Evolution towards specifications environment: experiences with syntax editors

M V Zelkowitz

Language-based editors have been thoroughly studied over the last 10 years and have been found to be less effective than originally thought. The paper reviews some relevant aspects of such editors, describes experiences with one such editor (Support), and then describes two current projects that extend the syntax-editing paradigm to the specifications and design phases of the software life-cycle.

software design, environments, specification, syntax editors

SYNTAX EDITORS

Syntax-editing (or alternatively language-based editing) is a technique that had its beginning about 20 years ago (e.g., Emily[1]) and blossomed into a major research activity 10 years later (e.g., Mentor[2], CPS[3]). During the mid-1980s, major conferences were often dominated by syntax-editing techniques[4,5]. Many of these projects, however, have since been terminated or have taken a much lower profile. There are few widely used commercial products that use this technology. Why?

This paper briefly introduces the concept of syntax editing, describes one particular editor, and explains some experiences in using it. It is then shown how the syntax-editing paradigm is powerful but perhaps misapplied in the domain of source-program generation.

Just using a syntax editor for source-code production does not result in significantly higher productivity. By integrating specification generation with this source-code production, however, the author believes that increased productivity can be provided by making more of the life-cycle visible to the programmer. Two extensions to the current environment are described that apply syntax editing within a specifications environment to provide additional functionality over that of standard syntax editors.

With a conventional editor, the user may insert an arbitrary string of characters at any point in a file, and a later compilation phase will determine if there are any errors. With a syntax editor, however, only those choices permitted by the language grammar can be inserted, and the generation of source program and the processing of the program's syntax are intertwined operations. For example, for the statement nonterminal <stmt>, there are only a limited set of statement types that are permitted and only those legal strings can be entered by the user in response to that nonterminal on the screen.

The user interface is a major component of syntax editors. Depending on editor design, syntactic constructs can be specified via a mouse and pull-down menus, function keys on the keyboard, or special editing prompt commands. If the cursor is pointing to the <stmt> syntactic unit and the user specifies the **if** statement, then the text

if <expr> **then**
<stmt>
else <stmt>

will replace <stmt> on the screen. Each nonterminal <...> is considered as a single editing character and syntactic constructs must be added or deleted in their entirety. In essence, the programmer is building the source-program parse tree in a top-down manner.

Pure syntax-editing is a simple macro-like substitution, and such macro substitutions exist in several conventional editors. For example, Emacs and Digital's LBE (Language Based Editor) both permit such substitutions anywhere in a program. Here, however, editors that go beyond simple substitution are being considered. Screen layout is often specified (e.g. unparsing the program tree to a 'pretty-printed' display), semantic information is usually checked (e.g., variable declarations, mixed types), and often the editor is part of an integrated package or environment of editor, interpreter, and debugging and testing tools.

Early on, many advantages of a syntax editor were stated:

- Source-program generation would be efficient as a single mouse or function key click would generate an entire construct.
- Productivity would increase as numerous errors such as missing **begin—end** pairs could not occur and mixed mode expressions would immediately be found by the editor at the point of insertion. Users could more easily use an unfamiliar language.
- Screen layout would be predefined, providing a uniform structure to all programs.

Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA.
Paper submitted: 27 August 1989.
Revised version received: 20 November 1989.

 0950-5849/90/030191-08

- The integrated package of tools enables testing and debugging to proceed more rapidly.

As shall be seen, the last of these reasons does indeed seem to be true; each of the others, however, seems to have a serious drawback as well as the supposed benefit.

As an example, the Support environment, designed by the author, is briefly described as an instance of the integrated syntax-editing genre[6]. It has many of the features implemented in such tools and is the basis for the extensions to specifications described later.

Support design

Support is an integrated environment built to process the CF-PASCAL subset of PASCAL and was used for three years (until the course contents changed) as the programming tool in the introductory programming course at the University of Maryland. It runs on both Berkeley Unix and IBM PC systems.

Design

Major features of Support include the following.

Text input Support uses both the command and function key mechanisms for input. If the cursor (represented by reverse video) covers the < stmt > unit, a menu at the bottom of the screen gives the available choices. For example, to insert an **if** statement, either a response of *.2* or depressing function key 2 (on the PC keyboard) will insert the **if** construct.

