CONSIM:

A CONVERSATIONAL SIMULATION LANGUAGE IMPLEMENTED THROUGH INTERPRETIVE CONTROL SELF-MODELING

by

Sallie S. Nelson(*) and Gary Lindstrom(+)

UUCS-77106

(*) Department of Computer Science
Texas A & M University
College Station, Texas 77843

(+) Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

July 7, 1977

Computing Reviews categories: 4.22; 4.13, 8.1.

Key words and phrases: conversational simulation language, interpretive control self-modeling, control structures, coroutines, SIMULA, interactive programming.

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

Abstract

This paper describes an implementation technique termed interpretive control self-modeling (ICSM) and outlines its application in the implementation of CONSIM, a prototype conversational simulation language. ICSM may be defined as the use of a higher-level programming language (HLL) to specify its own control organization through an interpreter administering each control event in a "reflexive" fashion. That is, recursion in the subject program is implemented via recursion in the interpreter, coroutines via coroutines, etc. Thus the run-time control state of the interpreter evolves in a manner directly paralleling that evolving in the subject program.

Although ICSM is a familiar idea in the context of LISP-like languages, it appears not to have been applied in more general purpose settings. We report here on the conceptual and practical advantages found in using ICSM in a SIMULA-67 environment to design and implement the conversational SIMULA variant CONSIM. Benefits resulting include conciseness and clarity of interpreter organization, ease of system modification, and control compatibility of CONSIM with SIMULA, thereby facilitating conversion of stable programs to compilable SIMULA. Disadvantages include system run-time size and speed, and awkwardness in doing control extensions beyond the scope of the underlying system. Future research suggested includes the formal specification of the ICSM process, adaptation to compiled

systems, and more thorough investigation of economic trade-offs involved in selecting ICSM as an implementation strategy.

1. Interpretive Control Self-Modeling

The problems of language description confront all individuals dealing with a higher level language: the designer, the specifier, the implementor, and the user. A major portion of these problems involve the control structures of that language, i.e. its facilities for logical program structuring and execution sequencing. Three approaches to control description can be identified:

i) abstract: in which control events are described in terms of an axiomatic foundation and invariant properties (e.g. Wang and Dahl's SIMULA work [1]);

ii) translational: in which control patterns are specified by means of an algorithm translating them into a language with known control semantics (e.g. the Vienna Definition Language), and

iii) interpretational: in which an algorithm is provided that directly performs control events when applied to a subject program (e.g. LISP's EVAL).

A programming concept which has proven useful in the specification of a certain class of control regimes in higher level languages is interpretive control self-modeling (ICSM), which deals with the use of particular control forms

to directly achieve their own modeling. ICSM is a refinement of interpretive control description in which the following added constraints are observed:

i) the language being described (the subject language) and the language in which the interpretive definition is expressed (the description language) are the same, and

ii) each control variety in the language (as subject language) is phrased in the interpreter (using the language as a description language) directly in terms of itself.

The foremost example of this effect is the definition of LISP through the LISP function EVAL. The conciseness and extensibility of this description has contributed to LISP's popularity as a base for language experimentation during the last fifteen years. Despite the attractiveness of ICSM in the LISP arena, there seems to have been little application of this technique in more general language settings.

2. CONSIM and SIMULA

The conversational simulation language CONSIM provides a vehicle for illustrating the broader application of ICSM. CONSIM was developed as a prototype system to demonstrate the feasibility and utility of combining the traditional facilities of simulation languages (e.g. coroutines, scheduling, and advanced data structures) with the advantages offered by a conversational environment capable

of supporting mid-execution editing of both programs and data [2]. Although such languages have been proposed and discussed in the literature ([3], [4], [5]), no full implementations appear to exist. "Interactive" simulation systems are available (see for example Fox and Pritsker [6]) which allow some on-line interaction between the modeler and the running program. Such systems, however, typically restrict the user's interactive options to interrogation of variables and/or supplying data to predefined input routines. Languages which meet our notion of conversational must be less restrictive and support more elaborate user control. In a truly conversational system, for example, the user must be able to interrupt execution at any time, interrogate and update variables, edit the program, and then continue execution from a user-specified point.

In order to support such a capacity for mid-execution editibility either an interpretive implementation or an incremental compiler is appropriate to provide the necessary dynamic run-time organization. We chose a mixed strategy for CONSIM, combining incremental syntax analysis with interpretive execution.

