A STRUCTURED EDITOR FOR PROCESS SPECIFICATION

by

Jeffrey C. Sedayao and Lawrence A. Rowe

Memorandum No. UCB/ERL M90/48

24 May 1990

A STRUCTURED EDITOR FOR PROCESS SPECIFICATION

by

Jeffrey C. Sedayao and Lawrence A. Rowe

Memorandum No. UCB/ERL M90/48

24 May 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A STRUCTURED EDITOR FOR PROCESS SPECIFICATION

by

Jeffrey C. Sedayao and Lawrence A. Rowe

Memorandum No. UCB/ERL M90/48

24 May 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A Structured Editor for Process Specification[†]

Jeffrey C. Sedayao[‡]

Lawrence A. Rowe

Computer Science Division - EECS
University of California
Berkeley, CA 94720

[†] This research was supported by the National Science Foundation (Grant MIP-8715557) and The Semiconductor Research Corporation, Philips/Signetics Corporation, Harris Corporation, Texas Instruments, National Semiconductor, Intel Corporation, Rockwell International, and Siemens Corporation with a matching grant from the State of California's MICRO program.

[‡] Current address: MS SC1-02, Intel Corporation, P.O. Box 52126, Santa Clara, CA 95052.

SUMMARY

A process-flow language for integrated circuit computer integrated manufacturing defines the operations performed on a silicon wafer to manufacture IC's. It is used as input to a variety of systems, including processing equipment, simulators, and schedulers. Since process-flow specifications will play a central role in wafer fabrication, many people with widely varying programming skills must be able to create, edit and examine them. A structured editor and process-flow browser for the Berkeley Process-Flow Language was developed using the Comell Program Synthesizer Generator. Process-flow procedures are stored in a relational database and a simple version control system and process-flow library browser was implemented.

KEY WORDS: Structured editor generators, Process-flow languages, Database browsers

INTRODUCTION

The fabrication of Integrated Circuits (IC's) requires that a sequence of treatments or operation steps be performed on a silicon wafer. This sequence of operations is known as a process-flow. Current research in IC Computer Integrated Manufacturing (IC-CIM) involves the development of a language to specify process-flows. This language is the input to interpreters that operate processing equipment, simulate the process, and schedule work in a fabrication facility (FAB). In such an environment, process-flow programs are the specification that drives the FAB.

With process-flow specifications playing such a central role, a FAB control system must fulfill a number of requirements. First, a process-flow language must be accessible to a wide variety of interpreters. Consequently, process-flow specifications should be stored in some kind of database, preferably a relational database. Second, process engineers must be able to edit and change the specifications. Older versions of a specification should be kept, and these versions should be accessible to the engineers and FAB control personnel. Finally, the system must allow process engineers to browse through the large number of specifications that will be stored in the database. A mechanism is needed that allows the user to query the specifications because hundreds of specifications may be present in the

database.

The Berkeley Process-Flow Language[1] (BPFL) is a process-flow language developed at the University of California at Berkeley. BPFL is derived from Common LISP[2], with new functions added to the language for describing processing and scheduling constraints.

One controversial aspect of BPFL is its LISP syntax. Members of the process engineering community express confusion and horror at the prospect of editing specifications with lots of parentheses. "We do not want to have to hire a computer science Ph.D just to type in process flows," complained one person from the semiconductor industry. This article describes the Structured Editor for Process Specification (SEPS), an editing and browsing system designed to solve this problem.

SEPS has three requirements. First, it must have an editor that simplifies the examination and modification of process specifications. This editor must deal with the LISP syntax of BPFL. In particular, it should enter the parentheses and make sure that they are matched. The editor should also exploit bitmapped workstations and use modern user interface tools (e.g. mouse, menus, dialog boxes, etc.). Second, SEPS must store process specifications in the FAB database. An integrated, shared dtabase is a key feature of a modern CIM environment. Finally, SEPS must have an interface that allows process engineers to browse the specifications in the database.

An editor tuned to the syntax of a language can ease the development of specifications. These editors are called "structured editors" or "language sensitive editors." Two alternatives for the implementation of a language sensitive editor were considered. The first alternative considered was to use EMACS[3] which is a general-purpose, extensible editor. EMACS already has a LISP sensitive mode that could be extended to handle BPFL syntax and semantics. The second alternative considered was to use a structured editor generator to produce a BPFL-specific editor. We did not consider building a structured editor from scratch because this alternative is too expensive. We decided to a use a structured editor generator because we thought it would be easier to implement the editor in this way.

The structured editor was designed to check both the syntactic and semantic correctness of a BPFL program. A syntax error like the following, a division function without a divisor:

(/ 3)

would be prevented. While syntactically correct, the next example contains a semantic error

(/ 3 0)

because division by zero is not permitted.

The Cornell Program Synthesizer Generator[4] (CPSG) was used to build SEPS. CPSG is given a description of a language, the format to display the language, and the semantic checks to be performed. It generates a structured editor for the language described. The input language to CPSG is called the Synthesizer Specification Language (SSL). The generated editor runs under the X Window System[5] and provides a workstation interface (i.e. pop-up menus and mouse input).

CPSG was chosen for a number of reasons. First, CPSG's use of an input language to generate an editor supports rapid prototyping of the structured editor. Second, CPSG editors have more capability than other structured editor generators that were available[6], and we wanted to evaluate CPSG as an editor generator.

After the SEPS Structured Editor was built, the need for some way to browse a collection of process specifications became apparent[7]. Two alternatives were considered. The first alternative was to use the RTI INGRES Forms system[8] a browsing facility. A SEPS user would use INGRES forms to look at what specifications were present, and then select a specification that would be passed to SEPS. One advantage of this approach is that the forms system is used by other applications in the Berkeley FAB. A disadvantage of this approach is that the RTI INGRES Forms System does not provide a workstation interface, so users could not use a mouse.

The second alternative was to use CPSG to create a browsing interface. This alternative would provide a consistent user interface. A mouse could be used to select menus and to choose programs to edit. CPSG has facilities to add new commands and dialog boxes to an editor, so this alternative was feasible. Consequently, this alternative was chosen.

The remainder of this report describes SEPS. SEPS facilities to browse and edit BPFL specifications are demonstrated in the next section. The third section describes the implementation of SEPS. The implementation of the browser and the database representation are described. The next section discusses the experiences of people who have used SEPS. The last section examines the current state of SEPS and proposes areas where it could be improved.

AN EDITTING EXAMPLE

SEPS allows users to browse through a database of BPFL routines. The user can select routines to edit with the SEPS structured editor. Multiple edit windows can be open at the same time, so the user can edit several routines simultaneously.

The first part of this section describes the SEPS interface, and provides the background necessary to describe a user session. The second part shows the browsing capabilities. The third part demonstrates the structured editor. The last part shows examples of syntax and semantic error checking implemented by the editor.

SEPS Windows

SEPS contains three types of windows. The *browse window* lists BPFL routines in the database. The user can browse through the database and select a routine to edit. Once a routine has been selected, an *edit window* appears. The edit window contains a BFPL specification. Unlike a browse window, several edit windows can be open at a time. Some SEPS commands require parameters from the user. The user is prompted for these parameters by a *dialog box*, which is a window dynamically displayed to the user (i.e. they pop-up).

