parallels in computer architecture and require an extensive run-time system. Furthermore, because much time must be spent in that run-time system in any case, to interpret HSL is much *less inefficient* than it is to interpret a language such as Pascal. The loss of efficiency between interpreter and compiler is therefore not nearly as great as is normally expected. Thus, HSL not only reflects the above-mentioned advantages, namely portability and malleability, but also a reasonably efficient, interpreter-based implementation that is distinct from implementations of other interpreted languages.

LANGUAGE OVERVIEW

This paper is mainly intended for implementers, *not* so much users, of programming languages. Detailed coverage of HSL, from a programmer's standpoint, is therefore not included. However, enough information is presented about the nature of the language to give justification to the implementation techniques that were chosen.

This is done by means of a fairly simple, generic system model. The reader is assumed to be knowledgeable in language design as well as, to a reasonable extent, in modelling and simulation methodology. The example is portrayed in a fourfold manner: (1) a brief description of the system to be model led, (2) a graphic version of the model, depicted by Figure 1, according to the dichotomous HSL structure, (3) the corresponding two-part HSL program, listed in Figure 2, and (4) a characterization of the kinds of language statements contained in each of HSL'S two modules.

With reference to Figure 1, the modelled system involves two major processes, ProcA and ProcB. These encapsulate the programmed activity sequences that simulate the behaviors of corresponding system resources, ResA and Res B. The resources could represent two different computer systems, two different stations in a manufacturing plant, or two different offices in a business organization. The major processes have hierarchically subservient, lower-level processes (e.g. ProcA1 and A1 a for ProcA), programmed to simulate the detailed (or more refined) behaviors of component resources (e.g. ResA1 and ResA1a. The latter could be the CPU and memory subsystems, in the case of a computer systems model.

The above-indicated set of (endogenous) processes, contained within the SIMULATOR module of the model, must have some entities (e.g. transactions for computer systems; or parts for manufacturing plants) to act upon. The example model is assumed to have two types of entities: the Trans1 entity class, defined in the ENVIRONMENT, with entities created by the (exogenous) process labelled Gen_Transl, in the SIMULATOR; likewise, members of the Trans2 entity class are generated by the Gen_Trans2 process.

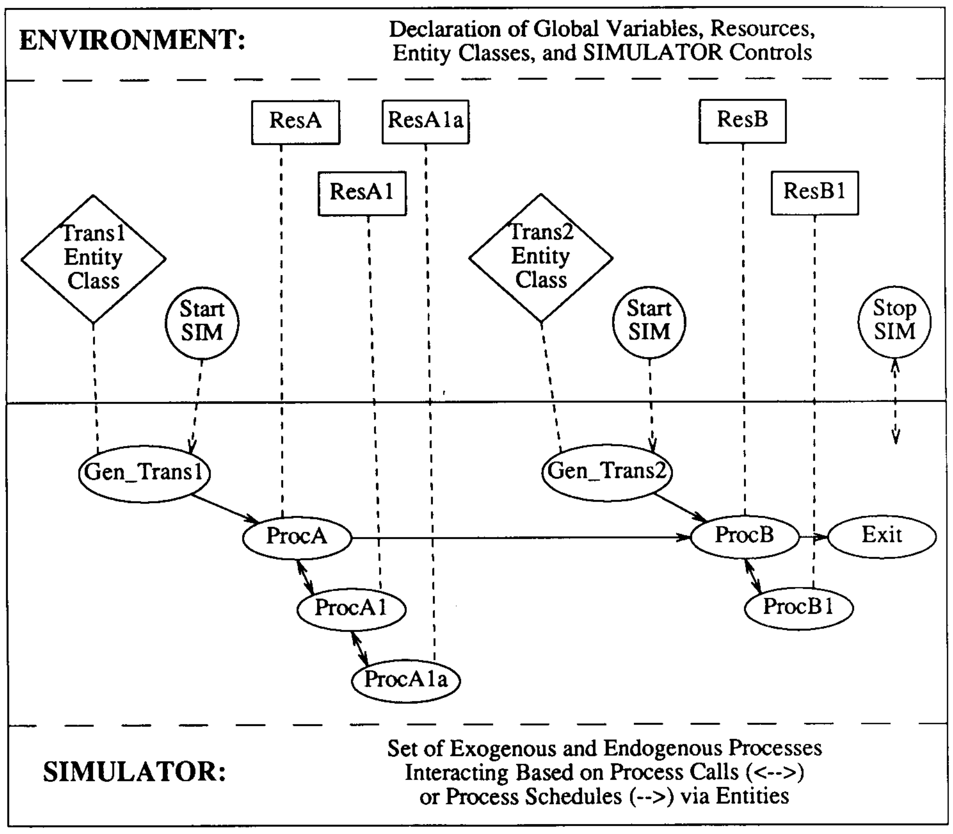
ENVIRONMENT:
Declaration of Global Variables, Resources, Entity Classes, and SIMULATOR Controls
ResA
ResA1a
ResB
ResA1
ResB1
Trans1 Entity Class
Trans2 Entity Class
Start SIM
Start SIM
Stop SIM
Gen_Trans1
Gen_Trans2
ProcA
ProcB
Exit
ProcA1
ProcB1
ProcA1a
SIMULATOR:
Set of Exogenous and Endogenous Processes Interacting Based on Process Calls (<-->) or Process Schedules (-->) via Entities

Figure 1. A generic model illustrating the HSL dichotomy

```
!HSL   ENVIRONMENT

model generic

      boolean stop_condition;

      resource ResA: fcfs;
      resource ResA1: fcfs;
      resource ResA1a : fcfs;
      resource ResB : fcfs;
      resource ResB 1: fcfs;

      entity Trans1:
            int Tl_field;
      end Transl;

      Trans 1 entity Trans2 :
            int T2_field;
      end Trans2;

      start Gen_Transl ();
      start Gen_Trans2();

      stop stop_condition = true;

end generic;
```

```
!HSL   SIMULATOR

process Gen_Transl()
       Transl  tl;
       loop
          delay expo(2,10,0);
          t1 .create()
          schedule ProcA() with tl;
       end loop;
end Gen_Transl;

process ProcA() accepts Trans1 t;
   call ProcAl() with t;
   ResA. request (l);
   delay normal(l,5.0,1.5);
   ResA. release (l);
   schedule ProcB() with t;
end ProcA1;

process ProcAla () accepts Trans1 tt;
   ResAla. request (l);
   delay normal(2,6.5,2.4);
   ResAl. release (l);
   call ProcAla() with tt;
end ProcAla;

process ProcAla() accepts Transl tt;
   ResAla. request (l);
   delay normal(3,7.0,2.5);
   ResAla. release (l);
end ProcA1a;

process Gen_Trans2()
       Trans2t2;
       loop
          delay expo(5, 7.0);
          t2.create();
          schedule ProcB() with t2 in 1 mins;
       end loop;
end Gen_Trans1;

process ProcB() accepts Trans2 tt;
   if tt. pop >100 then
       stop_condition:=  true;
   end if;
   call ProcB1() with tt;
end  ProcB;

process ProcB1() accepts Trans2 tt;
   ResB1  request (l);
   delay 2 mins;
   ResB1. release (l);
end ProcBl;
```

