

A Structure Editor for VHDL

*Matthew P. Phillips,
Defence Science and Technology Organisation,
Adelaide, South Australia.*
*Peter J. Ashenden,
Department of Computer Science,
University of Adelaide,
Adelaide, South Australia.*

Abstract

While VHDL is a powerful tool for the design and verification of digital electronics, this power is accompanied by great language complexity. This complexity can only be reduced by the use of intelligent tools capable of watching the design as it evolves, catching errors and speeding input. This paper describes the development of such a tool, a structure editor for VHDL.

1. Introduction

Hardware description languages (HDL's) have become an invaluable tool for the design and testing of electronic hardware systems. HDL's have many advantages over the more traditional schematic design technique but, most importantly, a HDL incorporates the behaviour of the hardware along with its structure. HDL models are also useful in that they allow the designer to verify a high-level behavioural description of the hardware and then gradually design its structural analogue. Furthermore, a HDL design is generally far easier for others to comprehend.

However, there is a price to be paid when using a HDL, in that the engineer must come to terms with a new and complex tool. To give the most flexibility, HDL's often require many syntactical constructs and complex semantic rules to regulate their combination. The designer must also face the task of programming and debugging the design, two new phases of development that require time and experience. Furthermore, the engineer may want more than just a view of the HDL text. It may be necessary to view the end-product structural design as a schematic outline and to have the hardware simulation displayed within this framework. Thus what is needed is more than just a standard text-editing system, but an integrated HDL programming environment, with an editor that can assist the designer in writing and debugging the

design, and the ability for the HDL code to be viewed and debugged in different ways.

1.1 The project

The goal of this project was to produce the first stage of an integrated programming environment for the VHSIC Hardware Description Language (VHDL) [10, 17, 18] – a VHDL-specific editor. The editor's task is to speed the design of VHDL models by both accelerating code entry and formatting and detecting syntactical errors as the model is created.

In order for this to occur, the editor must have a knowledge of VHDL syntax and formatting conventions. It must also be able to use this knowledge to provide intelligent editing commands and context-sensitive template creation for the main VHDL syntactical constructs. Furthermore, it must be able to share its output with other applications and preferably be able to operate in tandem with them. This sort of language-specific editor is often called a *structure editor* or *syntax-directed editor*. Therefore the goal of the project is to create a structure editor for VHDL that can be integrated with other design entry tools and with other applications.

In the remainder of this section, a survey of structure editor technology is presented. Section 2 discusses the editor environment (MultiView), while Section 3 discusses the editor implementation itself. Section 4 presents a conclusion to this project and some ideas for further work.

1.2 Structure editors

The most widely used programming editors today are purely text-oriented. Such textual editors generally make no use of the structured nature of programming languages for which they are used. Some text editors, such as Emacs [14], allow extensions that can provide a degree of language-specific editing, but these extensions tend to be based on ad hoc pattern matching

rules and can be unreliable. They also provide no automatic syntax checking, and indeed may go completely awry when presented with incorrect syntax.

A structure editor generally parses the program code as it is entered and stores the result as an abstract syntax tree (AST). The editor then allows the user to edit this tree within the syntactic constraints of the programming language. It may also allow the user to edit the program as text if required. The editor's internal AST representation of programs raises two issues that are not present in textual editors: the editing and the display of an abstract syntax tree.

1.2.1 Editing

A pure structure editor only manipulates AST's via tree create, modify and delete operations that are allowable within the syntax of the language. This sort of editor faces the challenge of providing the user with an easy way of controlling such tree operations. The most common way for the user to edit AST's in such an editor is by *selection and expansion*, which is a process of successively deriving non-terminals in the AST until the desired result is achieved. This is a procedure resembling, in some ways, stepwise refinement. However, as Allison points out, this is not an intuitive way for most people to enter programs [2]. In fact, while people tend to think of programs as text with underlying structure, their editing preference seems to be text-oriented. The selection/expansion approach has three main disadvantages:

- it is less straightforward than text entry,
- it requires more steps than simply typing the constructs
- it forces the programmer to know the formal syntax of the language in order to know which branches of the syntax tree to expand.

Another problem with a strictly tree-oriented approach is that can make it more difficult to modify existing code. For example, the user may wish to make the change to a *loop* construct as illustrated in Figure 1. In a text editor, the user would simply make the changes, with the program passing through some inconsistent state in the process. This sort of inconsistency cannot happen in a strictly tree-based editor

```
while <expr> loop      loop
  <statements>        <statements>
end loop;              →  exit when not <expr>;
                       end loop;
```

Figure 1. Converting between two loop structures

because there is no way to represent the intermediate state as an AST.