All windows in SEPS have the same format, whether they are browse windows, edit windows, or dialog boxes. An example is shown in Figure 1. A SEPS window contains four horizontal panes. At the very top of the window is a *title bar*. The title bar contains one line of status information about the window and is displayed in reverse video. The window in Figure 1 is a browse window, as noted in the title bar. Below the title bar is the *command line*. The command line displays the name of the

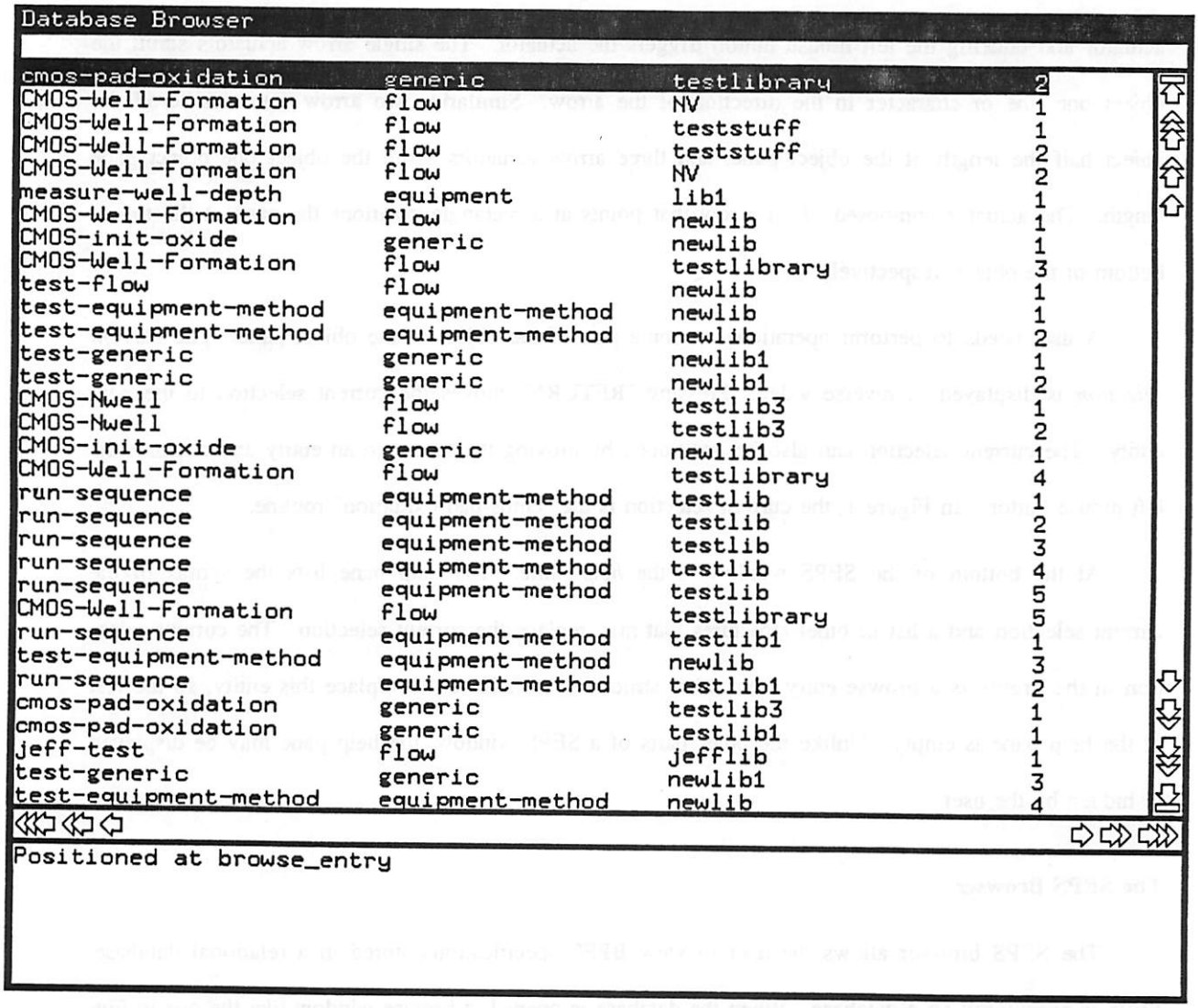

Figure 1: SEPS browsing window

command being executed or an error message. The command line in Figure 1 is blank.

Below the command line is the *object pane*. This pane contains either a table of specifications to be browsed, a BPFL routine to be edited, or a dialog box to be filled in. In Figure 1, the object pane contains a list of BPFL routines in the database.

On the right and at the bottom of the object pane are *scroll bars*. The scroll bars contain *actuators* (i.e., the arrow icons) that control how far the object is scrolled. Positioning the cursor over an actuator and clicking the left mouse button triggers the actuator. The single arrow actuators scroll the object one line or character in the direction of the arrow. Similarly, two arrow actuators scroll the object half the length of the object pane, and three arrow actuators scroll the object one object pane length. The actuator composed of an arrow that points at a rectangle positions the pane at the top or bottom of the object, respectively.

A user needs to perform operations on some part of the object in the object pane. The *current selection* is displayed in reverse video. Pressing "RETURN" moves the current selection to the next entity. The current selection can also be positioned by moving the cursor to an entity and clicking the left mouse button. In Figure 1, the current selection is the "cmos-pad-oxidation" routine.

At the bottom of the SEPS window is the *help pane*. The help pane lists the syntax of the current selection and a list of other structures that may replace the current selection. The current selection in the firgure is a browse entry. No other structures are allowed to replace this entity, so the rest of the help pane is empty. Unlike the other parts of a SEPS window, the help pane may be displayed or hidden by the user.

The SEPS Browser

The SEPS browser allows the user to view BPFL specifications stored in a relational database. SEPS is executored on a database. When the database is opened, a browse window like the one in Figure 1 is displayed.

A routine description includes the routine's *name, level, library,* and *version number. Level* is an attribute that specifies how it is used. BPFL version 1 used the run-time level to indicate whether it

defined a process-flow ("flow"), generic equipment ("generic"), or specific equipment ("equipment") abstraction of the process. In addition, a routine can be a procedure or a method. *Library* names a collection of routines with which the specification is associated. *Version* is the routine version number. Libraries and versions are discussed in the next section.

Browsing through a database that contains hundreds of specifications can be time consuming. Consequently, SEPS permits a user to query the database for the specifications of interest. The **filter-specifications** command selects the routines in which the user is interested. A dialog box is displayed to the user when the command is executed. The user can enter a pattern to select the desired routines.

Figure 2 shows an example of a filter dialog box. Next to **Function Name:**, **Function Type:**, **Library name:**, and **Version:** are the patterns that must be matched. Specifications that match all of the patterns are displayed for browsing. Asterisks are wildcards that indicate that any value will match. The pattern in Figure 2 selects all routines in the library "testlib3."

Dialog boxes have buttons in the command line to signal completion of the dialog **(Start)** or to cancel the dialog **(Cancel).**

Figure 3 shows the browse window after the database has been queried to find the routines in **testlib3**. At this point, the user selects "CMOS-Nwell version 2" and executes a command to create an edit window shown in figure 4. The title bar of the edit window includes the level, name, library, and version number.

The SEPS Structured Editor

One goal of SEPS was to eliminate the typing of parentheses. Consequently, SEPS has a template menu with an entry for each BPFL construct including the parentheses. The user enters statements by selecting a template and filling it in.