Figure 2. Dichotomous HSL program

It is evident that an HSL model, consistent with the nature of the modelling and simulation application, has a dichotomous structure: the SIMULATOR represents the dynamic system behavior to be simulated by the process-oriented simulation module, and the ENVIRONMENT contains the required global variables and constructs as well as the logic to control the SIMULATOR'S actions. This dichotomy is reflected significantly by the HSL implementation, as discussed later. It is also very apparent in the example program listed in Figure 2, corresponding to the generic model of

Figure 1. HSL programming statements [15] can be characterized according to the following outline of sets. Statements are designed:

1. For the ENVIRONMENT,
   - (a) to configure the modelled system or the system characteristics that remain static during a particular simulation run. Included are data structures in the global scope of an HSL program, e.g. required resources, entity types, statistics and queues pertaining to the modelled system.
   - (b) to enable control over the actual execution of the simulator module, namely the start and stop simulation statements as well as the specifications of conditions under which tracing information and reports are to be produced.
   - (c) to support the above by means of declarations of appropriate global constants and global variables, both simple and abstract. Basic data types are integer, real, boolean and string. Abstract data types, each with a full range of attributes and functions to implement semantic actions, are statistics, queues and resources.
   - (d) to provide for abstract data-typing capability using the entity class. With these the attributes can be represented for each member of a class of transaction objects flowing through a modelled system. Each entity attribute may be of any of the above-mentioned data types, except of an entity class itself. An entity class may only be declared, not also defined, in the environment; definition must take place in the simulator section. Entity attributes may be inherited.

2. For the SIMULATOR,
   - (a) to define processes that describe an entity's activities in the modelled system, thereby forming the basis of model execution for simulation. A process contains a name, a formal parameter list (including number, type, mode of argument), an optional entity parameter, and a sequence of statements. If the optional entity parameter is not specified, the process is exogenous and can only be initiated by the start statement in the environment; otherwise, the process can be activated in the simulator, with reference to the named entity.
   - (b) to define functions, each including a name, a formal parameter list, the function's type, and a sequence of statements. Functions are available for computational ease only. They are not illustrated in the example program of Figure 2.
   - (c) to declare local variables and constants within processes and functions. Local constants must be of a simple type; local variables must be of a simple type or an entity type.
   - (d) to provide processes and functions with each of the following: simple instructions (e.g. assignment, read, write); typical arithmetic, logical and relational operations; and several control constructs to aid the programmer in simulation-oriented features (e.g. schedule, call, delay, suspend/awaken) and in the simulator program itself (e.g. if-then-else-endif; if-case; loop).

Most of the above-outlined sets of HSL statements are illustrated, in simple versions, in the generic program of Figure 2. Noteworthy is the fact that processes, in the SIMULATOR module, can interact with each other in two different ways: (1) hierarchically, using the process call (e.g. ProcA calling ProcA1), with expected return of control from the lower-level process to the suspended process that originated the call, and (2) laterally, using the process schedule (e.g. ProcA scheduling ProcB), with asynchronous 'passing' of an entity to the scheduled process and no return to the originating process, which does not suspend execution. The choice between these two statements, in any particular instance, depends on the type of hierarchical (dependent) or parallel (independent) behaviour that is to be simulated in a system.

IMPLEMENTATION PHASES

The two candidates considered for HSL'S host language were Modula-2 [16] and C+ +. [17] After extensive experimentation with the two languages, C++ was chosen. The reasons for this decision included [18] that (1) it was, at the time, a front end translator to C, making it more compatible with UNIX* and enabling use of the Lex [19] and Yacc [20] tools with it, (2) in the tests conducted, C++ ran faster, and (3) it was available on a wider choice of machines.

The major components of HSL and their relationships were implemented as depicted in Figures 3 and 4. They can be divided roughly into two phases: the front end, which is composed of two parts, and the back end. The front end extends from the beginning through the sim parser. It translates the HSL source into intermediate code (IC). The back end is the interpreter, which interprets the intermediate code. This is a key feature of the HSL implementation. The intermediate code is based on three-address statements called quads. [8] Each quad consists of an operator and up to three operands. The last one is usually the destination. The sched operator, which schedules a process to run in the future, is the only one that has four operands, thus requiring two quads to be specified; the second quad merely holds the last operand. In general, except for the branch operators, whose destination operand is the index into the quad array where the locus of control should be transferred, operands are variables, temporaries, indirect temporaries, or constants.

As much as possible is encoded into the intermediate code instructions. This is to avoid imposing further work on the interpreter, thereby enhancing its execution speed. Depending on the kind of HSL program statement or module involved, the quads are then organized into one of several different IC tables (Figure 3): (1) the condition IC table for simulation control, (2) the start IC table for starting exogenous processes, and (3) the IC table containing the groupings of quads representing the simulation processes and functions.

The symbol and type tables (Figure 3) interact with all the major front-end components. Although available during the interpreter's execution, they are at present not used by it extensively. However, they can be used to give run-time error diagnostics by the interpreter and also to allow the user to modify the global environment during a simulation.

Between the sim parser and the interpreter are the disk files produced by the sim parser and the env parser of the front end. Once these are available, the back end can be run repeatedly with them, without invoking the front end. This separation is important in HSL and is explained further below.