Support also permits textual substitution for any syntactic unit. A user can type in an arbitrary line of characters, and an internal LALR parser builds the subtree for that construct. If the root of that subtree is permitted by the current cursor position, then it is attached to the program tree at that cursor position.

Using either input mechanisms, invalid syntax can never be entered. Using the menu for input permits only correct responses, and, for textual input, if the parser cannot resolve the typed-in text to a correct syntactic unit, an error is displayed and the program is not modified.

Windows Horizontal windows dividing the CRT screen are the major interface with the user. Each tool within Support controls its own window, and from two to four windows will typically be displayed at any one time.

Tools Various tools within Support aid in program design and development. The relationship among procedures in a program is handled by the Design window; an interpreter executes partially developed programs and includes features such as variable and statement tracing and breakdown monitoring. Statement trace and statement coverage windows are part of this structure. Data are displayed via the variable trace and the run-time display windows.

As an extension to the textual input mode, a small (i.e., size of screen) text editor called the Character Oriented EDitor (or COED) was implemented. Users insert or modify arbitrary sequences of characters in this window, have the text processed by the LALR parser mentioned above, and then have the text inserted into the program tree at the appropriate place in the program. The user can also pull an arbitrary section of program text into this window for modification. This also gave an easy cut-and-paste feature and the ability to move sections of code around in the program as a means to address some of the syntax-editing deficiencies that turned up.

Language and screen displays The grammar processed by Support (e.g., CF-PASCAL) is defined via an external data file that defines the syntax, some semantics, and screen layout. This feature turned out to be a major factor in allowing Support to be extended for other applications.

Table 1. Background of students

	Semester 1	Semester 2
First university computer course (%)	73	82
Took this course previously (%)	12	9
Took high-school course (%)	59	55
Never previously used computer (%)	26	24
Own microcomputer (%)	49	51

Experiences

Support was used from 1986 until 1989 in Computer Science I by approximately 200 to 300 students each semester. During the first two semesters data were collected from the 543 students that enrolled in the course. The background of the students is summarized in Table 1. As shown, about 75% had previous experience with programming and about half own their own computer.

Based on a 1 to 5 rating scale (1 = poor), students who owned their own computer (and presumably had more experience in programming) rated satisfaction with Support lower than those without their own computer (2.8 to 3.2). More revealing, students rated Support's text-editing capabilities much lower than those of an IBM mainframe also used during the semester (2.7 versus 3.7 for one semester, 3.3 versus 3.8 for the other). The author believes that users with experience with general text editors felt more restricted by the syntax-editing paradigm. On the other hand, novices with no previous experience felt aided by such restrictions.

Students using Support rates its debugging capabilities higher than those available on the IBM mainframe (3.8 versus 3.1 for one semester, 3.0 versus 2.9 for the other). The PC system was also rated as more available compared with the mainframe (3.9 versus 2.8 for one semester, 3.0 versus 2.9 for the other). Other results are presented elsewhere[7].

In summary, syntax editing seems to be viewed as a restriction on program development, but the integrated development and testing environment appears to be desired. A tool that simply develops source text does not seem to produce a large productivity increase. The results here are comparable to those found with other editing environments.

Retrospective

After several years of use and several redesigns and enhancements based on user needs and experiences, the four advantages claimed for such editors can be addressed more clearly. As shall be seen, for most of the advantages, there are some serious problems to overcome.

Efficient generation of source programs

For entering much of the text of a program, this is true, but unfortunately there are enough complications to slow down experienced programmers. For example, the PASCAL **if** statement has an optional **else** clause. Should the editor automatically insert the **else** and have the programmer delete it if not desired, or should it not be included with the corresponding need to add it if wanted? Support chose the latter model, but in either case the editor will be wrong about half of the time.

In Support's case, the screen displays no information about optional syntactic units, so the user needs to know where such units are located. There are two modes of moving forward through a program: the → key moves to the next syntactic unit displayed on the screen, while the enter key is similar but will insert any optional phrases between displayed syntactic units as it moves. In POE's case[8] the opposite occurs. All optional units are displayed initially, and the user must delete them if not specifically wanted.