During CONSIM's design phase we noted that using an existing simulation language as a model would offer certain advantages. Such a technique, for example, would shorten the design phase by allowing us to take advantage of work

already accomplished in creating the existing language. Besides facilitating design reuse, fashioning CONSIM after an existing compilation-oriented language would insure that the new language would have at its core a control regime known to be suitable for later compilation. Thus stabilized conversational programs would have an improved chance of being readily translatable into the base language for compiled efficiency. Compatibility between an existing language and CONSIM would offer a further advantage in that users already familiar with the existing language could more easily adapt to the new environment, since their past experiences would be directly applicable.

For these reasons CONSIM was modeled after the existing simulation language SIMULA 67 [7]. After a careful survey of the currently available simulation languages, SIMULA 67 (hereafter referred to simply as SIMULA) was selected because it is a powerful modern language offering a comprehensive assortment of control features for both coroutining and simulation. SIMULA contains an image of ALGOL 60 as a subset, thus providing a good general purpose basis for the language. Furthermore, it is a "second generation" simulation language benefitting from considerable experience with its predecessor, SIMULA I.

These same high level features led us to select SIMULA as the implementation language as well as the design model

for CONSIM. We felt that SIMULA's selection as the implementation language would minimize reinvention, since many of CONSIM's features could be directly implemented using SIMULA's features. For example, SIMULA's statistical facilities and random number generators seemed prime targets for reuse in CONSIM.

Moreover, as we further examined the features of SIMULA applicable to our implementation needs, it became clear that SIMULA's control structures were ideal for administering CONSIM's control forms in the interpreter. Thus our implementation task became an ideal test case for evaluating ICSM beyond the LISP-like language domain. By utilizing the techniques of ICSM we were able to make extensive use of SIMULA's coroutine generation and sequencing primitives, as well as its simulation facilities, to implement directly the corresponding facilities for CONSIM.

3. Application of ICSM to CONSIM Implementation

An interpreter for a conversational system such as CONSIM must support many functions not found in a conventional interpreter, e.g. a terminal handler (with interrupt processing capability), a program editor, and a run-time program increment linker. These aspects of CONSIM's implementation, as well as the phases of the system which are wholly traditional in nature (lexical analyzer,

syntax analyzer, I/O routines, etc.), are tangential to our purpose here, and will not be considered in this paper (see [2] for a discussion of these topics). Instead, we focus on the control organization of the interpreter itself, which operates on a post-syntactic program representation (termed "triples") functionally equivalent to postfix code.

3.1 ICSM as an Implementation Strategy

Using ICSM as an implementation strategy, the CONSIM interpreter is organized so that its run-time structure (i.e. that of its dynamic block instances) evolves in a manner directly paralleling that evolving within the subject program being interpreted. Thus when a dynamic block instance is created and entered onto the operating chain in the CONSIM subject program, a corresponding block instance of the CONSIM interpreter is created and entered onto the SIMULA operating chain. In particular our adherence to ICSM means:

i) local control (i.e. sequential control and jumps within a block) is administered by local control within the current interpreter block instance;

ii) procedure entry and exit is administered by the creation, execution, and termination of a new interpreter instance invoked by the currently operating interpreter instance positioned at the CONSIM procedure's place of call;

iii) coroutine creation, activation, detachment, etc. are administered by the corresponding

actions on an interpreter instance dedicated to the execution of that coroutine instance in the CONSIM subject program, and

iv) process creation, scheduling, etc. is administered by the corresponding action on an interpreter process undergoing the same control actions.

(The remainder of section 3 assumes a basic familiarity with SIMULA and its control facilities. The reader is referred to [8] for suitable background information, if needed.)

3.2 Use of ICSM in CONSIM's Implementation

The realization of ICSM as described above implies that CONSIM's interpreter must be able to assume a number of different guises as source program control events dictate. That is, individual instances of an interpreter must model different control units (e.g. procedures, coroutines, processes) as those units dynamically arise during CONSIM program execution. At first examination, it may appear that four slightly different but highly similar interpreter definitions would be required for processing the main program, procedures, coroutines, and processes of the CONSIM program. While much of SIMULA's semantics are unchanged with these various control units, some contextual restrictions do exist (for example, detach is valid only in class bodies). The prefixing feature of SIMULA classes and blocks, however, facilitates an alternative to such a

multiple interpreter organization.