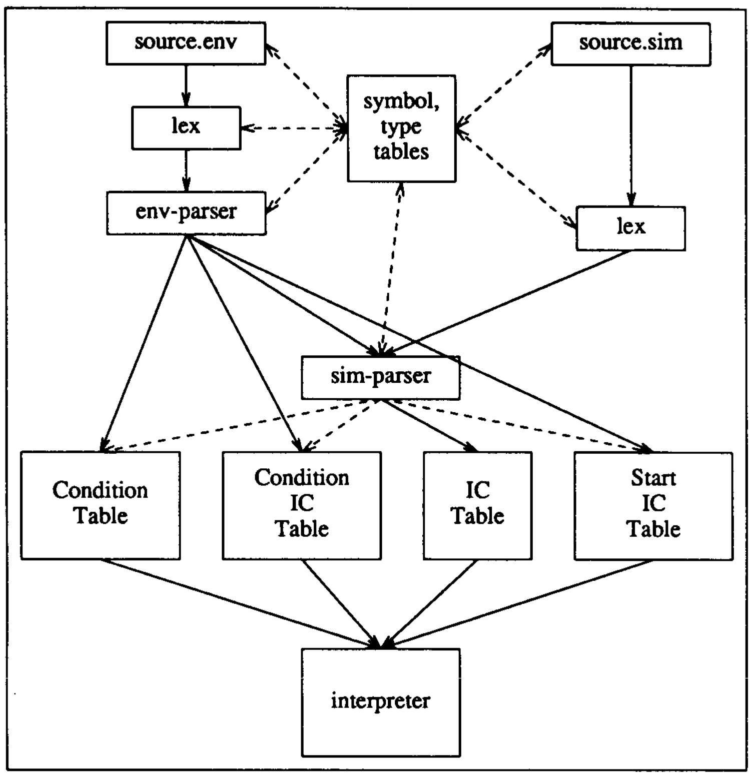

Figure 3. Major HSL components

FRONT-END PHASE

As already suggested, the front end must be subdivided into two parts (Figure 4) in order to process the two distinct modules of an HSL program. The environment part parses the .env source files with the env parser, and the simulation part parses the .sim source files with the sim parser. These two parts and the interpreter communicate through disk files.

Lexical analyser

Even though there are two separate parsers (see below) for an HSL model, only one lexical analyser is needed. This is because no discrepancies exist in the structure of the lexemes between the environment and simulation parts. The lexical analyser is built using *Lex* —a lexical analyser generator.[19] It enables specification of the patterns, using regular expressions, to be recognized in the input. Specific actions can be attached to these patterns.

The two main ways for recognizing reserved words in a language are (1) using a pattern to describe each word, with an action returning an indication of the token, and (2) recognizing only identifiers, and then indexing the identifier into a reserved word table. If the identifier is found, an indication of the kind of token it is is returned. Otherwise, the returned ID signifies that it is not a reserved word. The

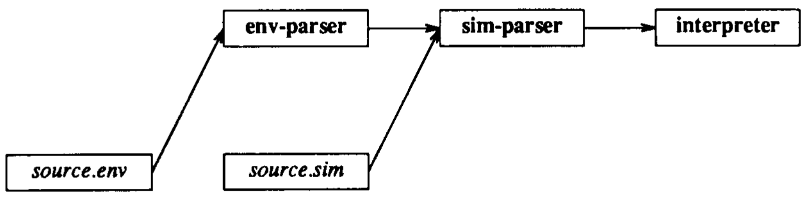

Figure 4. Front end to back end relationship

first approach was chosen for HSL because of its simplicity and speed, although the latter claim is debatable. The resulting lexical analyser is bigger, but the search through the reserved word table is avoided. Either way, the difference in speed is not significant, and the simplicity argument lends the most weight.

Two parsers

Two separate parsers were implemented, one for the environment and one for the simulation part of an HSL program. Even though this causes some overlap, especially in parsing expressions, the replication of code seems insignificant in view of the advantages gained. Principles of software engineering, mainly modularity and simplicity, provided motivation for this HSL design decision. Each separate parser module is smaller and simpler than would be a monolithic version. In addition, the operational interaction with an HSL-programmed model makes it possible to use one parser but not the other, under certain conditions. For example, if the simulation part only is to be modified, because environmental specifications remain the same, then only the sim parser is run, not also the env parser.

The two parsers are constructed using *Yacc*, a parser generator, [20] which builds an LALR(l) parser. [1] Yacc input consists of the grammar rules with embedded actions, to be executed when a particular point in the reduction of a rule is reached. Because AT&T's C++ was used, the Yacc output is run through the C++ preprocessor.

Data structures and disk files

The env parser must communicate information to the sim parser, and the sim parser must do likewise to the interpreter. In addition to the above-indicated advantages of having the two parsers, this scheme enables the interpreter to be run several times without having to re-run the front end at all. Information is passed from one phase to another in disk files. To avoid complicated encoding and decoding of data structures as they are written out to disk files and read back in, a simple block write/read of a chunk of memory containing the data structure can be performed. However, this means that true memory pointers cannot be used, since there is no guarantee that one phase occupies the same virtual address space as the previous one.

The decision to use disk files to communicate between the env parser and the sim parser, and between the latter and the interpreter (Figure 4), is for reasons of efficiency. This may not be obvious. However, the high-level dichotomy of a simulation model and the way simulation runs are typically carried out suggest that, if a modeller wants to change the simulator section, there should be no need to recompile the environment section. Also, if the model is to be run several times, there should be no need to recompile either part.

In addition, the use of disk files was influenced by the fact that the HSL implementation had to be ported from the NCR Tower, running under Unix, to an NCR PC916, under NCR-DOS. DOS implements pipes by running programs in a pipeline sequentially. Because of this, any attempt to extract the minimal speed advantage from communicating among the parsers and the interpreter through the pipeline mechanism would have been foiled. In a situation in which disk I/O is truly undesirable and to be avoided, it would be possible to use a RAM disk, thereby in effect doing I/O at memory access speeds.

All the data structures that conceptually use pointers in the HSL implementation instead use indices into arrays. However, if the designs of the data structures and the algorithms that manipulate them are clean, it becomes irrelevant whether true pointers or indices are used. In fact, the symbol table algorithms and data structures were intially written using pointers, and then, in an almost completely mechanical fashion, they were changed to indices. Most of the changes involved something like the following:

```
symtab_entry*next  ⇒ int next
```

The only tricky part came with the allocation of dynamic memory. Since deletions are not performed in a symbol table, there is no concern about re-using allocated space. This makes for very simple memory allocation design. The solution was to pre-allocate a large array and to keep an index to its beginning. Every time a new element is requested, the index is incremented and its previous value is returned.

Environment parser

The environment parser translates .env files. As indicated in Figure 3, it generates symbol and type tables and several condition and IC tables. After parsing, all the data structures set up are saved in disk files, to be read by the sim parser.

Global symbol table

This table contains identifiers and their associated attributes. The attributes of identifiers include their *kind,* i.e. name of a variable, constant or process; for a variable, its type and offset; for a constant, its literal value; for a process, its attribute pointer. Each identifier is stored in a record.

Several reasonable choices [21,22] of a data structure for the symbol table are available. A hash table with open hashing [1] was chosen because of its efficiency and simplicity. Therefore, in case of collision, a symbol table entry must also contain a field holding the index of the symbol table entry to which it chains, as demonstrated in Figure 5. The ELEM table indicated in that Figure holds the symbol table's entries for variables and constants as well as temporaries and unnamed constants. It is also used with the IC tables, as shown later.