A more serious consequence is that syntactic units are added top-down, but programmers usually think of algorithms as sequential actions. For adding new statements, there is not much difference between sequential insertion and top-down development of the BNF:

```
<stmt list> ::= <stmt>; <stmt list> |
                <stmt>
```

as both generate statements in a left-to-right manner. Insertion of expressions such as A+B*C, however, essentially means to build the tree in postfix order (e.g., "–", "A", "*", "B", "C"), which is not the natural sequence.

In some environments, such as CMU's Gandalf[9], this top-down linking to the program's parse tree is embedded in the user interface; in Support's case, however, the LALR parser mentioned earlier was added. Straight text will be parsed and entered in its true infix format. The COED editor within Support was a valuable extension that permitted programmers to add small sections of program text (up to 22 lines of input) without violating the basic top-down nature of program generation in a syntax editor.

Early detection of syntax and semantic errors

While true, this is not much of a benefit if its consequences are considered. Experienced programmers generally do not make many syntax errors as they enter text, although novices do. (This might explain Support's greater popularity among non-programmers than among programmers.)

There are cases where this supposed benefit is actually a hindrance. If an experienced programmer thinks of a sequence of code to enter and makes an error in input, a standard editor will ignore the error and continue entering data. After finishing entering code, the programmer can fix the earlier problem. With a syntax editor, however, only correct syntax can be entered. The system will usually halt and beep until corrective action is taken. Thus there is a disruption in a train of thought where some deep semantic issue needs to be put aside (and forgotten?) to fix some simple syntax.

Looking at both of these reasons, as languages get more complex (e.g., ADA) syntax editing might make more sense, but in relatively simple languages, like PASCAL and C, there seems to be few benefits. There is little experience with such editors for complex languages. Arcturus[10] is a prototype of an ADA editor, but it was not made commercially available.

Screen layout is predefined

This is also true, but again the predefined layout might not be what the programmer wants in all cases. It certainly helps the novice generate nicely indented listings, but as the programming task grows more complex, the number of special cases increases.

The placement of comments seems to pose a problem with all such editors. Comments are generally outside the language's defining BNF. Where do they appear in the listing? In Support they are tagged before the defining nonterminal. This works in some cases, but not all.

Uniform debugging and testing tools

This again is true, but a syntax editor is not needed for this feature. An integrated framework and data repository are needed for a source program. The current interest in CASE (computer-aided software engineering) tools exemplifies this, and Support is simply a CASE tool with a syntax editor for a base.

In summary, the experiences with Support are by no means unique and closely mimic experiences others have had with syntax editors. For example, Mentor, initially developed about eight years earlier at INRIA, has had a similar pattern of development and use[11]. Similar to experiences with Support:

- Novices used menus but experienced programmers rarely did.
- Experienced programmers wanted the full-screen Emacs editor for textual input and modification (providing functionality similar to the COED editor described here) using automatic parsing and unparsing of the Mentor input.
- Switching between Mentor and Emacs was difficult due to the inherent problems in placement of comments. On the other hand, Mentor was a powerful source-code maintenance system due to the integration of many program analysis tools for obtaining semantic information about a program. But just as in Support's case, such tools are mostly a function of

Mentor being an integrated environment and not simply an editor.

In conclusion, the drawbacks seem to be as serious as the advantage in syntax editing, which probably explains their lack of growth and popularity since the early '80s. As a final comment, source-code development is often stated as 15% of total life-cycle costs. Even if the editor reduced coding time to zero, that would still mean a productivity improvement of only 15%. Industry is looking for more than that.

SPECIFICATIONS

The previous discussion indicates that while syntax editing of source programs is a powerful technique, it probably has minimal effect on programmer productivity. As requirements, specification and coding take up to 75% of the costs to develop a system, however, improving those phases of the life-cycle might have a more dramatic impact on productivity. In addition, a mechanism to improve the flow between specifications to design to code would probably lead to fewer interface errors, hence decreasing the effort needed in testing and further increasing improved productivity.

For coding source programs, there are several programming techniques: procedural languages (e.g., PASCAL, C, ADA, COBOL), applicative languages (e.g., LISP, PROLOG), object-oriented programming (e.g., SMALLTALK, C++), etc. Their relative strengths and weaknesses for specific applications are fairly well established. For specification of a program, there are also several models (e.g., axiomatic, denotational, algebraic, functional); however, as yet there is no clear consensus as to which is most effective and how each applies to different application domains. This is still very much an open research question, with many ongoing projects studying various specification strategies.