Figure 5 shows an edit window opened on the CMOS-Nwell routine with the selector positioned on **<Flow level entry>**. Choices for flow level entries are displayed in the help pane. A user can enter, for example, a **comment**, a **Declare** statement, or an **Anneal** function in place of the current selection. The user can either type in the entry, or select an entry in the help pane with the mouse.

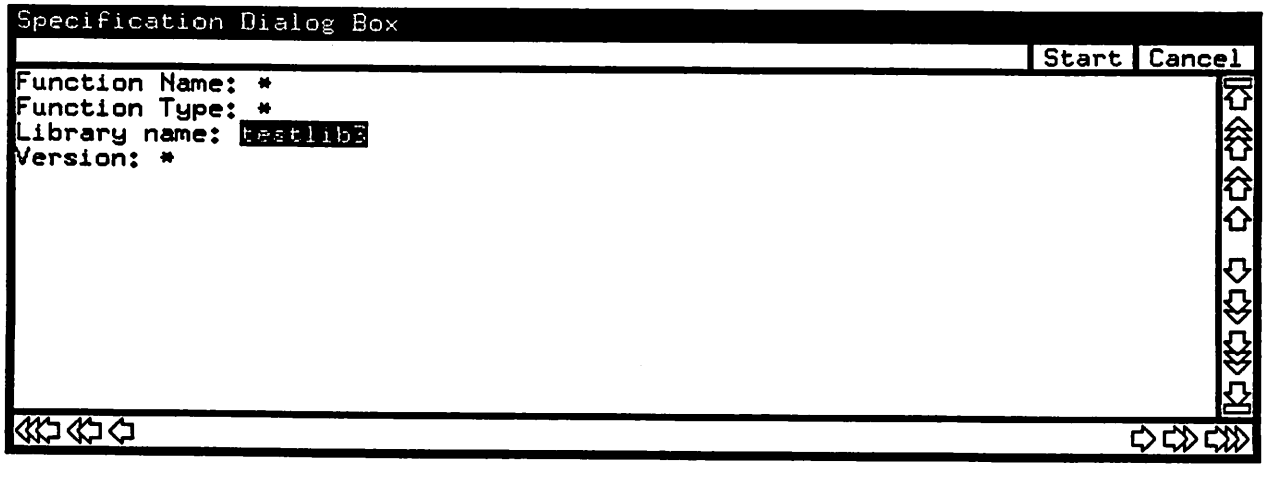

Figure 2: Dialog box for setting a filter

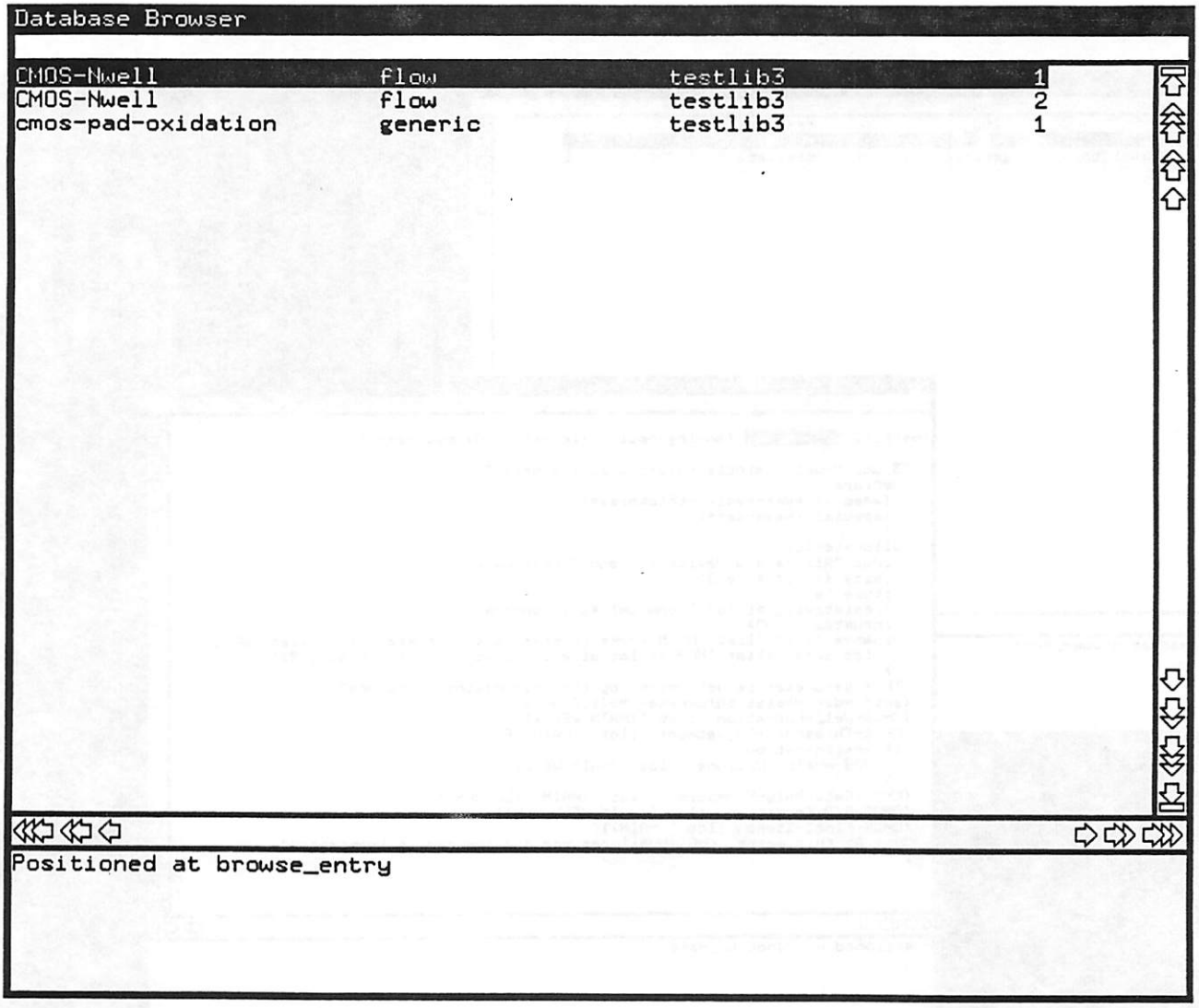

Figure 3: Result of a filter operation

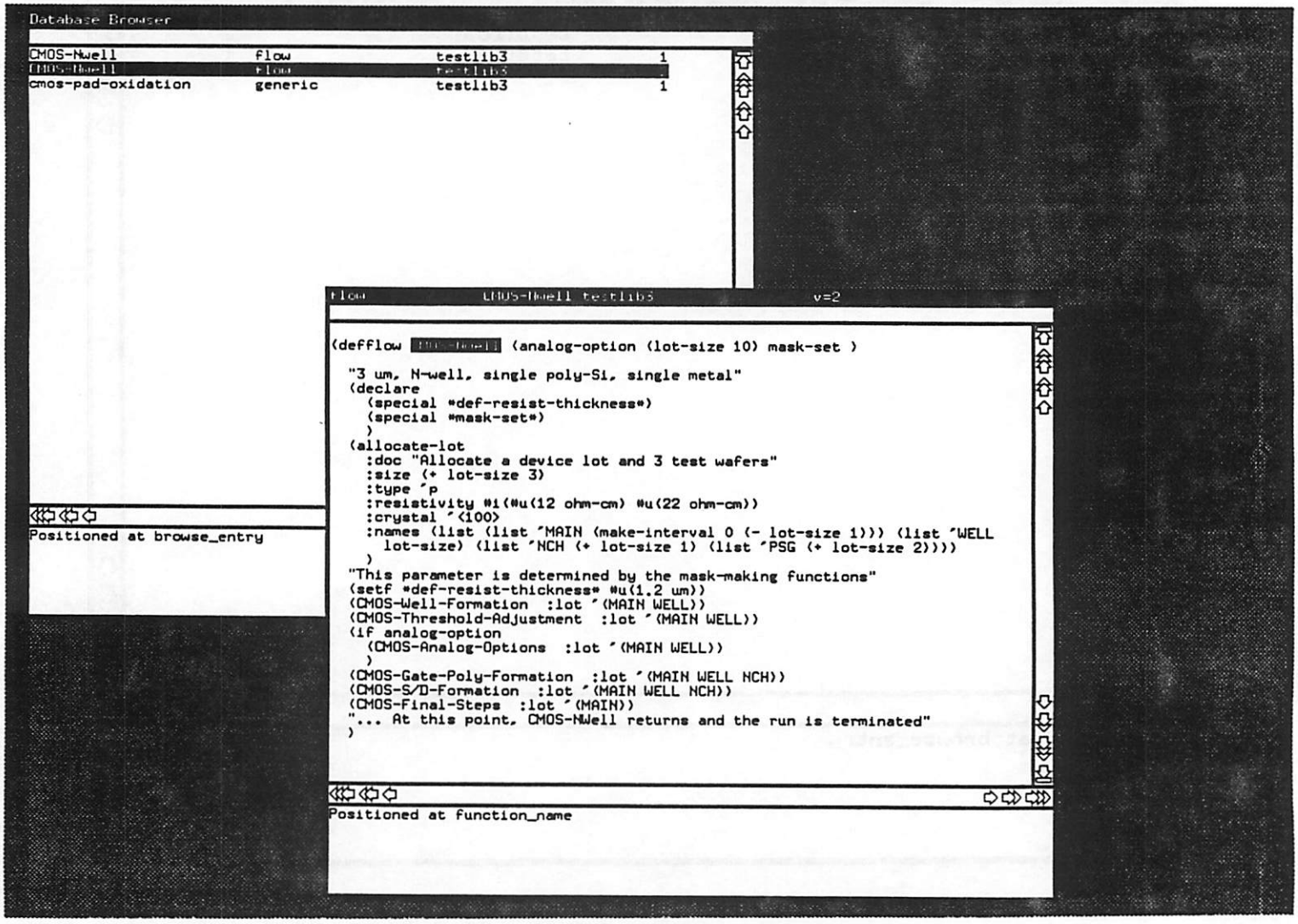