Variables, which can be global or local, need addresses or offsets: for globals, offsets from the beginning of the global space; for locals, offsets from the beginning of the current activation record. The information indicating whether a variable is

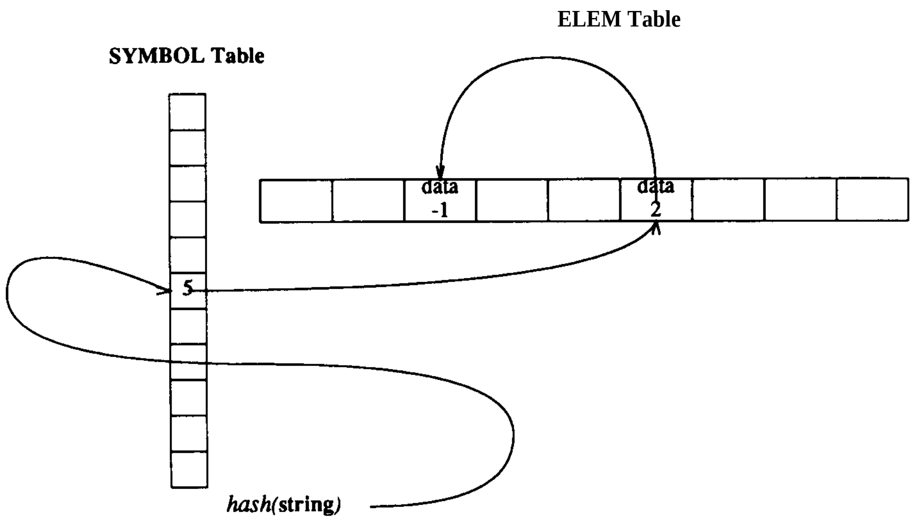

Figure 5. Symbol table using indices

local or global is encoded, by positive and negative signs, in its *offset* field in the symbol table. For identifiers such as variables, it is necessary to have their types. The well-known notion of setting up a separate data structure, namely the *type table,* was used. The symbol table entries have indexes into the type table.

Type table

Each element of this array is a variant record describing a particular type. In C terminology, each element is a union of several structures, with a tag field to specify the active union member. Thus, in general, any complex type that is itself composed of several other types has its description completed not by replicating those composite type descriptions several times, but by having pointers to the type table. If ttable is the name of the type table, then ttable[0] may be an array of reals, ttable[1] an integer, etc. (Figure 6).

An entity type is described in the type table as follows: the first entry contains the entity type, the entity class name, an index into the type table describing its first field, and an index into the type table describing the type of its super class, if any; a field has its name, an index into the type table describing its type, and an index to the next field, if any. The next-field index of the last field of an entity has an index to the entry representing the list of predefine entity attributes (eventually terminated by – 1), if that entity has no super class, or, if it does, an index to the entry describing the first field of its super class (Figure 7). Type table entries for fundamental types are not needed. They are described by specific integers not indexing into the type table.

Condition control tables

The model control actions (stop, trace and report) to be taken during the running of a simulation, are based on the truth values of certain expressions. Any expression

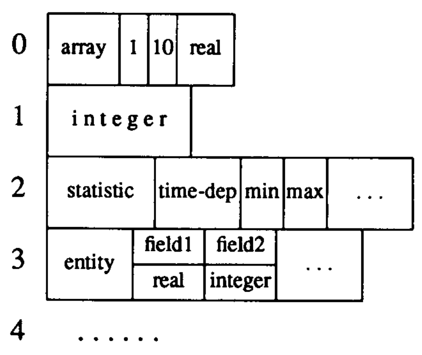

Figure 6. Type table

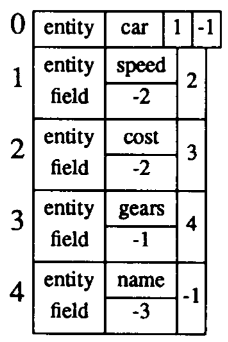

Figure 7. Entity type description: entity car: real speed, cost; int gears; str name; end car;

that evaluates to a boolean value can be used. For example, stop | = 10; stops the simulation when the global variable | is equal to 10. The env parser builds two tables to handle the conditions, the *condition table* and the *condition IC table* (Figure 3). It is also concerned with creation of the *start IC table* and the initialization of globals.

Condition table

This table has entries for each of the following: stop, trace, report and the report subcategories report-stat, report-entity, report-queue, report-resource and report-process. The default condition is to report everything. Since there can be multiple expressions for each condition, a count is maintained of the number of expressions in each entry, and an array of indices identifying the beginning of an expression in the condition IC table (Figure 8), described next. The conditions are checked every time control returns to the simulation executive from the interpreter. When there are multiple expressions for a condition, they are evaluated in a logical-or manner, until it is determined that the disjunction of all the expressions is true or false.

Condition IC table

The expressions specified in control statements are quads that are inserted, as they are generated, into successive locations of the condition IC table (Figure 8).

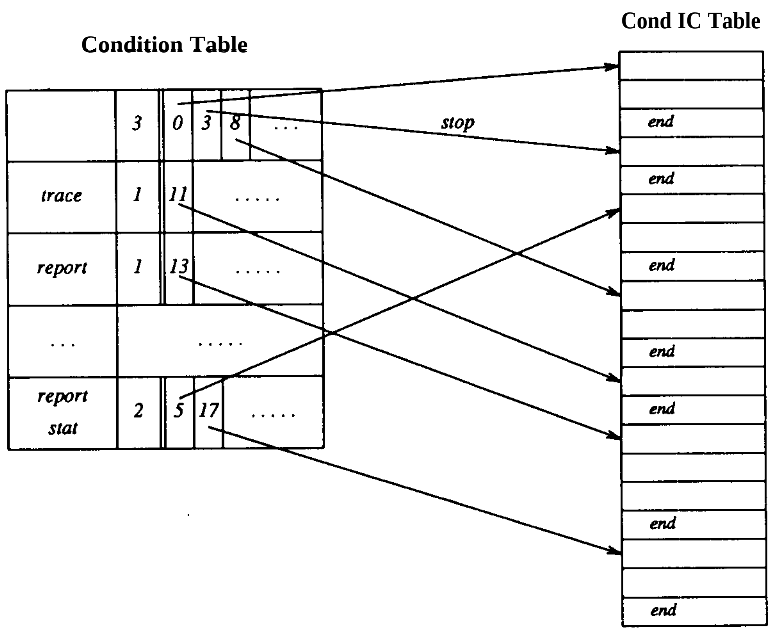

Figure 8. Condition table and condition IC table

To separate expressions, a special opcode is inserted into the table after a complete expression has been parsed.

Start IC table

For a simulation to run, one or more exogenous processes must be specified in the start statements (see Figures 1 and 2). The process specified in a start statement is placed on the *future events queue* (feq) by the interpreter at the beginning of the simulation. The env parser translates the process invocations from the start statement into quads that are placed in the start IC table. For every process in the start statement, its name is placed in the symbol table, to be verified as valid later by the sim parser, and an index to it is returned. Then a start quad is generated. The quad contains the opcode quad, the index to the process name in the symbol table, and the number of parameters passed to the process at invocation. The parameters passed can be arbitrary expressions. Quads are generated to compute them. Eventually a quad to pass the parameter is generated. It consists of the arg opcode and the result of the computation of the expression. For instance, Figure 9 shows, the contents of the start IC table after parsing the statement: start CAR_PROCESS(10, 73), in conjunction with the ELEM table.