SIMULA's prefixing facility for classes and blocks provides a means of concatenating both the code and the data declarations of such textual units. In addition certain attributes such as labels may be declared to be "virtual", allowing their redefinition in particular prefixed subclasses at lower levels. With this capability in mind, an interpreter definition for the most general CONSIM control environment, namely for procedure activations, was written and used as a prefix to definitions for interpreters supporting the remaining control varieties. Because the basic CONSIM interpreter for processing procedures (a SIMULA class called INTERP) is used as a prefix to the definition of the interpreter for coroutines (called COINTERP), for simulation processes (called PINTERP), and for the main program block, only the parts which are different must be defined in the subclass declarations. The virtual facility allowed the redefinition of labels to reference specific overriding semantic routines, thus accommodating the differences between processing for procedures and for the other control blocks. Figure 1 shows a skeleton of the interpreter.

```
SIMULATION BEGIN
        .
        .
        CLASS INTERP ( ... );
                VIRTUAL: LABEL EX9,...,EX37,...;

        BEGIN
                SWITCH SWG0:=EX1,EX2,...,EX128;
                .
                .
                ! sequential interpreter cycle;
                MAINLOOP:NEXTTRIP;
                ! advance object program counter;
                GOTO SWG0[TR2];
                ! procedure termination;
                EX9:    GOTO ENDINT;
                .
                .
                ! DETACH illegal unless in a coroutine;
                EX37:   ERROR(7,1); GOTO MAINLOOP;
                .
                .
                ! procedure call;
                EX39:   NEW INTERP(...); GOTO MAINLOOP;
                .
                .
                ! coroutine and process creation;
                EX45:   IF ID.VAL=1 THEN
                        RIDENT.EXEC:-NEW COINTERP( ... )
                        ELSE RIDENT.PEXEC:-NEW PROC( ... );
                !  for coroutines create a COINTERP
                   instance and for processes create
                   a PROC instance;
                        GOTO MAINLOOP;
                .
                .
                ! PASSIVATE current process;
                EX85:   PASSIVATE;
                        GOTO MAINLOOP;
                .
                .
                ENDINT:
        END;
```

FIGURE 1: Interpreter Outline

```
INTERP CLASS COINTERP;
BEGIN
        .
        .
        ! DETACH for coroutines;
        EX37:   CORTN.SELF:-NONE;
                DETACH;
                CORTN.SELF:-THIS COINTERP;
                GOTO MAIN;
        .
        .
END;

INTERP CLASS PINTERP( ... ,PSELF);
BEGIN
        .
        .
        !DETACH for processes;
        EX37:   CORTN.PSELF:-NONE;
                RESUME(MAIN);
                GOTO MAINLOOP;
        .
        .
END;

PROCESS CLASS PROC( ... );
BEGIN
        REF(PROC)PSELF; REF(PINTERP)P;
        PSELF:-THIS PROC;
        P:-NEW PINTERP( ... ,PSELF);
END;

! main program;
INTERP ( ... )  BEGIN
                    .
                    .
                    ! MAIN program may not RETURN;
                    EX9:    ERROR(7,2);
                    .
                    .
                    ! MAIN program may not DETACH;
                    EX37:   ERROR(7,3);
                    .
                    .
                END;
END
```

FIGURE 1: Interpreter Outline (continued)

The local control structure within the INTERP class body consists primarily of a loop (beginning at label MAINLOOP) utilizing a switch to effect transfers to the appropriate semantic processing routines, most of which end with a goto back to the beginning of the loop. For each cycle through the loop the procedure NEXTTRIP is called to retrieve the next object program triple for interpretation. The EXEC numbers in each triple provide an index to the appropriate processing routine. Unless the CONSIM statement being executed alters the flow of control, the next sequential triple is always selected (via NEXTTRIP) for processing, providing sequential control within the block.

For each control action requiring generation of a new block instance (i.e. for a procedure, coroutine, or simulation construct), the currently operating interpreter creates an individualized instance of the appropriate interpreter. This interpreter instance then enters and leaves the SIMULA operating chain in the same manner as the corresponding source program activation record. The operating chain of the CONSIM program being executed consists of activation records for interpreter instances only, each processing specific activations of CONSIM control types (e.g. main program, procedures, coroutines, and simulation processes). This can be illustrated by six sample control events discussed in the following subsections: procedure call, procedure exit, coroutine

creation, coroutine detachment, process creation, and process passivation. Implementation of additional control events is given in [9] while a more detailed and complete presentation can be found in [2].

