CONSIM: a study of control issues in conversational simulation

Sallie S. Nelson

Department of Computing Science, Texas A & M University, College Station, Texas 77843, USA

CONSIM is a prototype conversational simulation language which provides simulation types of control and allows mid-execution editing of both programs and data. In its development CONSIM served as a vehicle for exploring the control issues inherent in the design, implementation, and use of such a programming system.

SIMULA 67 was used in the development of the prototype system both as a model for CONSIM's control features and as its implementation language. SIMULA was selected primarily because of its high level control facilities, including both coroutine and simulation sequencing. The dual use of SIMULA in this project allowed an implementation strategy called interpretive control self-modelling to be used.

This paper provides an overview of the CONSIM development by discussing the motivation of the research, highlighting CONSIM's interesting and unique features and illustrating its utility with excerpts from a model-building scenario. The prototype implementation is described and, finally, the results and promising areas for future research are summarised.

This work was supported in part by the National Science Foundation under grant DCR73-03441 A01 to the University of Pittsburgh.

(Received January 1978)

1. Conversational simulation

1.1 *Background*

In higher level programming languages the control related features, which deal with overall program logic and the management of run time environments, can be distinguished from the data related features. These control related facilities can be further categorised as either control simple (e.g. aspects such as operators, expressions, conditionals, and iterations) or control rich (e.g. aspects such as blocks, procedures, and coroutines).

A three year study of control definition in programming languages has recently been completed at the University of Pittsburgh (Lindstrom, 1977a). Our goal in this research was to seek ways to refine, formalise and appraise control as an avenue for systematic language design. This study has provided a better understanding of and insight into the control aspects of higher level languages which we hope will be of interest and benefit not only to language users but also to other language researchers, designers, and implementers. Some of the results of this study have been reported in Lemon, Lindstrom and Soffa (1977), Lindstrom (1977b), Lindstrom and Nelson (1976), Nelson (1978), Nelson and Lindstrom (1977), Soffa and Lindstrom (1976; 1977), Strauss (1976). This paper concentrates on one aspect of this research, namely the development of an experimental conversational simulation language called CONSIM (Nelson, 1977).

1.2 *Research value of CONSIM*

The development of a conversational simulation language contributed to our research in control definition in several ways. First, in designing CONSIM two control rich language types (i.e. conversational and simulation) were merged. To meet our definition of conversational, CONSIM had to support mid-execution editability of both programs and data, as well as provide an interactive user/system interface. To be classified as a simulation language CONSIM had to provide traditional facilities such as pseudoparallel processing, scheduling, and queue handling. Supporting the control richness of conversational and simulation processing simultaneously required new approaches and techniques. CONSIM, therefore, served as a vehicle for exploring the problems and possible solutions encountered in designing and implementing a control rich environment.

Because CONSIM explores a programming environment of use to simulation programmers and model builders, its development is of more interest than just as an academic exercise in control design and implementation. Conversational simulation has been proposed and advocated by several authors (Kiviat, 1974; Lindstrom, 1973) as a means of providing more powerful and useful language tools for simulation applications. Although designs for such languages have been proposed (see for example Jones, 1963), we know of no implementations which meet our definition of conversational simulation. CONSIM's implementation, therefore, serves as a vehicle for establishing the feasibility and utility of this new language environment.

1.3 *Control issues*

The CONSIM research concentrated on the control aspects involved in the design and development of this high level language and its required support systems. Issues such as the following were considered:

(*a*) control statements in the conversational simulation language

(*b*) user control over operation of the system (e.g. model initiation, interruption, and termination)

(*c*) inclusion of the user as a control entity within the model

(*d*) control aspects of the implementation of the above features and

(*e*) styles of programming possible in the control enriched environment provided by the conversational simulation language.

Each of these areas is discussed in more detail later. Section 2 considers some of CONSIM's unique or interesting design features while Section 3 illustrates the utility of CONSIM's control richness, especially in model building. Section 4 provides an overview of the prototype implementation. The last section summarises our conclusions from the CONSIM study and suggests additional research based on this work.

2. CONSIM's design

The philosophy followed in CONSIM's design was to maximise

user flexibility by minimising user restrictions. Because the CONSIM implementation was intended to be an experimental system for language study, an attempt was made to avoid *a priori* decisions as to what might or might not be useful and desirable from the user's perspective. In the prototype implementation the user is assumed to know what he or she is doing; processing continues at the user's direction as long as possible, even overriding CONSIM error warnings. Operation of the resulting system, therefore, requires that the user have some knowledge of the internal organisation of the system.

2.1 *SIMULA as CONSIM'S syntax model*

The syntax and simulation facilities of CONSIM are patterned after those of SIMULA 67 (Dahl, Myhthang and Nygaard, 1968). In addition to the facilities of ALGOL 60, SIMULA offers the class and coroutine concepts, reference variables, simulation primitives, and extensive text and input/output capabilities. SIMULA was selected over other candidate languages (notably SIMSCRIPT and GASP) primarily because of its high level control facilities for both coroutine and simulation sequencing. This approach of modelling CONSIM after an existing simulation language offered several advantages over designing a totally 'new' language. First, it shortened the design phase by allowing us to take full advantage of work already done by SIMULA's designers. Since SIMULA 67 is itself a 'second generation' simulation language (benefiting in its design from users' experiences with its ancestor SIMULA I), it provided a particularly stable and well tested basis for CONSIM.