A partial solution is to provide tools that make these sorts of changes automatically. If well-designed, such tools might even speed up some changes, but it does not seem possible to foresee and provide for all the editing operations a programmer might need.

Another solution is to provide the ability cut and paste subtrees to and from a number of "clipboards". In this sort of editor, to make the changes in shown in Figure 1, the user would cut the *<expr>* and *<statements>* subtrees into two separate clipboards and then delete the entire *loop* subtree. Then the user would create an appropriate new *loop* template and paste the old subtrees over the placeholders. This is adequate for this example, however for more complex changes this approach still requires considerably more effort than a few text cut-and-paste operations.

In practice, a simple and effective solution is to provide the ability to edit selected portions of the program as text and then reintegrate the changes into the tree when the construct is fully entered. Editors using this approach are termed *hybrid structure editors* and are by far the most common. Examples of successful hybrid structure editors include the Cornell Program Synthesizer [15] and CAPS [19]. An interesting alternative approach to hybrid editing is to provide intelligent subtree search and replace as in SED [2].

1.2.2 Display

A structure editor must have some means of showing its internal AST representation of the program in textual form. Unfortunately, the process of parsing usually means that the original textual layout of the program is lost. The layout can be recreated automatically by traversing the tree and using *unparsing* rules to govern how given subtrees are displayed as text. This has the advantage of providing a consistent layout for any code processed with the editor.

Most structure editors provide the user with customizable unparsing and some also provide ways to *elide* constructs below a certain level. This provides a useful way of seeing the overall structure of the code in much the same way as might be shown with an outlining tool. Most structure editors can also display *handles* for required or optional constructs and allow the user to quickly select and expand them.

1.2.3 Structure editor survey

A brief survey of structure editors is presented in this section, outlining various features and approaches that have been tried over the past 20 years.

One of the most well-known, and powerful, structure editing environments is the Cornell Program Synthesizer (CPS) [15]. This environment provides both structured and textual editing at user-defined levels. It also provides the ability to catch static semantic errors at the time of entry and provides an integral interpreter and debugging system. The Synthesizer Generator [13] is a related development designed to be easily customised for different languages.

Emacs [14] and Z [20] both provide pseudo-structured editing via suites of language-specific extensions to their text-editing modes. Although this type of "structured" editing is widely used, it has a number of drawbacks, including the inability to perform syntax checking.

There exist many editors with specific enhancements to the usual structure editing features. The Pascal-Oriented Editor (POE) [4] provides an error-correcting parser that can dynamically detect and correct entry errors. The Display-Oriented Structure Editor (DOSE) [6] allows changes to its language specifications to be made on the fly and can even be used to edit its own language specification language. Finally, SED [2] is a pure structure editor which allows trees to be modified using a powerful tree match-and-replace system.

There also exist integrated programming environments such as MultiView (see section 2.1) and PECAN [12] which provide structured editing in tandem with cooperative tools such as debuggers, declaration editors, flow-chart views, etc.

2. Implementation environment

The initial implementation plan for the project was to build the VHDL structure editor from scratch, using a commercial package to handle the storage of VHDL modules in a central database. The editor was to have been written in C++ and use the Motif toolkit for the user interface. Many ideas for the editing interface were discussed and research into existing structure editors was undertaken. It was decided that the goals for the project could be faster achieved by building a VHDL version of the MultiView Integrated Software Engineering Environment [3, 7, 8, 11]. MultiView would provide the integrated environment, while the structure editor would be a VHDL TextView.

2.1 MultiView

MultiView is a language editing environment designed to support multiple views of a program. The MultiView system provides an integrated environment where program units may be viewed and edited concurrently in different ways. For example, program text may be entered within a textual view and its overall structure viewed graphically within another view. MultiView is designed to be language non-specific, and a number of tools are provided to accelerate the process of specialising MultiView for different languages.

A MultiView system consists of a database, a database server and one or more views. The relationship between these components is shown in Figure 2. The server and views are implemented as separate UNIX processes and communicate via a message-passing interface. The messages are structured using a Protocol Specification Language (PSL), which is a language designed to easily define the communication structure between a view and the database server. Most views retain a cached copy of the unit they operate on to reduce the load on the database server and to accelerate editing operations. A sophisticated caching control system is provided by the database server to ensure cache coherency.

MultiView has recently undergone a major revision of its communication subsystem [9], and a number of new tools have been created to allow easier implementation of new views and adaptation to different languages.