Procedure Call: Procedures are invoked by the semantic routine at label EX39 of Figure 1. Note that a totally new instance of INTERP is created, which oversees the entire execution of that subroutine instance. Because INTERP does not contain any SIMULA detach or resume instructions, each activation of INTERP obeys a stack-like protocol on the operating chain of dynamic block instances. Thus, although INTERP is itself a SIMULA class, on the operating chain its instances behave like SIMULA procedure instances.

Procedure Exit: When a procedure instance terminates (at label EX9) the paralleling INTERP instance also terminates by a jump to its exit label ENDINT. Since no references to this INTERP instance remain, it is no longer accessible in any way (consistent with SIMULA procedure activations). Control returns to the creating interpreter instance (i.e., the caller of the subroutine), poised in its EX39 routine. sequential control then continues from the place of procedure call in the object program.

Coroutine Creation: By utilizing the coroutine primitives new, detach and resume, the user can explicitly control the creation of coroutine instances and the suspension and resumption of their actions. With the inclusion of such facilities the operating chain of activation records no longer obeys a stack-like protocol in general. Coroutine instances which are suspended must be "swapped" off the chain and retained in such a way that they can be "swapped" back on if they are subsequently resumed.

When a coroutine is created, an action (EX45) takes place similar to that for a procedure call, except that a COINTERP subclass (of INTERP) is created rather than one of INTERP itself. This permits the general block mediation code of INTERP to be extended by the new definition (via virtual label placement) of routines particular to coroutine semantics. This is illustrated by the next subsection.

Coroutine Detachment: Only coroutines are permitted to detach (and not, e.g., procedures), so the occurrence of a detach in either the main program or a procedure instance results in generation of an error, as may be noted by the error call in the EX37 routine of INTERP. COINTERP and PINTERP, however, do permit detachment, accomplishing it by direct modeling in their respective EX37 routines.

When a CONSIM detach is encountered by a COINTERP or

PINTERP instance, a SIMULA detach is executed. This places the interpreter instance in the "detached" state, suspending CONSIM interpretation at that point. The suspended COINTERP instance is automatically retained by the SIMULA run-time system for possible resumption (e.g., via a resume or call statement).

Process Creation: We planned to implement CONSIM's simulation primitives in an analogous manner utilizing ICSM; that is, for each CONSIM process create an instance of PINTERP (the process interpreter class) and directly schedule it via SIMULA's simulation primitives. However, in order for PINTERP to be "schedulable" by SIMULA, it has to be prefixed by the system-defined class PROCESS. This was not possible however, since PINTERP is already prefixed by INTERP (in utilizing SIMULA's virtual facility to accommodate semantic variations between procedures and processes). Therefore a new class called PROC (shown in its entirety in Figure 1), was defined and prefixed by SIMULA's class PROCESS. Its primary action is to create a PINTERP instance for interpreting the CONSIM process. Thus the PROC/PINTERP pair represent a single CONSIM process on SIMULA's operating chain.

When a CONSIM process is created (at EX45), two dynamic block instances are generated: an instance of PROC which is directly schedulable as a SIMULA process, and an instance of

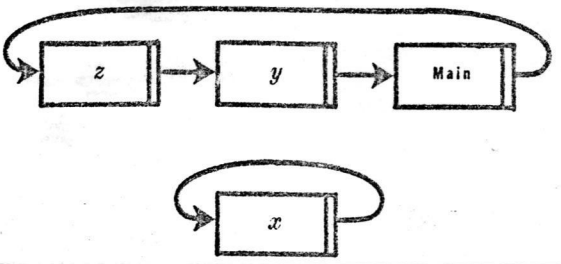
PINTERP which performs the "execution" of the CONSIM process's triples. This is a slight non-uniformity in ICSM faithfulness, but in no way encumbers the implementation.

Process Passivation: Once SIMULA's scheduling facilities were made available for direct use in CONSIM's implementation (through the use of SIMULA's standard class PROCESS), the ICSM strategy was very easy to apply. For example, the semantic processing for passivate (EX85) is shown in Figure 1, consisting simply of the SIMULA statement passivate.