Initialization of globals

One way to handle initialization is to place the information into the symbol table entry for the variable initialized, and then replace, where possible, the uses of the variable with the value directly (constant propagation). The other way is to generate

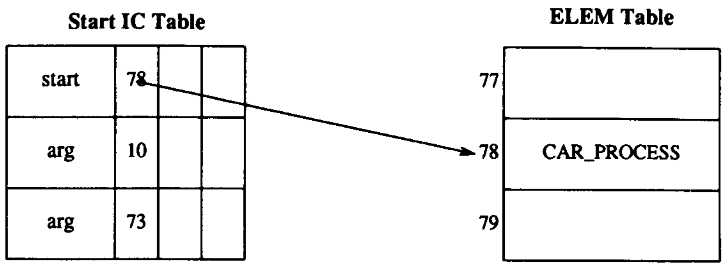

Figure 9. Start statement implementation

code to assign the values to the variables during execution. The former way is rather complicated, with significant overhead, and not very useful unless various optimizing transformations are performed on the code. A few assignment statements do not significantly affect the execution speed. The second way is straightforward and is the one chosen for HSL. The only question was where to place the initialization code. The simplest solution is in the start IC table. Statements in it are executed before the simulation begins running.

Simulation parser

The sim parser recreates all the data structures setup by the env parser, by reading them in from disk. It then uses them to perform its main purpose, which is the translation of process and function definitions into intermediate code. When it is done, it saves all it has created in disk files, to be used by the back end.

IC table

This stores the intermediate code generated for HSL processes and functions. It contains the quads produced by the sim parser for execution during the simulation itself. It is, therefore, separate from the start IC table, which is created by the env parser for purposes of triggering the start-up of specific, exogenous processes from the outside, i.e. the environment.

For the intermediate code, a consistent form of operand addressing facilitates both the semantic routines of the parser and the functioning of the interpreter. Therefore, an operand is generally an index into the ELEM table (Figure 10), which was already mentioned in conjunction with Figure 5. An operand is not an index into the symtab table, but directly into the ELEM table. Two levels of indirection in the interpreter, where the speed is more important, would be wasteful. In any case, in our implementation it is impossible, because several ELEM entries may be chained from a single symtab entry. A constant declared in an HSL program is placed with its value into the symbol table. Constants used explicitly are entered into the ELEM table. In this way, whether an operand is a variable or a constant, only its index is manipulated in the quad array. Many of the semantic functions handle any kind of operand. Simplification of the code is thus considerable, especially in conjunction with the semantic stack provided by yacc, and the actions that can have values, as previously described. Even though there is some overhead in the indirection required to access

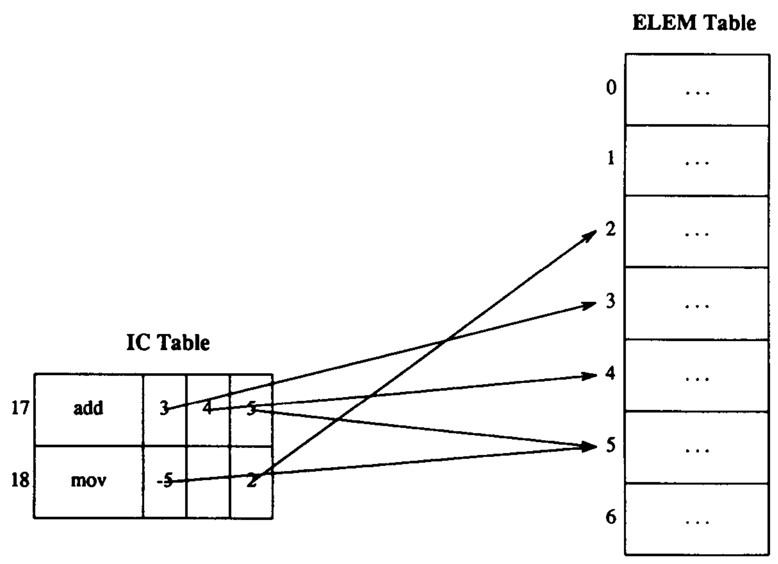

Figure 10. The IC table

a constant in the ELEM table, as opposed to having it literally in the quad as an operand, the simplification in the encoding of the quads more than outweighs the overhead.

Indirect temporaries, which are usually generated by computing array references, are indicated as such in the quads by negative numbers. For example, –5 is an indirect temporary, the entry for which is at ELEM [5].

Mode determination

Since the delineation of tasks of the front and back ends is somewhat arbitrary, there are options for where a particular task can be accomplished. As stated earlier, towards making sure that the back end (interpreter) runs as fast as possible, we relegated as many tasks as possible to the translation process and quads generation of the front end. This was also done for the operand mode determination.

An operand can be of several kinds: local, global, constant. For each of these options, it can be direct or indirect. Thus, with each quad it is convenient for the interpreter to have that information about its operand, to avoid computing it. Likewise, a separate but related task is to resolve the types of arithmetic operands. The opcode can indicate that both operands are of real type, to relieve the interpreter of additional computation. A separate phase of the front end, a filter, handles the above tasks. It reads the tables generated by the front end proper, performs its determinations, and writes the modified tables back to disk.

Local symbol tables

In addition to the global symbol table visible to all processes and functions, each process and function can declare local variables. This necessitates the creation of a local symbol table for each. Since the number of processes and functions defined is not known ahead of time, it would be unreasonable and wasteful to place some arbitrary limit on them. Only the space for the global symbol table is allocated from the outset. When a function or process definition is encountered during parsing, a new symbol table is allocated. But, it is allocated to be contiguous to the global symbol table, to enable the same functions to refer to the local and global symbol tables. It is done using the Unix realloc function. The variable Lsymtab (local symbol table), which is passed to the symbol table manager routines, is just an integer, indexing into the symbol table area at the point where the new symbol table was allocated.

BACK-END PHASE

The back end consists of the simulation executive, the interpreter proper, and the various library functions that implement simulation primitives. The simulation executive and the interpreter are related in a coroutine-like manner.

Simulation executive

The fundamental functions of the simulation executive are to manage the future events queue (feq), to update the clock, to check the model control statements (stop, trace and report), and to inform the interpreter where to continue execution. The high-level algorithm of the simulation executive is outlined in Figure 11.