Figure 4: SEPS windows on a workstation display

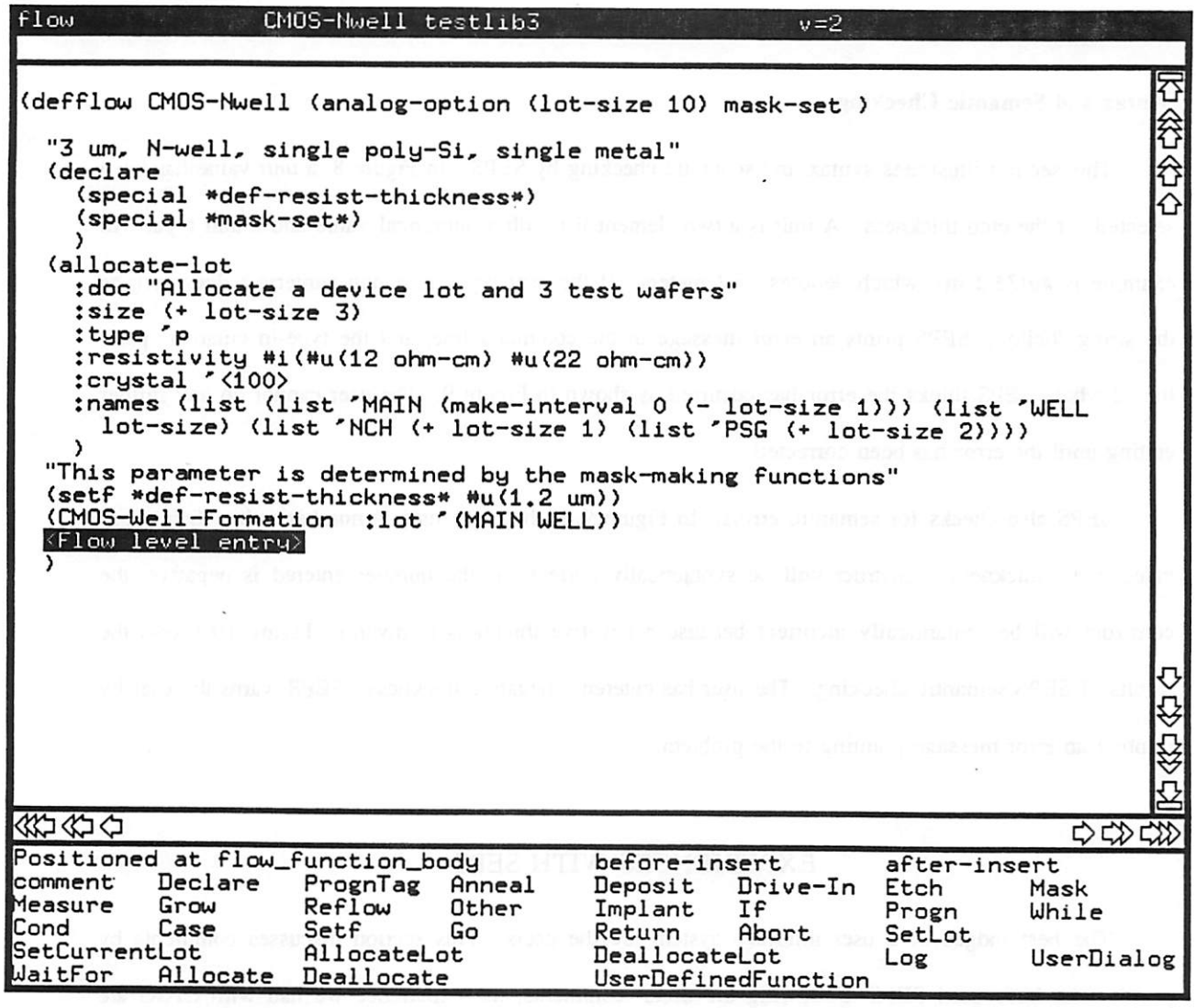

Figure 5: Positioned at "Flow Level Entry"

Figure 6 shows the edit window after the user has selected a flow-level statement (an **Etch** command). A template for the **etch** function is displayed, and the selection is now positioned on an "etch keyword." * Notice that keywords are now displayed in the help pane. Figure 7 shows the result after the "thickness" keyword has been chosen. The function is completed by entering a value for the etch thickness. No parentheses were typed. The user only needed to select the choices in the help pane or type in the entry.