Secondly, fashioning the conversational language after an existing compilation oriented language ensures that CONSIM has as its core a regime which is known to be suitable for later compilation. Thus, although CONSIM is implemented via an interpreter, stabilised programs are readily translatable into SIMULA for maximum production efficiency. Because CONSIM is a prototype language, only a subset of the facilities which would be desirable for a full user oriented implementation have been included. Features which relate to the control issues enumerated above as well as other representative simulation facilities were selected for implementation in CONSIM. A BNF specification of CONSIM is given in Appendix 1.

2.2 *Uniform language*

Language elements are needed in the conversational system which enable the user to direct the operation and processing of programs (e.g. **run, stop, edit,** etc.). Such elements may be separated into a distinct language, as in the job control language of batch environments. Alternatively these directives, which we call commands, may be integrated with the language statements used in expressing the algorithmic steps of the program, thus forming a single uniform language.

In the conversational environment the uniform language concept greatly simplifies system operation for the user since he or she is not required to distinguish between the command and statement sublanguages. He or she does not have to keep track of the currently valid sublanguage and the sublanguage membership of each element: all commands and statements are acceptable at any time. Because CONSIM was designed as a uniform language, its commands and statements belong to a common language as can be seen by the BNF specification given in the appendix.

2.3 *Commands and increments*

Special instructions called 'commands' have been included in CONSIM to aid the user in directing the construction, testing, and execution of programs. **Table 1** shows some of these and briefly describes their meaning.

Table 1 CONSIM user commands

Function	*Command*	*Meaning*
Entry	NEW-TASK	Set up a new task to be associated with user supplied identifier.
	COMPILE	'Compile' disc source code saved from previous session into post syntactic form for interpretation.
	SOURCE	Accept and compile new source statements into post syntactic form for later interpretation.
Edit	SHOW	Display entire CONSIM source program.
	SHOW *n*	Display increment *n* of source program.
	FIX\old\new\	(Must be preceded by command SHOW *n*). Search increment *n* for 'old' text and if found replace by 'new' text. Post syntactic representation of edited text replaces previous version.
	DELETE *n*	Delete increment *n* from both source and post syntactic form files.
Execution	IMMEDIATE	Take the given CONSIM source and execute it in the immediate mode. Source and post syntactic forms are not saved.
	RUN	Begin execution (i.e. interpretation) of CONSIM program.
	HALT	Halt processing of CONSIM program.
	CONTINUE	Continue processing of program from point of interruption.
	CONTINUE *n*	Continue processing of CONSIM program from the beginning of increment *n*.
Termination	FINISHED	Terminate this task.
	STOP	Terminate this interactive session.
Status	STATUS	Display a status report of current system activity.

The reader will note that several of these commands make reference to 'increments' (e.g. **source, show, fix, delete,** and **continue**) An increment is a program fragment capable of being accepted by CONSIM's syntax analyser. It consists of a string of program elements (tokens) ending with a token from the distinguished set of terminal symbols {"**end**", ";"}. Additional restrictions require that no more than one unmatched **begin** or **end** may occur within a single increment. (These restrictions could be relaxed by increasing the size of CONSIM's defining grammar but were considered acceptable in the prototype implementation).

The notion of increment in CONSIM provides a means of referencing partial programs, thus enhancing communication between the user and the system. The user supplies an identifying number with each increment at source entry. Because syntax analysis occurs as the code is entered or edited, the user is notified and allowed to correct any syntax errors im-

mediately, rather than after all code for an entire program has been entered (as with most compiler systems). The syntax analyser produces an independent module containing an intermediate post fix form of the source code for each increment. The CONSIM source and intermediate-form modules are maintained by the system in ascending order by user supplied increment number. Thus, the increment numbers not only provide a means of identifying a particular increment (e.g. as required in the **show, delete** and **continue** commands) but also specify the order in which increments are to be combined to form the CONSIM program. This allows the user to enter or edit source code for increments in any order, with the system automatically updating the current source and intermediate-form modules as appropriate.

2.4 *Multiple tasking*

Multiple tasking in the CONSIM environment refers to a facility whereby the user may have more than one 'task' in progress simultaneously under a single computer job. Each task may consist of any of the CONSIM activities such as constructing, modifying, or executing programs. As each task is initiated (via the **new task** command) the user chooses an identifier for the task which is used in task specific user/system communication.

The possibility of multiple tasking in CONSIM was suggested by the multiple processing feature of Swinehart's COPILOT system (1974). (Because of the special connotations of the term 'process' in a simulation environment, the term 'task' was substituted in the CONSIM work.) One of Swinehart's primary goals was to create a programming environment in which the user could achieve a good 'behaviour match' with his or her normal working patterns. He believes that even while pursuing a single goal the user's attention may be swapped among several related or unrelated activities.