Given the powerful syntax editing paradigm and its relative inability at improving source-code generation, the author decided to investigate it within a specification domain. After all, most specification languages have a syntax and semantics more complex than most programming languages, and some anecdotal data do seem to indicate that programmers would prefer syntax editors for sufficiently complex languages.

As stated previously, Support processes a language defined by an external grammar file, and it is constructed as a set of independent tools, each writing to virtual windows that are mapped to the actual computer screen. By modifying this grammar and by adding new support tools, Support becomes an interface 'shell' for a series of integrated environments. It can be used as a language processing meta-environment by providing the capabilities to read input, parse text, build parse trees, and manipulate multiple windows simultaneously. Using Support, two such extensions were developed that are described here: AS* (based on algebraic specifications) and FSQ (based on functional specifications).

```
(1)   sort sequence [sort something] is
(2)   constructor
(3)       epsilon;
(4)       cons : something, sequence;
(5)   operation head : sequence → something is axiom
(6)     head(epsilon) == ?;
(7)     head(cons(X,Y)) == X;
(8)   operation count : sequence → integer is axiom
(9)     count(epsilon) == 0;
(10)    count(cons(X,Y)) == 1+count(Y);
(11)    end;
```

Figure 1. Example of sequence specification

AS* for executable specifications

An algebraic specification is a series of axioms that link together the operations that can be applied to an abstract data type. As an extension to the Support environment, a specifications extension based on these algebraic axioms has been defined.

An AS* specification contains three features:

- a set of sort names that define new abstract objects and their constructors
- a signature, which defines a set of defined operations for manipulating the abstract objects
- a set of oriented equations (or axioms) that relate the defined operations and constructors to each other

Figure 1 gives a simple example of a specification for a sequence. Line (1) specifies that a class of objects of sort (i.e., type) 'sequence' is being defined and indicates that the new object will require as a parameter a sort 'something' that will be specified in a later binding. A generic class of sequences that will be instantiated by this later binding to 'something' is being defined. Lines (2)—(4) define the two constructors needed to create an object of this sort: 'epsilon' to return the empty object of sort 'sequence' and 'cons', which takes an element and a sequence and returns a new sequence with the element in it. The functionality of each constructor is given after its name with the sort name 'sequence' implied as last (e.g., 'epsilon' returns an empty 'sequence' and 'cons' requires a 'something' and a 'sequence' and returns a 'sequence'.) 'Epsilon' initializes objects of this sort and 'cons' creates new complex objects.

This object is manipulated by means of a set of defined operations. In this simple example, operations 'head' and 'count' are given with their signatures on lines (5) and (8). They are defined by the rewrite rules (axioms) on lines (6)—(10). 'Head' says to return the element last included into the sequence by the 'cons' function, while 'count' returns 0 for 'epsilon' (i.e., an empty list) or 1 plus the size of any non-null list with the first element removed. As can be seen, the formal definitions of each function includes recursive algorithms for computing its value by reducing any complex object to a finite set of applications of the constructor functions. The '?' on line (6) is equivalent to an error condition, and the implementation stops execution and issues an error message when

this occurs. (That is, it is illegal to take the 'head' of an empty list.)

For example, the list <X,Y,Z> is created by the construction:

```
cons(X,cons(Y,cons(Z,epsilon)))
```

and the operation 'count' uses this construction, as in:

```
count(<X,Y,Z>)=
1+count(<Y,Z>)=
1+1+count(<Z>)=
1+1+1+count(<epsilon>)=
1+1+1+0=
3
```

The use of the Knuth—Bendix algorithm[12] defines a proof of adequacy of the resulting algebraic equations by showing the equivalence of supposedly equal terms to the same ground (i.e., constant) terms. As the Knuth—Bendix algorithm is based on an ordering transformation from one term to a 'simpler' term, however, the algorithm defines an operation that can be 'executed' and proven to terminate. Therefore, any set of axioms that is 'Knuth—Bendix' can be transformed mechanically into a series of transformations that can be executed in some programming language, in this case PASCAL.