As seen from that Figure, the model control statements are checked once every time through the simulation executive loop. If a condition becomes true during the interpreter's execution, not immediately before control returns to the simulation executive from the interpreter, then action is taken only when control is returned to the simulation executive. Thus, there are two related problems: (1) action is not taken immediately when a condition is true, (2) the condition may become true, and then become false again, all before control returns to the executive. This means that action that should have been taken, is not taken. Any solution to these problems involves testing of the model control condition after every intermediate code instruction. This is clearly not a feasible alternative, given the goal of fast model execution. However, although these problems are theoretically possible, such pathological cases occur quite rarely. In fact, in over a year of experimentation, it has not happened once. Therefore, in view of its rarity, the effort to guard against it seemed prohibitive and was not expended.

```
while (not feq.empty and not stop) {
   get next event notice
   update clock
   check and respond to model trace and report
   resume process execution
   /* return here when blocked */
}
```

Figure 11. Simulation executive algorithm

The back end starts off by reading in all the tables generated by the env and sim parsers. It then sets up the global. data area. The amount of storage needed for all global variables is known to the sim parser and passed to the executive. It also initializes the free list for the activation records, sets up the initial feq, by calling the interpreter with the start IC table as an argument, and calls the simulation executive.

Event notices and the future events queue

An *event notice is* a structure containing important information pertaining to a simulation event. Some of the fields it has are the time of notice, the resumption point in the IC table (where to continue execution), a pointer to its stack of activation records, the entity associated with an event, and the priority of the entity. When a new event is to be posted, a new event notice is allocated, if there are no free ones. Initially no free event notices are available, since they are created by discarding existing ones. When an event notice is to be discarded, it is placed on the free list of event notices, to be re-used later on. When an event notice is posted, a pointer to it is inserted into the feq (Figure 12).

The feq is a priority queue, implemented with the heap data structure. It contains pointers to event notices, to make insertion and deletion of elements as efficient as possible. No copying of large structures is needed. The simulation executive, every time through its main loop, picks the next event from the feq, using the only deletion operation allowed on it —deletemin. The elements in the feq are inserted based on two keys—primary, time, and secondary, priority. The smaller the time of the event notice, the closer to the front of the queue the event notice is inserted. If two event notices have the same time, then their priorities are examined, and the one with the higher priority goes first. Equivalent priorities impose an arbitrary order on the event notices.

Activation of processes and functions

The opcode call is used for process invocation, and the opcode invoke is used for function calls. There are three ways in which a process may come to life. It can be picked off the future events queue, it can be call ed by another process, or it can be start ed by the start statement from the env section of the model. A function may be called by a process or by another function including itself. Functions and process may have parameters and local variables, which are unique to a particular invocation. Parameters can be arbitrary expressions. A call or invoke is generated first, followed by quads to compute expressions, if any, followed by actual parameter-passing quads. If process *A* invokes function *f*, with parameter *x,* and process *B* invokes the same function *f* with parameter y, before *f* returns to *A,* it is not only desirable but mandatory that the data calculated on behalf of process *A* not be corrupted by invocation of the function from *B.* Since it is also possible that another process C call s process *A*, it is likewise desirable that the two instances of process *A* not confuse or mix their data. Therefore, each particular instance of a process or function executes in its own environment. This environment is an activation record, containing the parameters and local variables, among other important data, of an active instance of a process or function (Figure 13). The *dynamic link* points to the invoking

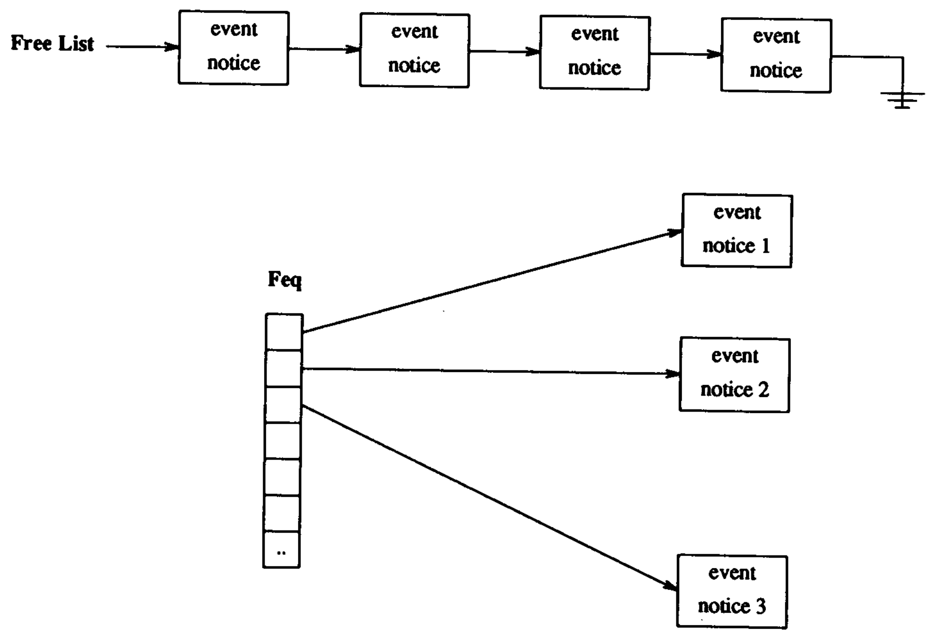

Figure 12. Feq *and the free list*

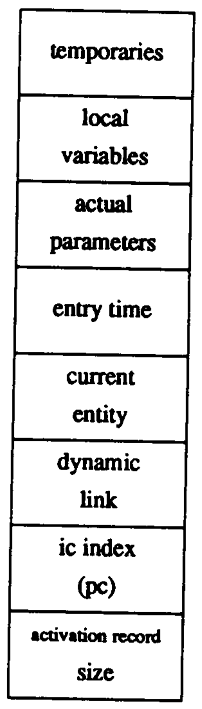

Figure 13. Process activation record

activation record (if any); the *ic index* holds the index into the IC table where execution should continue following the return from the process or function.

No static link exists to access the variable at the correct level. Only languages which allow nesting of function (procedure, etc.) definitions require these. In HSL, variables are either local or global to a process or function; there is no third alternative. However, HSL does not have dynamic scoping, in which the variable is sought in the call chain, if not found locally. The dynamic link is merely used to return to the calling activation record when the current invocation of the process or function is finished. Thus, when two instances of process *A* are created, as in the above example, each gets an activation record. If a local variable *x* in the first instance of *A* has the value 5, and, in the second instance, it is assigned the value 10, the value in the first instance remains intact. The situation is similar for function calls. Each instantiation of a function gets it own activation record, to preserve the integrity of its data.

Memory management

Memory is initially allocated as a large array. Each memory element is a union of types, since it can hold an integer, a real, a string and a pointer to a C++ object. The bottom portion of memory is allocated to the global variables, and a variable bp points to the top of the global space. Because global variables have negative offsets, a global variable is accessed as MEMORY[bp + var_offset].