The multiple tasking facility appears especially useful in the conversational simulation environment in two ways. First, after initiating a long execution of one model which will require only infrequent monitoring, the user may wish to begin entering source code for another (related or unrelated) model. Because both tasks are being done under one computer job, any messages from the executing model will appear on the single terminal. When such messages appear the user can ignore them or respond at a convenient time.

The second way in which multiple tasking could prove useful is if the user had two related models, for example submodels of a single system. While such situations can be handled in a single task environment, by declaring each submodel as a separate task in a multiple tasking system the user would have additional control over their interaction and assistance from the system in keeping their declarations separate.

3. Utility of CONSIM's control enrichment features

Although only an experimental implementation, the CONSIM prototype is a true conversational simulation language: i.e. it provides simulation facilities while supporting mid-execution editability of programs and data. Thus, CONSIM offers an environment in which to assess the utility of conversational simulation, allowing exploration of the types of model building possible with the new language.

In the following subsections the utility of some of CONSIM's control rich features are illustrated by excerpts from a scenario of a model building session, given in more detail in Nelson (1977). The model being developed is one of an automated car wash installed at a service station. The example was suggested by Birtwistle, Dahl, Myhrhang and Nygaard (1973), who described the problem and developed a possible solution in SIMULA.

3.1 *User control over operation*

CONSIM offers the user considerable control over operation since the user, working at an interactive terminal, dynamically determines what processing or actions are to be undertaken by the system. Two operational modes are available: an immediate mode in which the user's directions are executed upon receipt and a stored mode in which the user's input undergoes syntax analysis before being stored for later execution upon user command. The modes may be intermixed as appropriate to carry out the desired processing.

Typically the user begins constructing a model by entering source statements for later execution. If an error is identified, the system returns an appropriate message and the user may either edit the source just entered to make appropriate corrections or may elect to re-enter that increment of source. In either case, the new statement is reanalysed for additional syntax errors. This process is illustrated in the following dialogue. (System messages will be shown in lower case with user responses in upper case).

```
initialisation complete
ready for next task
$CAR % NEW-TASK
consim source file name?
WASHER.CSM
car command?
% SOURCE
next increment number?
2
enter increment ending with #
PROCESS CAR; #
next increment number?
4
enter increment ending with #
IF LOUNGE.CARDINAL GT 0 THEN INVOKE
 GETWASHER;
syntax error in increment with stack of
@@@ ifcl unlabst;
car command?
% SHOW 4
4 if lounge.cardinal gt 0 then invoke getwasher;
car command?
% FIX\R;\R FI;#\
if lounge.cardinal gt 0 then invoke getwasher fi;#
car command?
```

In this excerpt the user opened a new task named CAR and began supplying CONSIM source, which is stored in a disc file, with the user named WASHER.CSM. During the syntax analysis of increment number 4, an error was identified. The prototype system does not provide comprehensive diagnostics: syntax errors are simply noted and current contents of the syntax stack are printed. In this example the user failed to enter the **fi** required by CONSIM's syntax to mark the end of the conditional statement. The error was corrected using CONSIM's edit command **fix**. The "%" which the user supplied preceding each command indicates that the instruction is to be executed upon receipt.

Once the model is sketched out, the user may request its execution by issuing the **run** command.

```
car command?
% RUN
consim running
car command?
```

Upon receipt of this command the CONSIM system completes a global syntax analysis on the stored statements (e.g. to match **begin/ends,** etc.) and initiates their interpretation. While the execution is in progress the user may interrupt by entering any command at any time. If the interrupting command calls

for interrogation and/or updating the current data or program, interpretation of that task must be explicitly continued, either from the point of interruption or some other user selected location. If the interrupting command indicates a change in the focus of attention to some other task, then execution of the CAR task will be automatically continued in a 'background' mode of execution.

The ability to perform mid-execution editing is especially useful in debugging and testing a model. When an error is identified (either by the user during model interrogation or by the system during execution), the user can edit the source program and, if necessary, modify data values. Execution can be continued, thus saving the time and effort which would be required to start the run over from the beginning. This process is illustrated in Section 3.3.

3.2 *The user as a control entity*

Besides controlling the overall operation of the CONSIM system, the user can be included in the model itself through a mechanism called the user portrayed process. When such a process is called for execution, interpretation of the model is temporarily suspended pending a **continue** command from the user. In the interim, the user has the full facilities of the CONSIM system at his or her disposal. The user may, for example, enter data, execute CONSIM code in the immediate mode to perform calculations or schedule processes, enter additional source for stored mode execution, or modify the model. Specification of these operations in the user portrayed process is facilitated and simplified by CONSIM's uniform language design.

Via user portrayed processes the model builder can direct processing for sections which have not been coded or for special testing and debugging. Through this mechanism the user may 'substitute' for uncoded portions of the model, either temporarily during model construction or permanently as a means of including the user within the model.