Similar to Larch and Larch/CLU[13], AS* specifications are independent of the underlying programming language and must be defined relative to any concrete language. Libraries of generic specifications can be used to form the basis of a reuse methodology where the generic specification is refined to an explicit specification in a specific programming language by binding the generic sorts to specific programming language types. In this case PASCAL is considered as the implementation vehicle, so to create ASPascal, the extension to PASCAL that contains AS* specifications, a link between a PASCAL object and an AS* sort must be indicated.

An explicit specification is created by a refinement of a generic specification via the **use** clause, as in:

```
sort intsequence is
    use sequence [integer]
    end;
```

which refines the generic sort 'sequence' given earlier and indicates that a new sort 'intsequence' is created by modifying 'sequence' with a binding of PASCAL integers to the free sort 'something' of Figure 1. The operations 'head' and 'count' in 'sequence' become 'intsequence_head' and intsequence_count' in the new sort, although the actual mapping to their new names is handled automatically and of no concern to the programmer.

The interface assumption is made that an explicit sort specification

```
sort newsort is ...
```

is equivalent to the PASCAL type declaration

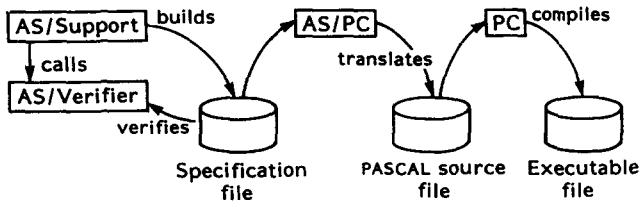

Figure 2. AS toolset*

```
type newsort = ...
```

The primitive PASCAL scalar types (char, Boolean, integer, real) may all be used in abstract sort definitions, and any explicit sort may also be used in a refinement. Thus

```
var A: intsequence;
```

simply creates a PASCAL variable *A*, which is of type 'intsequence'.

The power of this system is in alternative bindings. For example, real sequences could be created as

```
sort realsequence is use sequence [real] end;
```

Similarly, a sort such as a 'book' could be used to create a type 'library' as

```
sort library is use sequence [book] end;
```

As stated earlier, syntax editors might have greater use with more complex source languages, and the integrated tool set forms an effective basis for a CASE tool. Therefore, a prototype AS* system was built on top of the existing Support environment. Figure 2 represents this initial system that has been constructed. The four components are as follows.

AS/Support

AS/Support is a modification to the Support environment described earlier, which provides text-editing capabilities for creating specifications. It is also the control module that invokes the verification tool. AS/Support first checks axioms within operations for syntactic consistency. Because of the language-based design of the underlying environment, only syntactically correct axioms with the syntax

```
operation_name(<expression_list>)= =<expression>
```

can be entered by the user. After the user builds a sort, AS/Support formats the sort syntax into an appropriate format suitable for PROLOG and invokes AS/Verifier as a subprocess. AS/Verifier reads these axioms and checks executability. After passing all executability checks through AS/Verifier, the user may save the ASPascal program in a library for later translation by AS/PC or for later incorporation into another ASPascal program.

In case of failure, the causing axiom, if it can be determined, is highlighted to allow the user an interactive mechanism to change the specifications.

AS/Verifier

AS/Verifier, a PROLOG program, is called by AS/Support and verifies the set of axioms via the Knuth—Bendix algorithm. In general the axioms need to be a noetherian term rewriting system, and, if possible, AS/Verifier makes this determination. Of course, as the general problem is undecidable, in some cases the results are inconclusive. In any case, after one pass through the axioms, AS/Verifier will either succeed or indicate which axiom is currently failing so that the user may modify the definition and try again. As stated previously, if any error is found, an appropriate message is relayed back to AS/Support and displayed to the user.

For example, the 'sequence' definition of Figure 1 will be converted to the following clauses and passed to AS/Verifier:

```
as←sort (sequence, [epsilon, cons, head, count]).
function (1, epsilon, [], sequence).
function (2, cons, [something, sequence], sequence).
function (3, head, [sequence], something).
function (4, count, [sequence], integer).
axiom (5, head (epsilon), "?").
axiom (6, head (cons(x,y)),x).
axiom (7, count (epsilon),0).
axiom (8, count (cons(x,y)), 1 + count(y)).
```

(as←sort is the internal name for a new 'sort'.) The Knuth—Bendix algorithm either shows convergence of the axioms or indicates additional axioms that are needed; it may not indicate, however, when sufficient axioms have been added in the case of not converging rapidly enough (the usual problem with undecidability results). In this case, AS/Verifier does a single pass over the axioms and then terminates, indicating where the problem is with the axioms.