When the simulation executive initially sets up the free list of memory, everything but the memory occupied by the globals is placed in one large block. The size of the block is placed in the memory word right above the global space. The word immediately above it is used to point to the next free block. Thus, initially it is – 1. When an activation record is allocated, the free list of memory is searched sequentially using a first-fit strategy. The activation record is placed there, and the size and next variables are adjusted accordingly. The first-fit strategy was chosen because of its simplicity. Other methods [23,24] have some advantages but are complicated and were viewed as giving insufficient return in efficiency for the complexity of coding them. Figure 14 shows the memory after several activation records have been allocated and deallocated in it. The first entry after global space has the value 0, meaning that there is no free space in this block. An activation record is present there. The next entry has the index into the memory of the following block, etc.

Thus, there is no true stack or heap; some activation records are linked together, and some are not. Conceptually, with several processes running and calling functions, each process has its own chain of activation records, and there is a separate global area. The chains of activation records are not related. New chains are started when a process schedule s another, but not when a process call s another. In the latter case, the original chain grows.

Most activation records require memory allocation for temporaries, since temporaries are used in expression evaluation. Temporaries are re-used at the source statement level. In other words, for each process or function, the largest number of temporaries any particular source statement when translated requires is determined, and that is the number allocated in the activation record. For the environment part, the largest number of temporaries required by any of the expressions is found, and that number is allocated at the end of the global data area.

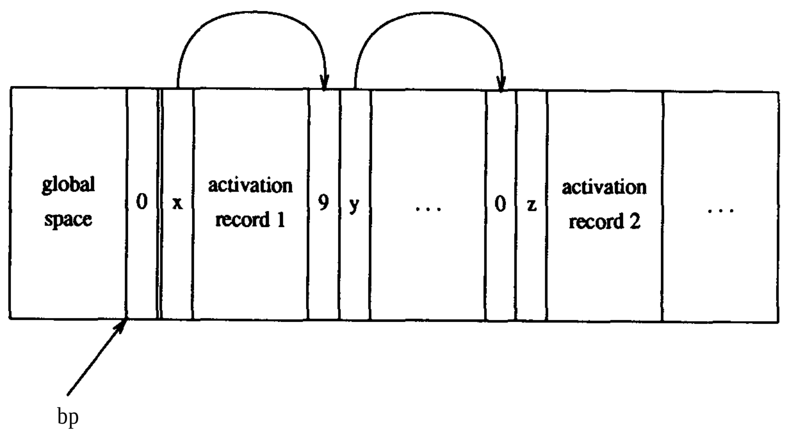

Figure 14. State of memory block

Interpreter

The interpreter executes the meanings of the intermediate code statements, present in a quad table passed to it as one of the parameters. It interprets code from several different quad tables and returns the current quad index, a boolean value. But, the meaning differs depending on which quad table is being processed.

One of the functions of the interpreter is to initialize the feq, by successively processing the quads from the start IC table, until all the start statements are processed. The start opcode is handled just like the schedule, with default settings. The return value signals the end of start IC table. The interpreter also initializes global variables, if any, by processing the start IC table. When the interpreter processes the condition table, it returns either TRUE or FALSE, based on the value of the condition it evaluates. When the interpreter processes the IC table generated by the sim parser, its return value can be ignored, unless the return is from a recursive call to itself. The heart of the interpreter is an infinite loop (Figure 15), which makes decisions based on the opcode, performing computations, moving results in memory, setting up activation records, calling library functions.

Arithmetic, boolean and branch opcodes

The interpreter has a function val that takes an operand and a mode and returns the address of its operand. Thus, if the opcode is MOV, and pc holds the pointer to the current quad, the following occurs:

```
case MOV:
  *val(pc–>op3,pc–>mode3) = *val(pc–>op1,pc–>mode1);
  pc++;
  break;
```

All the arithmetic and boolean operations are performed in a manner similar to the MOV above. Branches, unconditional, branch-on-true, and branch-on-false, just reset the value of the pc variable.

```
for (;;) {
    switch (opcode) {
        case ADD: get first operand;
                get second operand;
                get address of destination;
                add both operands,
                move result into destination;
        case MOV: ..;
        . . .
        case LIBFUNC: /* library function */
            get destination address;
            switch (function) {
                ...
                case UNIFORM:
                    call interpreter recursively
                        on each parameter computation, and
                        save each result;
                    call uniform with
                        computed parameter
                    assign result to destination;
            }
        case CALL:...;
    }
}
```

Figure 15. Main loop of interpreter (for (;;) *signifies an infinite loop)*

Input/output

HSL can read integers, reals, strings and boolean values. Because conversions between reals and integers are permitted, even in input, the most elegant way is to write a lexical analyser that recognizes either a string, a real or an integer. Then it compares the type of string recognized with what is expected and, if necessary, performs the conversions internally. Only if conversion is impossible (e.g. a real number is expected and the next token in the input is a string, or a string is expected and the next token in the input is an integer, or a boolean is expected and the next token is anything else), an error condition is indicated. The lexical analyser for the input routines, just like the lexical analyser for the parser of HSL, is written using Lex.[19]

The output is handled using the standard C library output function printf. Printf is called with a format command based on the type of the variable to be printed out. Since calls to read and write take nearly arbitrary expressions as parameters (nearly arbitrary because a constant expression to a read is illegal, for instance), they can be considered to be function calls. Implementation of I/O involves recursive calls to the interpreter, as described below.

Statistics collection

There are two types of statistics: predefine and user-defined. For predefine statistics, an explicit COLLECT is necessary to obtain its rein, max, sum, sumsq, ohs, prevtm, etc. Sumsq is sum of squares, obs is number of observations, and prevtm is the previous time. These are collected for time dependent statistics only. The predefine statistical attributes of queues, resources and entities are collected automatically as the objects are used. When an entity is enqueued, for instance, the queue length statistic is updated. Mean and standard deviation can be accessed by correspondingly named function calls, and all statistical information, if needed, can be displayed by means of the report facility.

Type coercion

The conversions go from integer to real and from real to integer. In those cases in which it is impossible for the front end to effect a type conversion, such as when an assignment is between two variables whose values are unknown, an explicit opcode is used, cnvtri or cnvtir, with the obvious meaning.

Library functions

The interpreter is called recursively to evaluate the arguments to the library functions, and then the appropriate library function is called. No activation record or anything usually associated with function call implementation need be set up, because of the available compiled support functions written in C++.