In the car wash scenario this facility may be used to allow the user to personally simulate the arrival of cars and their entry into the waiting line. The CONSIM source code for the generator routine is shown below:

```
20 process cargen;#
21 begin label loop;#
22 loop: immediate;#
23       go to loop end;#
```

When this CARGEN process is called for execution, interpretation will be suspended at the **immediate** instruction in increment 22 and the user will be given control. Any valid instruction supplied by the user at that time is then executed as shown below:

```
incr 22
enter statement for immediate execution ending with #
BEGIN REF C;
 C: = NEW CAR;
 C INTO WAITLINE;
 ACTIVATE C;
 HOLD 3
END #
```

Note that the immediate instructions included a scheduling statement **hold** which in effect causes this generator process to be rescheduled in three time units. The user may continue in this way, rescheduling the CARGEN process to occur whenever a car arrival is to be simulated. At some point the user may replace this user portrayed process with code, typically using pseudorandom numbers with a suitable distribution to simulate the arrival pattern.

Besides being useful in special testing of the debugging operations the user portrayed process is useful in providing for error processing within the model itself. By making certain error routines user portrayed, the model builder can dynamically determine the appropriate recovery procedure.

On a more permanent basis, user portrayed processes provide a means of actually including the user within the model, allowing certain portions of the simulated system to be reserved for dynamic modelling by the user. This is especially appropriate for sections of the model which require logically complex algorithms to provide for all possibilities, but which can be easily decided during execution by the user.

3.3 *New styles of programming*

In order to explore the types of programming possible in the new conversational environment, a number of test programs have been written and executed, using the prototype implementation. Although this phase of the research is by no means complete, this initial experience has shown that conversational simulation makes possible new techniques and styles of programming. Some of these are summarised below.

Examples in previous subsections of this paper have illustrated the highly interactive nature of the user/system interface. We have shown how features such as CONSIM's incremental syntax analysis and edit capabilities can facilitate the initial construction of programs. User portrayed processes can be used in program construction and testing as well as a means of including the user in the program to provide dynamic control decisions. We have also noted how the multiple task feature enables the user to pursue different programming activities all under a single computer job.

Perhaps the most unique and powerful feature offered by CONSIM is its mid-execution edit facility. This is especially useful in program debugging since, when an error is identified, both the code and program data can be corrected, as appropriate, and execution continued without restarting the program from its beginning.

The utility of this facility can be illustrated by a further excerpt from the car wash scenario described earlier. After the model has been executing for some time the user interrogates the model's statistics being maintained by the car wash program and discovers that a variable which is supposed to contain maximum queue length has an incorrect value. The source of the error is identified as a missing statement in the program. This statement may be added to the appropriate increment as shown below:

```
% SHOW 3
3 qlength: = waitline.cardinal;#
% FIX\#\IF MAXLENGTH LT QLENGTH THEN
    MAXLENGTH: = QLENGTH FI;#\
3 qlength: = waitline.cardinal; if maxlength lt qlength then
  maxlength: = qlength fi;#
car command?
```

The **fix** command corrected the error in the source, automatically updating increment 3's module of intermediate code. This solves the problem for the future but the current value of MAXLENGTH is incorrect. Rather than reinitiating the model from the beginning, the user may update the value using the **immediate** command.

```
% IMMEDIATE
enter statement for immediate execution ending with #
BEGIN MAXLENGTH := 1 END #
car command?
% CONTINUE
```

Once both code and data have been corrected the user is ready to CONTINUE execution. In the above example execution would continue from the point of interruption although the user could have specified some other resumption point.

Although the error illustrated above was simple, in a non-

conversational environment the user would have had to recompile the source and restart the model from the beginning in addition to correcting the statement. Restarting a model as small as the car wash represents only a minor inconvenience to the user but in larger and more complex simulation models significant amounts of user and computer time may be required for initialisation and achievement of steady state. With such models the mid-execution editability provided by a conversational environment can offer tremendous savings in both user and computer time.

CONSIM's mid-execution editing facility can also be used in modifying models during production phases. Unlike the debugging use, where mid-execution editing corrects a 'mistake' in a model, this second type of editing is planned as part of the model study. For example, a model of an assembly line may be constructed to be used to study the effect of upgrading one machine on the line with a newer version which contains more sophisticated quality control logic. In a conversational environment, such as that provided by CONSIM, the upgrade to the machine can be effected by allowing the user the halt the model at the 'time' of the upgrade, change the code which simulates the machine to conform to its modified specifications, and then continue the model's execution. Since only the internal configuration of the model for the affected machine is changed, the dynamic interrelationships among the components of the model are left intact: any queues, pending events, etc. are undisturbed by the changes.

Because of CONSIM's uniform language concept, modifications to executing models can be done in the immediate mode (as described above) or in stored mode. If the user, for example, wanted to study the effect of the machine upgrade under varying backlog conditions, he or she could write a CONSIM process which would modify the code for the machine to be upgraded (e.g. using **fix** in stored mode) and schedule this process to occur at the desired time of the machine modification.

The user, of course, is not limited in the number of either immediate or stored mode changes which he or she can make to an executing model. As a result in the CONSIM environment the model itself may be dynamic, undergoing various edits in the course of its execution. This aspect of conversational simulation is especially appealing in the use of simulation models in perturbation studies, where models are executed repeatedly with varying inputs. In the conversational environment more than just the inputs may be changed: the models themselves may be 'perturbed'.