AS/PC

AS/PC is the translator, written in YACC, that converts specifications into standard PASCAL source programs. The code generally consists of a sequence of **if** statements, each checking the validity of the left-hand side of the axiom before executing the Knuth—Bendix reduction.

PC

PC is the standard system PASCAL compiler. At this point, the specifications have been converted to standard PASCAL, and any comparable compiler can be used for compilation and execution.

Specifications appear in programs as function calls in the host programming language. To translate such calls, it is necessary to determine, for each function reference, which explicit specification is being used. Thus a reference to 'head(thing)' where 'thing' is an 'intsequence' is translated to a call to 'intsequence_head(thing)', while 'head(realthing)' will result in 'realsequence_head(realthing)' for variable 'realthing' of sort 'realsequence'. (The details of the AS* implementation appear elsewhere[14].)

It should be clear that this translation does not result in a particularly efficient implementation; as a specifications or prototyping tool, however, efficiency is not its overriding purpose. The goal is to provide easily a correct extension to an existing system and to provide a verification tool, e.g., an oracle, that can be used as a test against an eventual efficient solution to the problem.

FSQ for software reuse

In the previous section, AS* was described as an environment based on an algebraic specification model for program specifications. Support is also being applied using the functional correctness model[15]. In this model, both a program and a specification are viewed as functions, and techniques have been developed to determine if both represent the same transformation of the data. This model of program development is briefly summarized and how Support is modified to aid in this process is then demonstrated.

Functional correctness

A specification f is a function. A box notation [...] is used to signify the function that a given string of text implements. If character string α represents a source program that implements exactly f, then $[\alpha] = f$, and it is stated that α is a solution to f.

Sequential program execution is modelled by function composition. If a sequence of statements $s = s_1;s_2; \ldots s_n$, then $[s] = [s_1] \; o \; \ldots \; o \; [s_n] = [s_n] \; (\; \ldots \;)[s_1])) \; \ldots \;)$. Using techniques from denotational semantics, each statement s is a function from a program state to another state. Each program state is a function from variables to values and represents the abstract notion of data storage. Symbolic trace tables are used to derive the state functions for **if, while**, and assignment statements.

Program design is accomplished by converting a specification function f, written in a LISP-like notation, into a source program α, and then showing that $[\alpha] = f$. The specification f is called the abstract function and the program α the concrete design. Given this functional model, the basic theorem for functional correctness[16] can then be proved. Program p is correct with respect to specification function f if and only if $f \subseteq [p]$.

This model can be applied to three separate activities:

- Program verification. If f is a function and if p is a program, determine if they are the same function, i.e., $[p] = f$, or more generally $f \subseteq [p]$.
- Program design. If f is a function, then develop a program p such that $[p] = f$.
- Reverse engineering. If p is a program, then find a function f such that $[p] = f$