Program units are parsed by the MultiView system and stored in abstract syntax tree form (AST) within the database. The units can then be viewed or edited by one or more views which interact with the database server to store and retrieve units from the database.

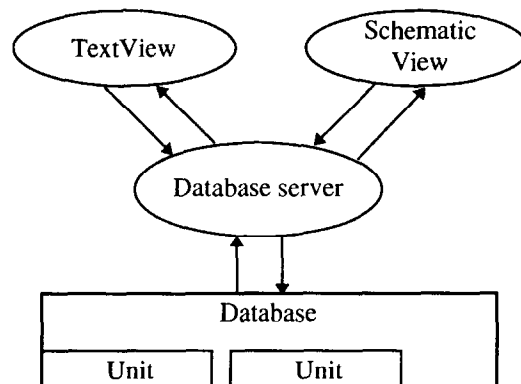


Figure 2. MultiView organisation

An important feature of MultiView is that more than one view of a given unit may be used at one time. The database server ensures that changes made in one view are reflected in other open views and that the units remain mutually consistent.

A version of MultiView specialised for VHDL provides an ideal way for a VHDL structure editor to combine with other views and debugging aids. A designer may build a model within TextView and then switch to viewing and editing in a schematic view. Whichever view the designer finds easiest for a particular operation can be used without leaving the environment and without adversely affecting other open views.

2.1.1 The advantage for this project

From its inception, the aim of this project was to produce a VHDL structure editor that can operate in tandem with other tools. In particular the editor would be complemented, and greatly enhanced by, a VHDL *schematic view*. Such a view would display the components of a VHDL model and their connections to each other in graphical network format familiar to most engineers. This sort of view and its merits are discussed more fully in section 4.1.3.

The advantage of the MultiView system is that it allows the seamless integration of later views with already existing ones. It also allows views to be extended to display dynamically changing data, or to integrate data from another source within a view. For instance, an extended TextView might interface to a VHDL simulation environment to allow the user to dynamically display and edit model variables from within TextView. Similarly, a schematic view may be extended to display a model's simulation graphically.

2.1.2 Abstract syntax

Abstract syntax trees are a tree representation of structured text. They provide a notation for programs that is independent of the actual concrete syntax used to textually represent the program. Each node in an AST corresponds an *operator* within the language. Generally there is one operator for each abstract action permitted by the language.

Each AST operator is defined to be of a particular *sort* or *phylum*. The sort of an operator is analogous to the *type* of a variable in a high-level programming language. Operators are logically grouped by their sort and they may only appear as children of a tree node where the "slot" for that child has the same sort. The number of child nodes a node has is termed its *arity* –

depending on the particular AST model, the node may either have fixed or variable arity.

As a simple example, the expression "1 + 2 * 3" might be parsed to the AST shown in Figure 3. In this example each node in the AST shows the sort of node at the top and the actual operator at the bottom. The leaf nodes have a literal value string rather than an operator.

2.2 TextView

TextView [3, 7, 11] is an X Windows-based textual view for the MultiView environment. It provides the following facilities:

- both structured and textual editing operations,
- automatic code formatting,
- immediate feedback on syntax correctness.

TextView is a hybrid structure editor. Text may be entered in the main text pane and then parsed at the user's command. Parsing errors are displayed in a window below the text editing pane – currently only syntax is checked. Program constructs may either be entered directly within the code pane, or inserted by selection from a list of currently valid constructs at the left of the code pane. The user can choose to show either the entire text of the program, or have some constructs shown as placeholders.

The VHDL TextView main window is illustrated in Figure 4. The cursor appears as a black rectangle and the current subtree within which it lies is indicated by underlining the corresponding text. A list of valid constructs for that subtree is displayed to the left of the pane where the user can readily access it. If the user clicks on one of these buttons, the current operator that the cursor lies on is replaced with the operator associated with the button. The "prepend" and "append" buttons at the top left of the screen allow the user to insert a new list element either at the beginning or the end of the current list.