Function invocations and process calls

The first thing done upon function invocation is the setting up of a new activation record (Figure 16). All of its fields, as described above, are given appropriate values, and then the interpreter is called recursively to evaluate the actual parameters. The main advantage of the recursive evaluation of argument quads is the resulting simplicity of the code. The variables that must be modified, such as the current quad index and the pc pointer, are global and thus are correctly updated by the recursive calls. Since an interpreter for the quads already exists, and since function arguments can contain within themselves other function calls, the scheme is described naturally

```
case INVOKE :
    create and setup new activation record,
    ic_index++;
    for (i = 0; i < number_of_ args; i++){
        ic_index = interpreter(ic, ic_index, curntt, ar);
        /*pass val params into activation record */
        m.mem[ new_ar + 5 + i ] = * val( pc->op1, pc->m1 );
        ic_index++;
    }
    save return IC address;
    point to new activation record;
    jump to beginning of function;
```

Figure 16. Function call implementation

by the recursion. The front end places a special quad after every function argument, informing the interpreter to return. This opcode can be encountered by the interpreter only if it is invoked recursively. A process call is similar, except for the extra code concerned with entities accepted by a process, and the collection of process statistics.

Delay and schedule

The similarity between delay and schedule is that both cause a process to be activated at some time in the future. Thus, for both of them an *event notice* is created and set up, and then inserted into the feq. However, for schedule, several more things have to be done before an event notice can be created and inserted into the future events queue. A new activation record has to be created and set up; statistics such as *entry time* have to be collected; and arguments to the process have to be evaluated and copied into the new activation record.

CONCLUSIONS AND PLANS

In designing and implementing HSL, the project team faced numerous design choices and attempted to render decisions based on specific design principles and objectives. As happens in practice, these were at times less than mutually consistent. For example, in the interest of enhancing simulator execution speed, type checking was delegated to the front end. On the other hand, in spite of the overhead it produces, but owing to the concern for simplicity, recursive interpretation of function and process arguments was favoured over a non-recursive alternative.

After HSL was implemented on an NCR Tower, at the end of the 1988 summer, it was evaluated in its application to the simulation of a particular system model .[25] The language worked very well on the test model. Its major desirable features were found to be its modularity (consistent with well-known software engineering principles), its hierarchy and ‘object’ orientation (corresponding to the processes, entities and resources that are so prevalent in discrete system simulation), and its flexibility (facilitating the modification and refinement of the programmed simulation model). HSL was also determined to be very conducive to enabling modellers to represent their systems in a manner that seems natural to the way they think. [26]

In early 1989, HSL was ported to a personal computer, an NCR PC916. Besides continuing improvements of HSL, current research involves the design of an interactive and intelligent modelling environment called N-CHIME . [27] It is expected to provide an attractive, well-structured interface between various kinds of users and the HSL software itself.

ACKNOWLEDGEMENTS

The implementation of HSL was critically dependent on two other graduate students in computer science: Pete Sanderson and Ravi Sharma. They and Roman Rozin did all the programming on this project. Appreciation is also due to Dr. Larry Rose who left the University of Pittsburgh in mid-1988. He was the principal driver behind the original HSL design.

This research was supported in part by NCR Corporation, Cambridge, OH, U.S.A.

REFERENCES

1. A. V. Aho, R. Sethi and J. D. Unman, *Compilers: Principles, Techniques, and Tools,* Addison Wesley 1986.
2. C. N. Fischer and R. J. LeBlanc, Jr., *Crafting a Compiler,* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1988.
3. D. Barron (ed.), *Pascal, The Language and Its Implementation,* Wiley, New York, 1981.
4. N. Wirth, Design and implementation of Modula', *Software—Practice and Experience,* **7,** 67–84 (1977).
5. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation,* Addison Wesley, Reading, Massachusetts, 1983.
6. N. Wirth, 'On the design of programming languages', *Proc. IFIP Congress 74, 1974,* pp. 386–393.
7. C. A. R. Hoare, 'Hints on programming language design', in E. Horowitz (ed.), *Programming Languages, A Grand Tour,* 3rd edn., Computer Science Press, Rockville, MD, 1987, pp. 31–40.
8. M. Marcotty and H. F. Ledgard, *Programming Language Landscape,* second edn, Science Research Associates, Sydney, Toronto, 1986.
9. R. Rozin, D. P. Sanderson and R. Sharma, 'Design of the interpreter for the hierarchical simulation language', *Technical Report 88–6,* Department of Computer Science, University of Pittsburgh, 1988.
10. R. Rozin, 'Implementation of a programming language for process-oriented discrete systems simulation', M.S. *Thesis,* University of Pittsburgh, 1988.
11. P. H. Winston, *LISP,* 3rd edn, Addison-Wesley, 1989.
12. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language,* Princeton University Press, 1986.
13. K. V. Nori, U. Ammann, H. H. Nageli and C. Jacobi, 'Pascal-P implementation notes', in D. W. Barron (ed.), *Pascal—The Language and Its Implementation, Wiley,* 1982, pp. 125–170.
14. J. Welsh and A. Hay, *A Model Implementation of Standard Pascal,* Prentice-Hall, 1986.
15. D. P. Sanderson and R. Sharma, 'User guide to HSL and its prototype interpreter', *Technical Report 88–8,* Department of Computer Science, University of Pittsburgh, 1988.
16. N. Wirth, *Programming in Modula 2,* 4th edn, Springer Verlag, Berlin, 1988.
17. B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1986.
18. D. P. Sanderson and R. Sharma, 'C++ and Modula-2: a comparison toward support for discrete systems simulation', submitted for publication, 1989.
19. M. E. Lesk, 'Lex—a lexical analyzer generator', *CSTR 39,* Bell Labs, Murray Hill, New Jersey, 1975.
20. S. C. Johnson, 'YACC—yet another compiler compiler', CSTR 32, Bell Labs, Murray Hill, New Jersey, 1975.
21. T. A. Standish, *Data Structure Techniques,* Addison-Wesley, 1980.
22. A. V. Aho, J. E. Hopcroft and J. D. Unman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley Publishing Company, 1974.
23. D. E. Knuth, *The Art of Computer Programming, Vol. 1,* second edn, Addison-Wesley Publishing Company, 1973.
24. A. V. Aho, J. E. Hopcroft and J. D. Unman, *Data Structures and Algorithms,* Addison-Wesley Publishing Company, 1983.
25. S. Treu, D. P. Sanderson, R. Sharma and R. Rozin, 'Testing and evaluation of HSL: modularity, flexibility. and hierarchical orientation', *Technical Report 89–2,* Department of Computer Science, University of Pittsburgh, 1989.
26. D. P. Sanderson, S. Treu, R. Sharma and R. Rozin, 'Simulation language design based on modeler-conducive structural features', *Modeling and Simulation,* **20** (3): in W. G. Vogt and M. H. Mickle (eds), *Computers, Computer Architectures and Networks,* Instrument Society of America, Research Triangle Park, NC, 1989, pp. 1295–1300.
27. S. Treu, D. P. Sanderson, R. Rozin and R. Sharma, 'High-level, three-pronged design methodology for the N-CHIME interface system software', *Information and Software Technology,* to appear, 1991.

*UNIX is a trademark of Bell Laboratories.