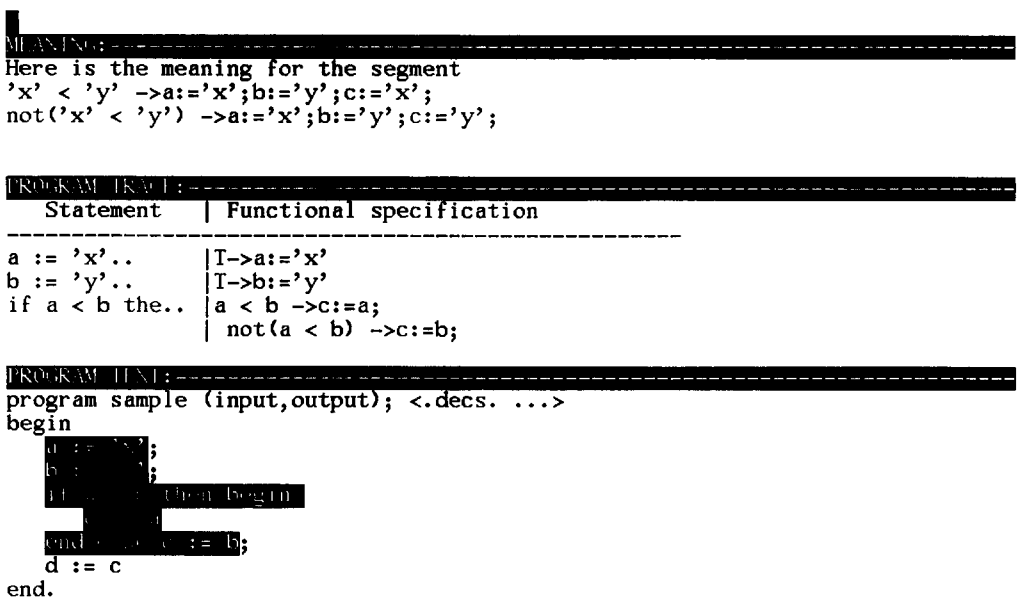

Figure 3. FSQ derived meaning for program fragment

FSQ extensions

The use of existing program fragments when developing a new program is one technique being studied for improving programmer productivity. Often, however, it is first necessary to determine exactly what these program fragments or procedures do. As formal specifications are rarely used, and documentation is generally quite inadequate, programmers are reluctant to use an existing procedure written by another from some previous project since the mental effort to truly understand that procedure is quite high.

To study this problem, the Support environment was extended with a new tool, Function Specification Qualifier (FSQ), to aid this process of determining the specifications for an existing component of a system. FSQ-1, a first prototype of this tool, is described.

FSQ is an additional tool to the basic CF-PASCAL programming environment in Support and works as follows:

- A programmer either builds a program using Support (and hence uses FSQ as a verification tool) or else reads one from the file system using the LALR parser internal to Support to build the parse tree (making FSQ a reverse engineering tool).
- The cursor is moved over the section of program that needs to be verified and FSQ is invoked via the command *.fsq*.
- FSQ symbolically executes each statement and determines its meaning. This is relayed back to the user, who either accepts this meaning (e.g., its specification) or manually simplifies it to another meaning.
- The derived meaning is stored in the Support syntax tree. If any part of a program is symbolically executed and already has a derived meaning, then that meaning will be used without further analysis. This meaning can then be carried along as part of the file system repository information on that object. Future users of that object will not have to derive the meaning again.

  Over time, more and more procedures in the system repository will have such derived meanings, making it more efficient to reuse such objects frequently.

Figure 3 shows a sample execution of FSQ. The top meaning window shows the desired result from the execution, the middle program trace window indicates each partial result, and the bottom window highlights the section of the source program that is under study.

FSQ executes over the covered portion of Figure 3 as follows:

- (1) For a: = 'x' the system derives the conditional T → a: = '*x*'. (This is similar to the LISP 'cond' and means 'True implies a: = '*x*'.')
- (2) For b: = 'y' the system derives the conditional T → b: = '*y*'.
- (3) For c: = a the system derives the conditional T → *c: = a.*
- (4) For c: = b the system derives the conditional T → *c: = b.*
- (5) For the **if** statement, FSQ combines steps (3) and (4) to produce:

  not (a < b) → c: = b;
  (a < b) → c: = a

- (6) Finally, for the entire sequence, FSQ combines the results from steps (1) through (5) to produce the function described in Figure 3.

Note that this process is simpler than general program verification (and potentially less accurate) as the programmer can override the system and insert arbitrary definitions. For example, in the program of Figure 3, the user, in the process of deriving the meaning of the **if** statement at step (5), could have either substituted the correct simplification

c: = min (a,b)

or any other correct or incorrect expression for the **if**. Thus the user must trade off between 'absolute' but extremely difficult correctness using a verifier and a system like FSQ, which performs efficient, but possibly imperfect, verification. The tool is truly interactive, with FSQ performing all the tedious bookkeeping procedures, and by having the user required provide for the creative program derivation activities. This avoids the general undecidability issues of general verifiers and permits the data-intensive functional verification mechanism to be used practically.

CONCLUSIONS

In this paper the basic features of syntax-directed editors have been described and possible reasons why such editors have not become more popular outlined. The author believes that their benefits do not increase productivity sufficiently to compensate for their deficiencies. Source-code generation, although labour intensive, is not a major cost factor in system development.