Syntax and Semantic Checking

This section illustrates syntax and semantic checking by SEPS. In Figure 8, a *unit* value has been selected for the etch thickness. A unit is a two element list with a numerical value and a unit type. An example is **#u(35.2 m)**, which denotes 35.2 meters. If the user types in a non-numeric value (such as the string "hello"), SEPS prints an error message in the command line, and the type-in cursor is positioned where SEPS thinks the error has occurred as shown in Figure 9. The user cannot do any further editing until the error has been corrected.

SEPS also checks for semantic errors. In Figure 9, if the user enters a number rather than a unit value, the ":thickness" construct will be syntactically correct. If the number entered is negative, the construct will be semantically incorrect because a negative thickness is invalid. Figure 10 shows the results of SEPS semantic checking. The user has entered a negative thickness. SEPS warns the user by printing an error message pointing to the problem.

EXPERIENCES WITH SEPS

The best judges of a user interface system are the users. This section discusses comments by people who have used SEPS. Following the users' comments, the experience we had with CPSG are described.

---

*A keyword is an identifier with a colon prefix. Keywords in LISP are used to name the formal parameter to which the actual argument is bound.

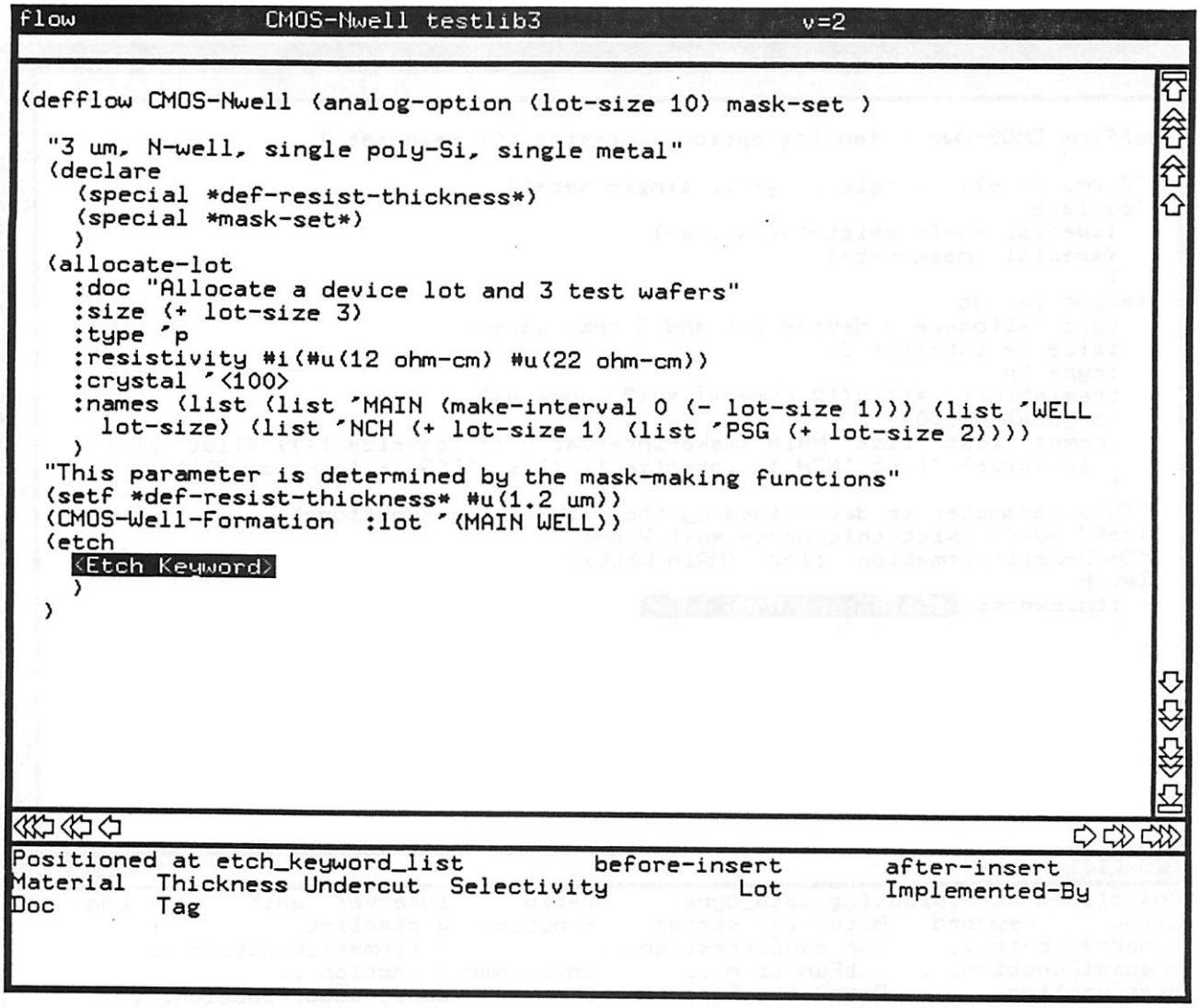

Figure 6: After "Etch" has been selected

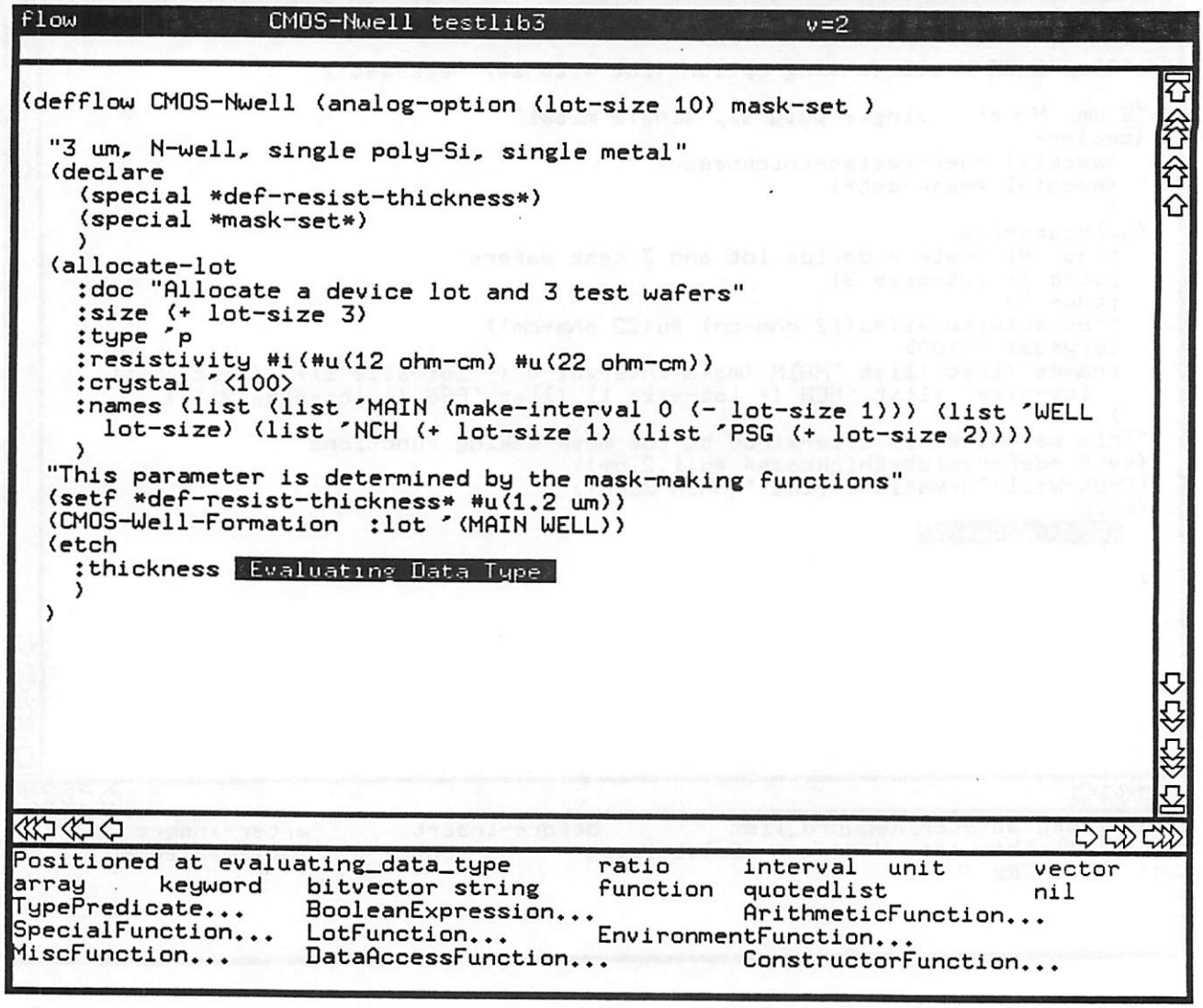

Figure 7: After "Thickness" has been selected