Other new styles of programming are possible in this conversational simulation system. For example, since CONSIM supports execution of incomplete programs, it provides a good environment in which to apply some of the techniques of structured programming. One such technique, top-down model construction, was recently advocated by Chattergy and Pooch (1977). They note that top-down construction allows the integration of simulation program design and verification, resulting in better error detection and simplification of verification problems. In CONSIM the portions of the model not yet specified in detail can either be handled with 'program stubs or dummy subroutines' as described by Chattergy and Pooch (1977) or can simply be left undefined, allowing the user to model dynamically the activity to the desired level of detail at execution time via CONSIM's user portrayed facility.

4. The prototype implementation

The implementation of the conversational simulation language not only combines the problems of its conversational and simulation ancestors, but introduces some additional complexities inherent to the combination. A major goal of this research was to study these problems and explore their possible solutions. To facilitate this study an implementation language for the prototype was needed which was high level and rich in control and data structuring features. SIMULA 67 met these criteria. The availability of the DEC system-10 implementation (version 3) of SIMULA for use in this study further enhanced SIMULA's appeal. Therefore, SIMULA was used as both the model for CONSIM's syntax and as the implementation language for the prototype system.

4.1 *Overview of system structure*

CONSIM's implementation is arranged into five major components, each having a logically separate function. These divisions are intended to remain invisible to the user and were established to facilitate experimentation with implementation approaches in each of the areas. The five components of CONSIM's implementation are:

(*a*) *the user input receiver* which accepts all messages from the user, discriminates whether they are user commands, source entry, or data for an executing model and initiates appropriate system action

(*b*) *the user command processor* which provides the required processing for each of the user commands

(*c*) *the source analyser* which accepts the CONSIM source statements, performs a lexical scan and syntax analysis, and produces a post-syntactic program representation

(*d*) *the interpreter* which, using the intermediate form produced by the source analyser, effects the appropriate semantic processing, thus 'executing' the program

(*e*) *the system driver* which monitors overall operation of the system, executing an 'idle-loop' when no other actions are pending.

4.2 *Use of coroutines*

Each of CONSIM's components is implemented by one or more coroutines (SIMULA class instances). Like the block and procedure concepts, the coroutine concept facilitates modularisation by allowing the definition of local variables and code to handle specific subalgorithms. As a control module, however, the coroutine notion offers two additional advantages over those of block and procedure. First, variables local to a coroutine retain their values between reactivations and, secondly, upon coroutine reactivation control continues from the point of suspension rather than at the beginning of the module.

The retention of local variable values between activations enables coroutines to retain local state information or previously determined results, reserving global structures exclusively for common data accessible from all components. This differs from a system utilising procedures as the primary unit of program modularisation, in which case global structures must be used to retain values and state information between activations. This facility, coupled with the ability to resume control at the point of suspension, may be used to implement two types of coroutine control:

(*a*) parallel in which coroutines operate independently, logically separated from the processing of other coroutines and

(*b*) co-ordinated in which two or more coroutines each implementing partial algorithms 'co-operate' on required processing.

Since both parallel and co-ordinated relationships exist among the components of CONSIM, the coroutine construct was a logical choice as a basic control module and has been used extensively in the prototype implementation. Separate coroutines were constructed for the five components with appropriately defined inter-routine control paths. The actual

sequencing of control among the coroutines is dynamically determined by user provided commands.

Several benefits were realised by the extensive use of coroutines in the prototype implementation. First, the high level of modularity among the modules enabled faster construction and debugging of the code. It also encouraged experimentation since modules representing different implementation approaches could be developed and linked together to form various CONSIM 'systems'. A further advantage of implementing the components as coroutines was that it reduced time and ordering constraints required on user responses and requests, since the components issuing the request for user input could simply be suspended at the point of the message until the required response was received from the user. Upon receipt of a response the appropriate coroutine was reactivated to continue processing at the point of the request.

The use of coroutines in the prototype system also greatly facilitated the implementation of CONSIM's multiple tasking feature. Multiple tasking is supported by creating multiple instances of the command processor component, each responsible for a particular task. It first appeared that a separate instance of the source analyser would also be required for each task. However, by using SIMULA's remote accessing to associate all data of a particular task with the appropriate instance of the command processor, one source analyser is sufficient. This means, for example, that as the intermediate form of each increment is generated by the syntax analyser it is remotely stored in the appropriate command processor. Thus only one instance of the syntax analyser is required for all tasks. Because all task-specific data is associated with the command processor, the user may switch among tasks (utilising their assigned identifiers), performing syntactical analysis without destroying or overwriting the data of one task with that of another.

4.3 *Interpretive control self-modelling*

The dual use of SIMULA as CONSIM's syntax model and as its implementation language allowed a strategy called interpretive control self-modelling (ICSM) to be used in the implementation of the interpreter component (Nelson and Lindstrom, 1977). ICSM is achieved when a language has its control features 'reflexively' expressed in a self-interpreter. That is, recursion in the subject program is implemented via recursion in the interpreter, coroutines via coroutines, etc. The foremost example of this effect is the definition of LISP in terms of EVAL, a form evaluation function. The primary advantage of ICSM as an implementation strategy is that it allows extensive reuse of the base language implementation (i.e. SIMULA), thus expediously producing a concise and lucid interpreter.