However, syntax editors can provide a consistent interface when system specification is integrated with source-code generation. To experiment with this, two specification projects have been described as extensions to an existing PASCAL development environment. In these extensions both algebraic specifications and functional correctness models of development were applied as extensions of automated tool support. Further work is needed to test the eventual applicability of this form of environment.

ACKNOWLEDGEMENTS

This work was partially supported by Air Force Office of Scientific Research grant 87-0130, Office of Naval Research grant N00014–87–K–0307, and NASA grant NSG–5123, all to the University of Maryland. Individuals who have contributed include: for Support: Bonnie Kowalchack, David Itkin, Jennifer Drapkin, Michael Maggio, and Laurence Herman; for AS*: Sergio Antoy (of Virginia Tech), Sergio Cardenas, Paola Forcheri and Maria Teresa Molfino (of I.M.A., Genoa, Italy), Stuart Pearlman, and Lifu Wu; and for FSQ: Victor Basili and Sara Qian.

REFERENCES

1 **Hansen, W J** 'User engineering principles for interactive systems' in *Proc. Full Joint Comp. Conf.* Vol 39 (1971) pp 523–532

2 **Donzeau-Gouge, V, Kahn, G, Huet, B, Lang, B and Levy, J** 'A structure assisted program editor: a first step towards computer assisted programming' in *Proc. Int. Computer Symp.* North-Holland, Amsterdam, The Netherlands (1975)

3 **Teitlebaum, T and Reps, T** 'CPS: the Cornell Program Synthesizer' *Commun. ACM* Vol 24 No 9 (1981) pp 563–573

4 *Proc. ACM SIGPLAN Symp. Language Issues in Programming Environments* Seattle, WA, USA (June 1985)

5 *Proc. ACM SIGSOFT Practical Software Development Environment Conf.* Pittsburgh, PA, USA (April 1984)

6 **Zelkowitz, M V** 'A small contribution to editing with a syntax directed editor' in *Proc. ACM SIGSOFT Practical Software Development Environment Conf.* Pittsburgh, PA, USA (April 1984) pp 1–6

7 **Zelkowitz, M V, Kowalchack, B, Itkin, D and Herman, L** 'A support tool for teaching computer programming' in **Fairley, R and Freeman, P (eds)** *Issues in software engineering education* Springer-Verlag, Berlin, FRG (1989) pp 139–167

8 **Fischer, C, Pal, A, Stock, D, Johnson, G and Mauney, J** 'The POE language-based editor project' in *Proc. ACM SIGSOFT Practical Software Development Environment Conf.* Pittsburgh, PA, USA (April 1984) pp 21–29

9 **Habermann, N and Notkin, D** 'Gandalf. Software development environments' *IEEE Trans. Soft. Eng.* Vol 12 No 12 (December 1986) pp 1117–1127

10 **Standish, T and Taylor R**, 'Arcturus: a prototype advanced Ada programming environment' in *Proc. ACM SIGSOFT Practical Software Development Environment Conf.* Pittsburgh, PA, USA (April 1984) pp 57–64

11 **Lang, B** 'On the usefulness of syntax directed editors' in *Proc. IFIP Workshop on Advanced Programming Environments* Trondheim, Norway (June 1986) pp 45–51

12 **Knuth, D and Bendix, P** 'Simple word problems in universal algebras' in *Computational problems in abstract algebra* Pergamon Press, New York, NY, USA (1970) pp 263–297

13 **Wing, J** 'Writing Larch interface specifications' *ACM Trans. Prog. Lang. Syst.* Vol 9 No 1 (1987) pp 1–24

14 **Antoy, S, Forcheri, P, Molfino, T and Zelkowitz, M** 'Rapid prototyping of system enhancements' in *Proc. 1st Int. Conf. System Integration* (April 1990)

15 **Gannon, J D, Hamlet, R G and Mills, H D** 'Theory of modules' *IEEE Trans. Soft. Eng.* Vol 13 No 7 (July 1987) pp 820–829

16 **Mills, H D, Basili, V R, Gannon, J D and Hamlet, R G** *Principles of computer programming: a mathematical approach* Allyn Bacon (1987)