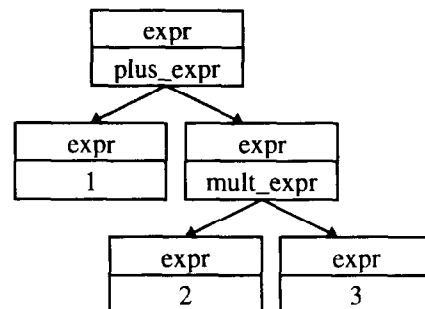


Figure 3. Abstract syntax tree for a simple expression

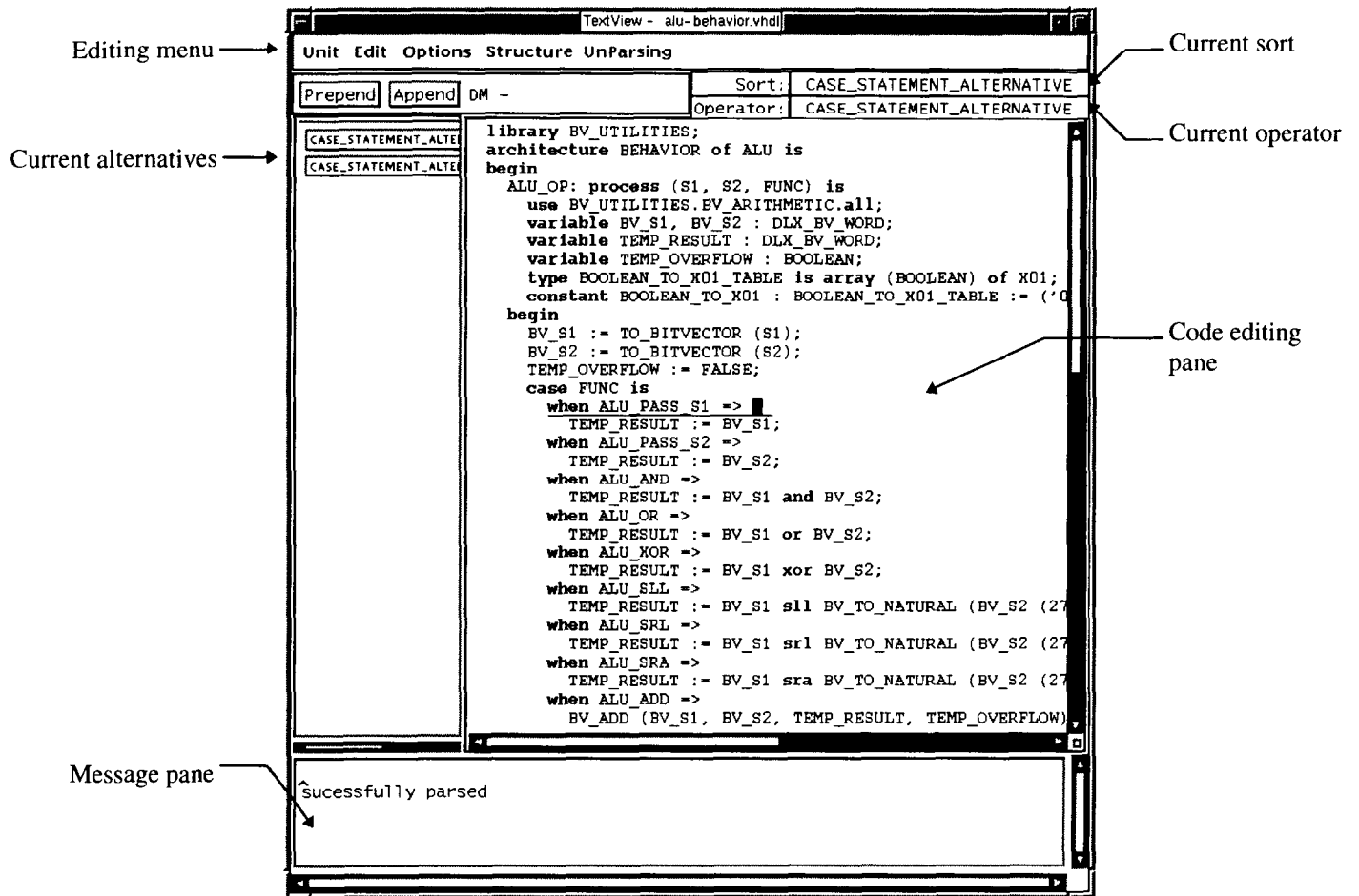


Figure 4. The TextView main window

When the user has finished editing a construct, a parse command is invoked. At this point any text that has been changed is sent to MultiView for parsing. If the parse is successful, the result is unparsed and displayed on the screen, giving the user immediate visual feedback. Any parse errors are displayed in the message pane at the bottom of the screen.

3. VHDL MultiView

In order to build a version of MultiView for a given language it is necessary to both describe the formal grammar of the language and how to build an AST for each production of that grammar. This description is done in a meta-language called the Language Specification Language (LSL), a compiler for which is provided by MultiView. The LSL compiler produces output suitable for automatic scanner and parser generators (Aflex and Ayacc) and a number of Ada code units that are later linked into the MultiView kernel.

The first stage of the project was to produce a full LSL description of VHDL-93. The next stage was to ensure that the parser generated from this description correctly handled VHDL-93 syntax.