Following the ICSM strategy, the CONSIM interpreter is organised so that its run time control state evolves in a manner directly parallel that evolving in the CONSIM subject program. Control simple features are administered by local control in the interpreter while control rich features are administered by the corresponding control action on an interpreter instance acting as a surrogate for its subject program control module. Thus, an interpreter instance is created for each CONSIM procedure, coroutine, and process being 'executed', as well as for CONSIM's main program. These interpreter instances then enter and leave SIMULA's run time operating chain in a manner paralleling that of the CONSIM program being interpreted.

This strategy requires multiple instances of the interpreter which can assume different modes of usage depending on the CONSIM subject program control events. It first appeared that four slightly different, but highly redundant, interpreter definitions would be required for processing the main program, procedures, coroutines and processes because of the variations in the semantic restrictions of each of these units (for example, **detach** is valid only in class bodies). The prefixing feature of SIMULA, however, facilitated a solution to this problem.

A basic interpreter definition was written as a class declaration called INTERP for one control environment, namely for procedure activations. This interpreter processes the intermediate form of the CONSIM source via a **switch** statement, branching to appropriate semantic processing routines each of which end with a **goto** back to the beginning of the loop. Although INTERP is actually a SIMULA class, it is coded so that it behaves like a procedure. The label for each semantic routine which is unique to the procedure environment is declared to be **virtual.** Definitions of interpreters for the other control environments (COINTERP for CONSIM classes and PINTERP for CONSIM processes) are then declared by using INTERP as a prefix and supplying appropriate redefinitions of the virtual labels of the semantic routines as needed. CONSIM's main program is interpreted by an INTERP-prefixed block within the main program of the implementation. A very brief skeleton of the interpreter structure, illustrating the redefinition of the semantic routine for CONSIM's **detach** (EX37), is shown below:

```
SIMULATION begin . . . ;
 class INTERP (. . .);
  virtual: label EX9, . . . , EX37, . . . ;
  begin . . . ;
   switch SWGO: = EX1, EX2, . . . , EX37, . . . , EX128;
   comment sequential interpreter cycle;
  MAINLOOP: NEXTCODE;
   goto SWGO [TR2]; . . . ;
   comment DETACH illegal unless in a coroutine;
  EX37: ERROR (7, 1);
   goto MAINLOOP:
  . . . ;
  end INTERP;
 INTERP class COINTERP;
  begin . . . ;
   comment DETACH for coroutines;
  EX37: CORTN.SELF:—none;
   detach;
   CORTN.SELF:—this COINTERP;
   goto MAINLOOP;
  . . . ;
  end COINTERP;
 INTERP class PINTERP;
  begin . . . ;
   comment DETACH for processes;
  EX37: CORTN.PSELF:—none;
   resume (MAIN);
   goto MAINLOOP;
  . . . ;
  end PINTERP;
 . . . ;
 comment main program;
 INTERP (. . .) begin
  . . . ;
   comment main program may not detach;
  EX37: ERROR (7, 3);
   goto MAINLOOP;
  . . . ;
  end prefixed block;
 end
```

Although the reuse of implementation through ICSM did shorten the time required to construct and debug the system and reduce the size of the prototype implementation (the interpreter, for example, is only 735 lines of SIMULA code

including generous comments), the resulting system is not very efficient in terms of memory requirements and execution time. For a full user oriented system some more efficient implementation techniques are required. The ICSM strategy, however, was found to be a good choice for the prototype system since it provided an environment which fostered experimentation and enhanced system clarity.

5. Results, conclusions and future research

Specific results obtained from the CONSIM project include the following:

(*a*) a running prototype system illustrating that conversational control and simulation control types can be coherently combined into a single usable system

(*b*) a non-trivial demonstration that ICSM can be used to express concisely and expediously a source language interpreter

(*c*) a further demonstration that SIMULA 67, in particular, has sufficient control richness to support both ICSM and control extensions appropriate to conversational computing and

(*d*) representative examples of new techniques for model development made possible in this new control domain.

Based on these findings and the insight gained from this research experience we have reached several conclusions. For example, combining other high level control types such as back-tracking and simulation appears a promising area of research. We anticipate that the major implementation problems which would be encountered in such projects would stem from the complexity of the internal control linkages required to support the control richness offered to the user.

Secondly, based on our successful use of ICSM in the CONSIM project, we feel that it can be a useful technique in other language experimentation studies. When experimenting with high level language forms, tools are needed which can help make the complexity manageable. We found ICSM to be such a tool since it allowed us to make extensive use of the control features of the underlying language (in our case SIMULA), thus enabling us to focus our attention on the extension to the control forms rather than the implementation of those control forms already included in SIMULA.

Our experiences further suggest that SIMULA should be given serious consideration as an implementation and experimentation language. We found SIMULA to be a well designed language, offering a richness in both control and data structuring facilities. The implementation of SIMULA which we used was remarkably error-free and its checkout package, called SIMDDT, was useful both in debugging the code and in monitoring the operation of the CONSIM system.