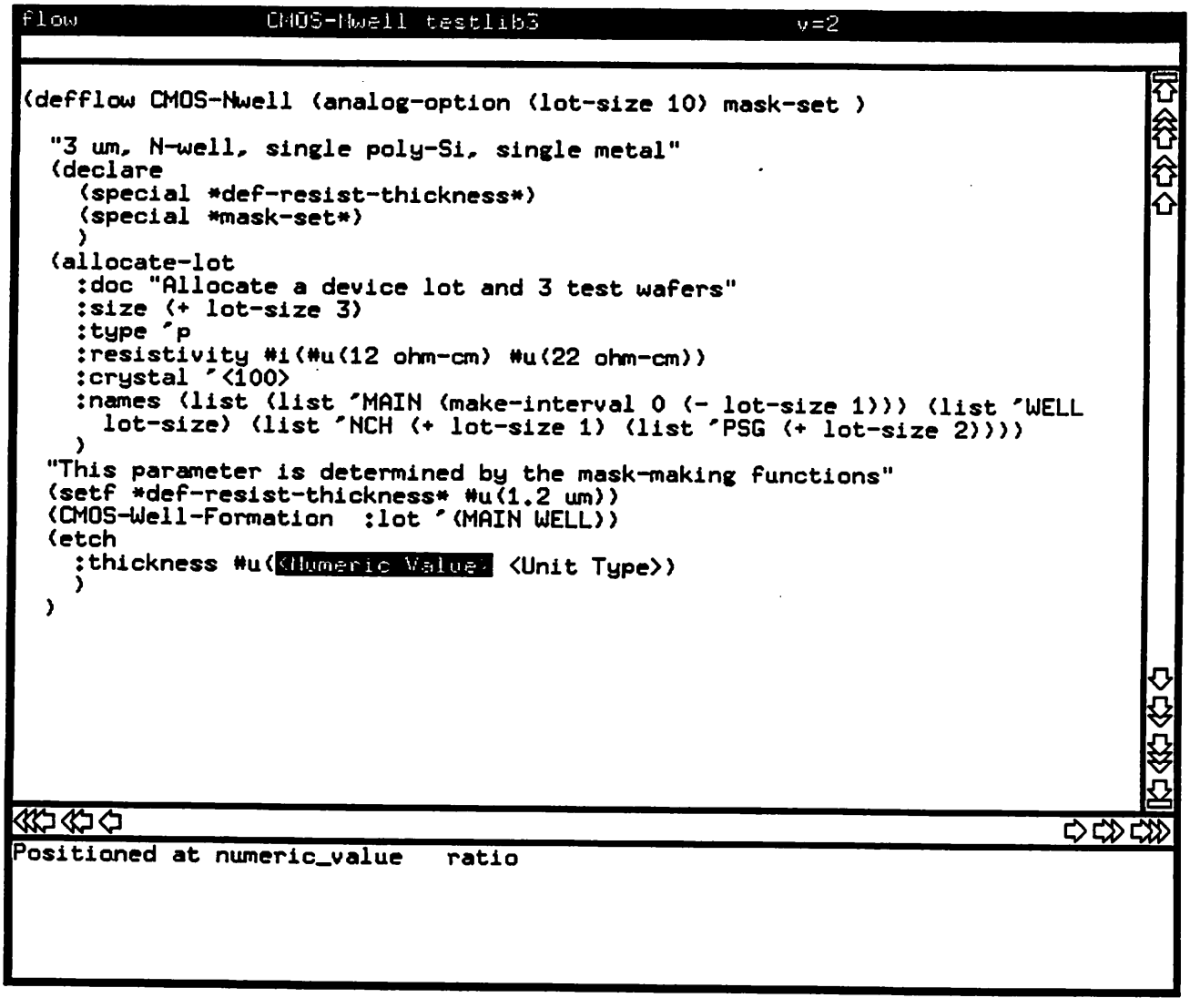

Figure 8: After "Unit" has been selected

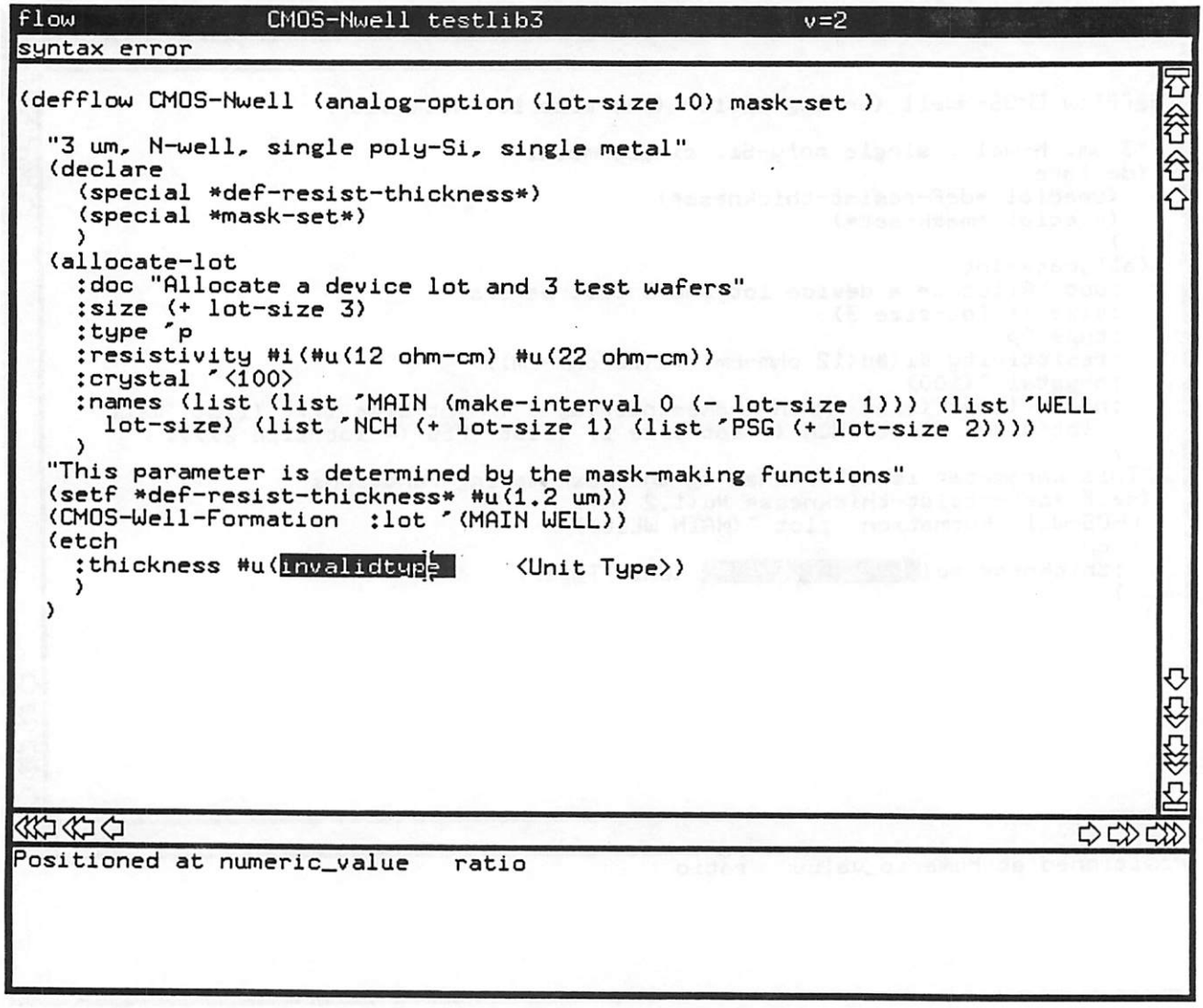

Figure 9: Result of Syntax Error

User Feedback

The structured editor portion of SEPS was used by several people. Most people already knew BPFL. Generally speaking, user feedback was positive. One users had previous experience with a structured editor, and that experience was not positive. Much to his surprise, he found SEPS easy to use and said that he would be more productive if he used it. One feature that he particularly liked was the ability to chose BPFL functions from a menu and to have the editor display a template for the function. Several people complimented the system on its robustness.

Users did encounter problems and features that they did not like. They complained that it could not be used as a simple text editor. On occasion, the user wanted to treat the program as text and edit it. CPSG generated editors do not allow users to edit the text freely since they generate a structured description of the program. Once users had become familiar with BPFL, they wanted to be able to type in whole functions and not use menu selections.

Another source of complaints stemmed from the way SEPS does error checking. The standard error message ("syntax error") is not conspicuous and it does not always identify the problem. This complaint reflects on the way SEPS handles errors. When a syntax error is found, the cursor is moved to the location of the problem, and "syntax error" is displayed. Even though SEPS points out the problem, the user is not told how to correct it.