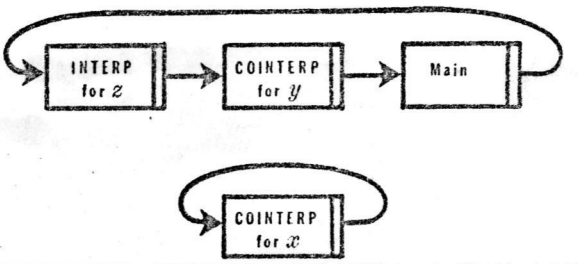
3.3 An Example

To further clarify the similarities of SIMULA and CONSIM processing, consider the operating chain for both systems while executing the following example. A main program creates coroutine instances X and Y. Each coroutine "detaches", returning control to the main program. The main program resumes coroutine instance Y which calls procedure Z. Using the Wang and Dahl operating chain diagrams the configuration at this point can be represented as follows:

SIMULA

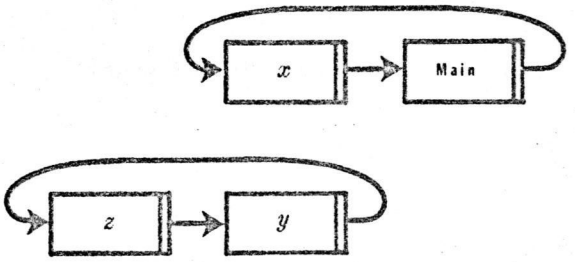


CONSIM

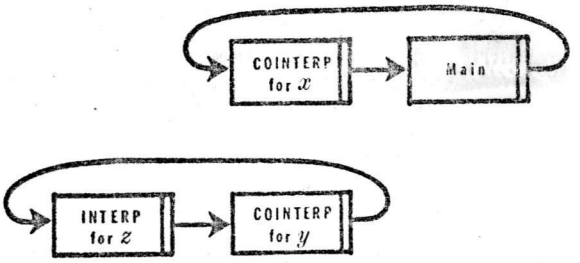


If procedure Z then executes the instruction resume X, the configuration becomes:

SIMULA



CONSIM



If coroutine instance Y is subsequently resumed, the activation records for both Y and Z will be swapped back onto the operating chain.

4. Advantages and Disadvantages of CONSIM's Use of ICSM

The use of ICSM greatly simplified the implementation of CONSIM's interpreter since the responsibility for much of the "bookkeeping" could be transferred back to the underlying SIMULA system. For example, because the interpreter instances enter and leave the operating chain as surrogates for the CONSIM activation records, the operating state of each interpreter instance ("attached", "detached",

or "terminated") directly reflects the state of the corresponding CONSIM control unit instance. As a result, the CONSIM implementation does not maintain separate state information for each source program unit; rather it makes use of the interpreter state as automatically updated by the SIMULA run-time system. Moreover, all the complexities of run-time storage management are "finessed" in this implementation through their complete delegation to the background SIMULA system.

Besides simplifying the interpreter's implementation, ICSM also clarifies its organization, making it attractive as a vehicle for language study and experimentation. As can be seen from the outline in Figure 1 the interpreter is not only highly readable but also isolates the processing for specific semantic actions (into the EX routines) and minimizes redundancy in the various control modules (e.g. COINTERP, PINTERP, etc.). The conciseness resulting from the use of ICSM is even more apparent in the full interpreter listing (given in [2]), which is only 734 lines long including comments.

One of the disadvantages of ICSM techniques in the implementation of CONSIM is the run-time size of the resulting system. For example, interactive construction and execution of a small model via the prototype implementation required 40K words on a DEC system-10 (see [2] for a

scenario of the model construction process). Although this level of overhead would normally be judged intolerable for a user-oriented implementation in a production environment, other factors, such as system development time and effort, as well as user flexibility, should be considered in assessing the true merit of this technique.

Although economics will probably necessitate the use of some other strategy in a full user-oriented implementation, ICSM was shown to be a useful approach in language experimentation, especially in the area of control structure design. The ability to use SIMULA's powerful primitives in implementing similar ones for CONSIM reduced the complexity of the interpreter to a manageable degree, facilitating experimentation and providing additional insight into system operation. This characteristic of ICSM makes it the kind of tool advocated by Winograd [10] for use in the design and development of today's complex systems.

5. Conclusions and Promising Areas of Further Research

The main question addressed by this paper can be succinctly stated as follows:

> What are the advantages, conceptual and practical, of using an advanced simulation language (SIMULA) to implement a replica of itself (CONSIM) through interpretive control self-modeling?