Based on our experiences in implementing and using CONSIM we feel that the following topics need further study preparatory to beginning a full implementation of a conversational simulation language:

1. *User guidance*
To be of maximum use systems as sophisticated as that required to support conversational simulation need to provide the user with some form of guidance in system use and operation. For example, consideration should be given to providing facilities for the user such as an on-demand menu of available options; warnings during syntax analysis of possible semantic errors (as a means of reducing run time errors); and suggestions and warnings on resumption points following user interrupts.

2. *System monitoring*
The system must provide the user with a comprehensive monitoring facility so that he or she may keep track of the status of operation. The work of Swinehart (1974) offers guidelines here.

3. *System state saving*
A mechanism for saving the state of the system is needed which can be used to 'reset' the system to a previous state of operation. Such a facility is useful for continuing work at one session from that done at a previous session and in implementing **forget** and **backup** features for the user.

4. *Compilation of stable programs*
Methods, such as incremental compilation, should be explored for use in optimising processing of stable programs.

5. *New programming styles and techniques*
Further work is needed in exploring the programming styles and techniques possible in the conversational simulation environment for both simulation and non-simulation use. CONSIM's multiple tasking facility appears to offer particularly enticing opportunities.

Appendix BNF specification of CONSIM

Increment syntax

```
1         <block> ::= <bh> <cpt> !
2                 <blkhd> <cpt>
3         <bh> ::= BEGIN
4         <blkhd> ::= <bh> <decl> !
5                 <blkhd> <decl>
6         <decl> ::= LABEL IDENT ; !
7                 INTEGER IDENT ; !
8                 REF IDENT ; !
9                 <cdecl> ; !
10                QUEUE IDENT
11        <cdecl> ::= <cohead> ; <block> !
12                <coparam> ; <block>
13        <cohead> ::= COROUTINE IDENT !
14                PROCESS IDENT
15        <clhead> ::= <fplist> ) ;
16        <fplist> ::= <cohead> ( IDENT !
17                <fplist> , IDENT
18        <coparam> ::= <clhead> INTEGER IDENT !
19                <clhead> REF IDENT !
20                <coparam> ; INTEGER IDENT !
21                <coparam> ; REF IDENT
22        <cpt> ::= <ct> <stmt> END !
23                <stmt> END
24        <ct> ::= <stmt> ; !
25                <ct> <stmt> ;
26        <stmt> ::= <unlabst> !
27                <labl> <unlabst> !
28                <cmd> !
29                <labl> <cmd>
30        <labl> ::= IDENT :
31        <unlabst> ::= <assign> !
32                <condl> !
33                GOTO IDENT !
34                READ IDENT !
35                PRINT <expl> !
36                <block> !
37                DETACH !
38                RESUME IDENT !
39                <sub> !
40                <subp> !
41                <sched> !
42                IDENT INTO IDENT !
43                IDENT OUT
44        <assign> ::= IDENT := <expl> !
45                IDENT := <RASS> !
46                IDENT := <rp> !
47                IDENT := <locref> !
48                IDENT := <qref>
49        <rp> ::= <rass> <aplist> )
50        <aplist> ::= ( <exp3> !
51                ( <locref> !
52                ( IDENT !
53                <aplist> , <exp3> !
54                <aplist> , <locref> !
```

```
55              <aplist> , IDENT
56      <sub> ::= INVOKE IDENT
57      <subp> ::= <sub> <aplist> ]
58      <rass> ::= NEW IDENT
59      <locref> ::= THIS IDENT !
60              CURRENT
61      <qref> ::= IDENT . FIRST
62      <exp1> ::= <exp1> + <exp2> !
63              <exp1> - <exp2> !
64              - <exp2> !
65              <exp2>
66      <exp2> ::= <exp2> * <exp3> !
67              <exp2> / <exp3> !
68              <exp3> !
69              IDENT
70      <exp3> ::= CONST !
71              ( <exp1> ) !
72              TIME !
73              IDENT . CARDINAL
74      <cond1> ::= <ifc1> <unlabst> FI !
75              <ifthe1> <unlabst> FI
76      <ifc1> ::= IF <relexp> THEN
77      <ifthe1> ::= <ifc1> <unlabst> ELSE
78      <relexp> ::= <exp1> EQ <exp1> !
79              <exp1> NE <exp1> !
80              <exp1> GT <exp1> !
81              <exp1> LT <exp1>
82      <sched> ::= <act> !
83              <act> AT <exp3> !
84              HOLD <exp3> !
85              PASSIVATE
86      <act> ::= ACTIVATE IDENT !
87              ACTIVATE CURRENT !
88              ACTIVATE <rass> !
89              ACTIVATE <rp>
90      <cmd> ::= <cmd1> !
91              <cmd2>
92      <cmd1> ::= NEW_TASK !
93              COMPILE !
94              SOURCE !
95              SHOW !
96              SHOW CONST !
97              DELETE CONST !
98              IMMEDIATE
```

```
99      <cmd2> ::= RUN !
100             HALT !
101             CONTINUE !
102             CONTINUE CONST !
103             FIX <TXTSTR> !
104             FINISHED !
105             STOP !
106             STATUS
107     <txtstr> ::= TXT
108     <ch> ::= <coparam> ; !
109             <cohead> ;
110     <rt1> ::= @ <block> # !
111             @ <block> ; # !
112             @ <blkhd> # !
113             @ <blkhd> <ct> # !
114             @ <bh> <ct> #
115     <rt2> ::= @ <ct> # !
116             @ <cpt> # !
117             @ <cpt> ; !
118             @ END # !
119             @ END ; #
120     <rt3> ::= @ <ch> # !
121             @ <bh> <ch> # !
122             @ <blkhd> <ch> #
123     <rt4> ::= @ <decl> # !
124             @ <decl> <ct> #
125     <root> ::= @ <rt1> # !
126             @ <rt2> # !
127             @ <rt3> # !
128             @ <rt4> #
```