Semantic errors are also handled poorly. While the error messages from semantic errors can be less vague than "syntax error", they can only be displayed in the object pane and not in the command line. An example of this problem was shown in Figure 10.

There were a variety of other complaints. One user thought that the editor should be more knowledgeable about BPFL. When creating a function, he wanted the editor to know about the most common arguments. For example, the "material" slot in the "deposit" function uses a small set of commonly used materials. He mentioned that it would be convenient to have these common materials on a menu with a choice for the general case. Other complaints centered around the CPSG generated interface. For example, users complained about the menu and scrolling abstractions. These features are inherent in CPSG.

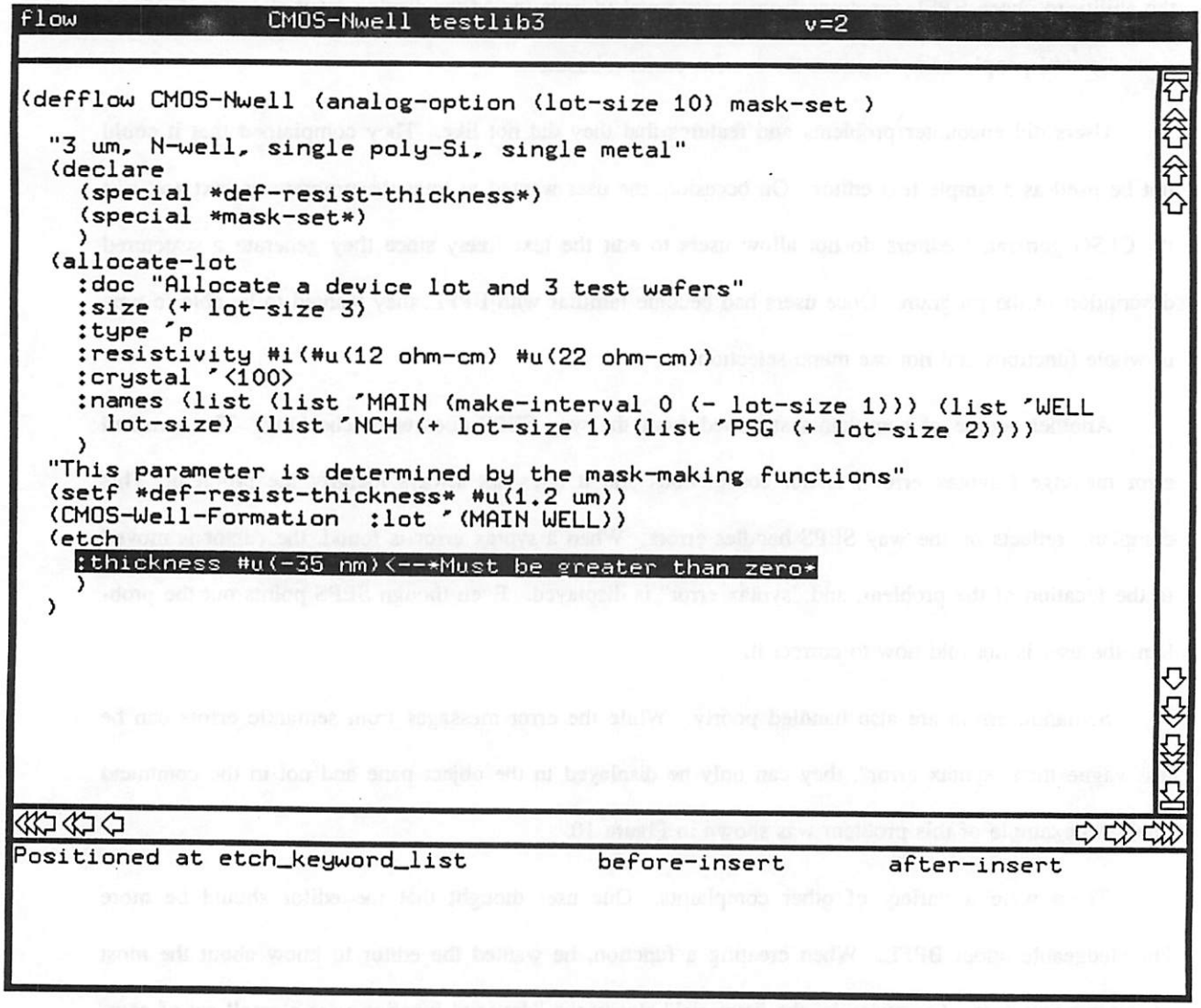

Figure 10: Result of Semantic Error

A number of conclusions can be drawn from these comments. First, SEPS does seem to be useful tool. Second, some problems we encountered are hardwired into CPSG. The user interface and the error message abstractions are inherent parts of CPSG. The on-line help facilities, another source of complaints, need work. Users complained that SEPS did not work like a regular text editor. Finally, the error handling in SEPS is a major flaw. Part of this problem is CPSG and part of the problem is SEPS. The uninformative message "syntax error" is built into CPSG generated editors.