Our conclusions to this question may be summarized as

follows:

i) HLL power: an obvious first advantage lies in the sum total of all the features that a modern HLL such as SIMULA offers to the language implementor: code generation leverage, advanced data structures, and support facilities (I/O routines, linkers, debuggers, editors, etc.);

ii) parsimony: ICSM results in less source code in the interpreter, since activities such as process creation, detachment, and scheduling can be accomplished directly and reliably through direct usage of the same primitives within the interpreter;

iii) design reuse: for the language design (e.g. for us, that of CONSIM), the availability of a comprehensive language such as SIMULA offers a control design that is already thoroughly designed, evaluated and packaged, and

iv) compatibility: when the describing language is compilation-oriented (as is SIMULA), then the subject language has as a control core a regime known to be suitable for later compilation.

Our experience in this research suggests that ICSM can be a viable and useful technique for language research in a broader arena than simply that of LISP-like languages. In particular, we found SIMULA to be quite suitable for ICSM. Our preliminary research indicates, however, that when the interpretive language is extended to include control facilities beyond those of SIMULA, the strategy becomes less straightforward. For example, addition of a multi-level detach statement to CONSIM would be more difficult to implement, since the corresponding control construct does not exist in SIMULA. The availability in SIMULA of

primitives by which the user could directly access activation records to control their entry and exit from the operating chain would be useful in this regard. Further study is needed to identify the characteristics which make a language most suitable for use with ICSM.

Our results here indicate that further research is merited into interpretive control self-modeling as a language property and as a programming tool. The attractiveness of icsm as a means of insight into HLL control structures would be further illuminated by examination of the following questions:

i) how can the process of ICSM be formally specified? (Reynolds [11] has offered some directions here.)

ii) what language control features are naturally suitable for ICSM, and why?

iii) how do we prove the correctness of an instance of ICSM?

Beyond its conceptual advantages, ICSM has been shown here to offer a practical programming aid in certain system implementation areas. A full assessment of ICSM's practical utility would include:

i) an analysis of run-time economic issues, especially in terms of storage management;

ii) a thorough study of the nature and attractiveness of ICSM when adapted to a variety of positions on the interpret/compile spectrum, and

iii) an investigation of what application areas ICSM techniques are best suited for (e.g., can the idea be adapted to support operating system construction using simulation languages?).

REFERENCES

[1] A. Wang, and O. J. Dahl, "Coroutine Sequencing in a Block Structured Environment," BIT 11 (1971).

[2] Sallie S. Nelson, "Control Issues in the Development of a Conversational Simulation Language," (Ph.D. Thesis) University of Pittsburgh, Pittsburgh, Pa. (April 1977).

[3] Malcolm M. Jones, "Incremental Simulation on a Time-Shared Computer," (Ph.D. Thesis) MIT MAC Report TR-48, 1968.

[4] Philip J. Kiviat, "Requirements for an Interactive Modeling and Simulation System," MULTI-ACCESS COMPUTING: MODERN RESEARCH AND REQUIREMENTS (ed. Paul H. Rosenthal and R. K. Mish), 1974.

[5] Gary E. Lindstrom, "Prospects for Conversational Simulation Programming," 4TH ANNUAL PITTSBURGH CONFERENCE ON MODELING AND SIMULATION, (April 1973).

[6] Mark A. Fox, and Alan B. Pritsker, "Interactive Simulation with GASP IV on a Minicomputer," SIGSIM, Vol. 6 No. 3, (April 1975).

[7] O. J. Dahl, B. Myhrhaug and K. Nygard, SIMULA 67 COMMON BASE LANGUAGE. Norwegian Computing Center, Norway (May 1968).

[8] J. D. Ichbiah, and S. P. Morse, "General Concepts of the SIMULA 67 Programming Language," ANNUAL REVIEW IN AUTOMATIC PROGRAMMING, Vol. 7, Part 1 (1972), pp. 65-93.

[9] Sallie S. Nelson, and Gary E. Lindstrom, "CONSIM: A Case Study in Interpretive Control Self-Modeling," Technical Report 76-12, University of Pittsburgh (1976).

[10] Terry Winograd, "Breaking the Complexity Barrier Again," SIGPLAN, Vol. 10, No. 1 (January 1975), pp 13-20.

[11] John C. Reynolds, "Definitional Interpreters for Higher-Order Programming Languages," PROCEEDINGS ACM NATIONAL CONFERENCE, (1974).