Global syntax

```
1       <blk> ::= <bhead> <ctail>
2       <bhead> ::= BLKHD !
3               BH !
4               <bhead> DECL !
5               <bhead> <cdecl>
6       <cdecl> ::= CH BLOCK ; !
7               CH <blk> ;
8       <cstmt> ::= CT !
9               <cstmt> CT
10      <ctail> ::= CPT !
11              <cstmt> CPT !
12              <cstmt> END
13      <pgm> ::= @ BLOCK # !
14              @ <blk> #
```


References

BIRTWISTLE, G. M., DAHL, O. J., MYHRHAUG, B., and NYGAARD, K. (1973). *SIMULA BEGIN*, Auerbach Publishers Inc., Philadelphia, Pa.

CHATTERGY, R., and POOCH, U. W. (1977). Integrated Design and Verification of Simulation Programs, *Computer*, Vol. 10, No. 4, pp. 40-45.

DAHL, O. J., MYHRHAUG, B., and NYGAARD, K. (1968). SIMULA 67: Common Base Language, Norwegian Computing Center, Forskringsveien 1B, Oslo 3, Norway.

JONES, M. M. (1963). Incremental Simulation on a Time-Shared Computer, (Ph.D. Thesis) MIT MAC Report TR-48.

KIVIAT, P. (1974). Requirements for an Interactive Modelling and Simulation System, *Multi-Access Computing: Modern Research and Requirements* (ed. Paul H. Rosenthal and R. K. Mish), Hayden Book Co., Rochelle Park, NJ., pp. 53-61.

LEMON, M. J., LINDSTROM, G., and SOFFA, MARY LOU. (1977). Control Separation in Programming Languages, *1977 ACM Annual Conf. Proceedings*, pp. 496-501.

LINDSTROM, G. E. (1973). Prospects for Conversational Simulation, *4th Annual Pittsburgh Conference on Modeling and Simulation.*

LINDSTROM, G. (1977a). Final Technical Report: Control Definition in Programming Languages, Grant no. DCR73-03441 A01 from National Science Foundation to University of Pittsburgh.

LINDSTROM, G. (1977b). CACTUS: A LISP Self-Extension with Generalized Control, Dept. of Computer Science, Univ. of Utah, 41 pp., being revised for journal submission.

LINDSTROM, G., and NELSON, SALLIE S. (1976). Implementing Simulation Languages Through Simulation Language Programming, *Proc. 7th Annual Pittsburgh Conf. on Modeling and Simulation*, pp. 249-253; expanded version available as Tech. Report 76-3, Dept. of Computer Science, University of Pittsburgh, 20 pp.

NELSON, SALLIE S. (1977). Control Issues in the Development of Conversational Simulation Language, (Ph.D. Thesis) University of Pittsburgh, Pittsburgh, Pa., USA.

NELSON, SALLIE S. (1978). Conversational Simulation: A Control Enriched Environment for Model Building, *Spring COMPCON 78 Digest.*

NELSON, SALLIE S., and LINDSTROM, G. (1977). CONSIM: A Conversational Simulation Language Implemented Through Interpretive Control Self-Modeling, Technical Report UUCS-77106, University of Utah, Salt Lake City, Utah, USA, submitted for journal publication.

SOFFA, MARY LOU, and LINDSTROM, G. (1976). Describing and Testing Generalized Control Regimes Through Implementation Modeling, Tech. Report 76-11, Dept. of Computer Science, Univ. of Pittsburgh; being revised for journal submission.

SOFFA, MARY LOU, and LINSTROM, G. (1977). Analytical Properties of Generalized Coroutines, Tech. Report 76-10, Dept. of Computer Science, Univ. of Pittsburgh, 35 pp.; submitted for journal publication.

STRAUSS, D. (1976). Data Flow as a Run-Time Control Discipline, *Proc. Conf. on Information Sciences and Systems*, Johns Hopkins Univ. available as Tech. Report 76-9, Dept. of Computer Science, Univ. of Pittsburgh, 54 pp.

SWINEHART, D. C. (1974). COPILOT: A Multiple Process Approach to Interactive Programming Systems, (Ph.D. Thesis) Stanford University, Palo Alto, California.