SEPS Implementation

SEPS was implemented in two parts. The first part implements the structured editor. This part consists of approximately 30 files containing 5,500 lines of Synthesizer Specification Language (SSL) that specify the editor. The second part implements the database browser. This part consists of 6 files with 1300 lines of C code and embedded database query language commands, and one file with 300 lines of SSL. CPSG contains hooks for linking the C code into the editor.

From an execution point of view, SEPS is a 1.2 Megabyte object file that contains both the editor and the browser. Figure 11 shows how the editor is generated. CPSG generates the file given an SSL specification and C code as input. The input is processed and linked with CPSG libraries to produce the SEPS executable file.

One reason for using SEPS was to see how easy it would be to produce a structured editor. Implementing SEPS was for the most part straightforward and was done relatively quickly. CPSG has a number of qualities that make it easy to build an editor. A first version of SEPS structured editor took less than two man-months once the input language was learned.

CPSG is powerful enough to describe a complex language such as BPFL. We never encountered a situation where SSL could not describe a language construct. In addition, CPSG allowed SEPS to cope with the parentheses problem quite well. With the "templates" available from CPSG, a SEPS user need never type any parentheses when he is constructing a program.

Editors created with CPSG are extensible. CPSG comes with a number of "hooks" that make it easy to add and change commands. In addition, CPSG has a small toolkit of user interface controls that

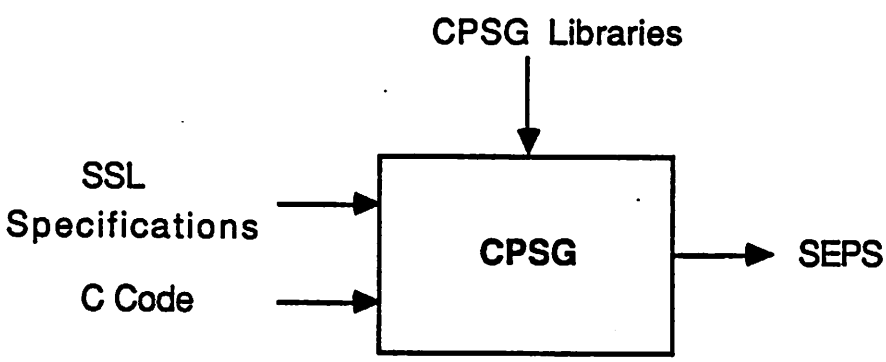

Figure 11: Compiling SEPS

can be used with new commands (e.g., menus and dialog boxes). The combination of hooks and the small toolkit made it easy to build the database browser.

However, some problems were encountered in the implementation of SEPS. The user interfaces provided with CPSG were inflexible, and were not pleasing to users. For example, SEPS uses scroll actuators instead of a scroll bar. Although, several users complained about this interface, nothing could be done about it.

The biggest problem with CPSG was that SSL is unwieldy and hard to learn. Each rule requires a lot of SSL, and learning the language takes time and effort. It took approximately 3 months to learn SSL. Finally, CPSG produced very large executable files. Fortunately, the size does not noticeably affect the performance of the generated editor when adequate hardware resources are available (e.g., workstations with 16 megabytes of memory).

FUTURE DIRECTIONS

This section discusses future work that could be done to improve the editor, the version control model, and the database representation.

SEPS Editor Issues

User feedback indicates several areas in which the editor could be improved. The first area is on-line help. SEPS needs a truly useful help facility. This facility should be context sensitive so that it will provide the most useful information. Second, error messages must be improved. "Syntax error" simply is not adequate.

SEPS Browser Issues

The SEPS version control model is very simple. New versions of specifications are stored in their entirety, and only a simple version sequence is provided. A more sophisticated version model that supports branches and system models is required.

First, version semantics are needed. A version should have semantics associated with it, such as "experimental" or "production." All versions of a routine are not equal, and some routines will be used

for different purposes. Adding a field for "version state" to the BPFL relation would accommodate this additional feature.

Next, branching should be added to the version model. Branching is a way to track separate threads of development. Versions are denoted by version of the form X.Y , where X is the branch number and Y is the branch version. To add this feature, version number processing would have to be changed.

Finally, work must be done to incorporate versions into the database representation. Currently, the entire text of each version of a BPFL routine is stored. This scheme wastes space, particularly when succeeding versions of a routine differ only by a small amount. Several possibilities exist for storing BPFL versions more efficiently. One method is to store only the differences between versions. Another is to take advantage of the structure of BPFL, in a way similar to textual cons-cell representation[9]. A third possibility is to take advantage of a DMBS version facility (e.g., POSTGRES[10]). The access time and storage space trade-offs of these two methods need to be studied further.

General Issues

Several general issues in SEPS needed to be addressed. First, the scheme for dividing the BPFL routine namespace needs to be examined and evaluated more closely. Dividing the namespace into libraries helps, but it is not clear that this method is best. Using a hierarchical name space as in Unix file systems has been suggested. Several schemes should be implemented and evaluated by users.

Finally, SEPS must be exposed to more users. Those who have used SEPS were software developers, not process engineers who are less experienced with software tools. At present, not enough software that uses BPFL exists for SEPS to be useful for typical process engineers. Only when this kind of user interacts with SEPS can the system be fully evaluated.

REFERENCES

1. C. B. Williams and L. A. Rowe, 'The Berkeley Process-flow Language: Reference Document,' *Electronics Research Laboratory Memo No. M87/73*, University of California, Berkeley, CA,

October 1987.

2. G. L. Steele, *Common LISP: The Language*, Digital Press, Bedford, MA, 1984.

3. R. M. Stallman, 'EMACS, the extensible, customizable, self-documenting display editor', *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, Portland, Oregon, June 1981.

4. T. Reps and T. Teitelbaum, *The Synthesizer Generator Reference Manual*, Department of Computer Science, Cornell University, Ithaca, NY, July 1987.

5. J. Gettys and R. W. Scheifler, 'The X Window System,' *ACM Trans. on Graphics*, 2, 79-109 (1986).

6. R. A. Ballance and M. L. Van De Vanter, 'Pan I: An Introduction for Users', *Technical Report 88/410*, Computer Science Division, UC Berkeley, September 1987.

7. L. A. Rowe and C. B. Williams, 'An Object-Oriented Database Design for Integrated Circuit Fabrication', *Proc. 1st Intl. Conf. on Data and Knowledge Systems for Eng. and Manuf.*, Hartford, CT, November 1987.

8. Relational Technology Incorporated, *INGRES/Applications: Applications-By-Forms User's Guide*, Relational Technology, August 1986.

9. M. H. Butler, 'Persistent LISP: Storing Interobject References in a Database' *Ph.D Dissertation*, UC Berkeley, Computer Science Division, November 1987.

10. M. R. Stonebraker and L. A. Rowe, 'The Design of Postgres', *Proceedings of ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 1986.