3.1 LSL

The LSL description of VHDL is a combination of the VHDL-93 grammar in pure Backus-Naur Form (BNF), plus an associated abstract syntax construction clause for each production. Pure BNF does not provide “optional” or “list” operators, which means the

```

sort statement; end;
oper assign_stmt: name expr -> statement;

<assign_stmt> ::=
  <name> '=' <expr> ';
  { assign_stmt (<name>, <expr> ) };

```

Figure 5. LSL description of an assignment statement

number of productions in BNF descriptions is greatly increased over the equivalent Extended Backus-Naur Form (EBNF).

The LSL format combines a description of the associated abstract syntax form with each BNF production. An example of the LSL description for a VHDL assignment statement is given in Figure 5.

In this example, an AST sort named *statement* and an operator named *assign_stmt* have been declared. The operator *assign_stmt* has two child nodes, of sort *name* and *expr*, and is itself of sort *statement*. This example AST description instructs the MultiView parser, on a successful parse of the associated `<assign_stmt>` production, to build a subtree with root node *assign_stmt* and to make the first child the subtree generated by the parse of `<name>` and the second child the subtree generated by the parse of `<expr>`. As an example, the string “`id := id + 1`” when parsed with this scheme would produce the AST shown in Figure 6.

MultiView uses a fixed arity AST model, which means that variable-length lists must be implemented with a right-recursive tree structure. This means that lists consist of a node with two children, the left of which is a list element, while the right child is either a list node itself or a null operator signalling the end of the list (see Figure 9 for an example).

3.2 LSL Implementation

The BNF used for the VHDL LSL implementation in this project was taken directly from the *VHDL Language Reference Manual* [18]. This was done in order to help the user of the editor to recognise the names of the productions rather than to adhere to an unambiguous definition of VHDL. However the BNF used was not designed to be used within the context of an automatic parser generator and contains many non-LR(0) productions. To some extent, such ambiguities are unavoidable since VHDL inherits much of Ada’s notoriously difficult-to-parse syntax.

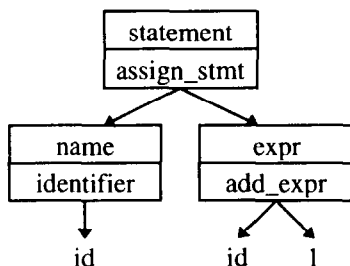


Figure 6. AST for an assignment statement

The automatic parser generator (Ayacc) that is used with the MultiView language compiler generates a shift-reduce LALR parser. This is effectively an LR(0) parser with some ability to disambiguate conflicts via lookahead. However, there are limitations to the LALR parser approach and, because of this, the initial LSL specification for VHDL had to undergo a long “debugging” stage for it to correctly parse VHDL-93 models. There is no standard validation suite for VHDL-93, so this debugging process involved passing various correct VHDL units through a test parser to test for syntax errors. When an error occurred, the behaviour of the parser was traced to discover at what point it took the wrong turn. When this was found some way of removing the problem had to be discovered, as discussed later in this section.

A typical example of the sort of problem that arose is the statement:

```
f (a, b, c) := 0;
```

The parser knows from the context in which this appears that this must be a statement, but it is unable to know what kind until it sees the ‘:=’ sign. The problem occurs when the parser reads in the ‘a’ and must make a choice to reduce it to a subprogram parameter or an array index. This is a problem because, at this point, the construct could be one of three things:

- an array assignment
- an assignment to the array result of a function call
- a procedure call

If the parser chooses to reduce the ‘a’ to a procedure call parameter, this will cause a syntax error to occur at the ‘:=’ operator later on in the parse. In a VHDL compiler, the solution would be to extend the lexer to look at a symbol table to find out what ‘f’ refers to and represent this as a separate token type [1]. Unfortunately, within the tools provided by MultiView the only solution was to merge the BNF productions for subprograms and array references into a single production that accepts both. This results in a parser that could accept some VHDL syntax that is technically

```
<discrete_range> ::=
  <range> | <subtype_indication>
```

```
<range> ::=
  <simple_expression>
  ('to' | 'downto')
  <simple_expression>
```

Figure 7. BNF for ambiguous array definition

<i>Format</i>	<i>Description</i>
<code>\$ string \$</code>	Print <i>string</i> using the keyword style.
<code>' string '</code>	Print <i>string</i> using normal text style.
<code>% <production> %</code>	Show <production> as a required placeholder.
<code>! <production> !</code>	Show <production> as an optional placeholder.
<code>/</code>	Begin a new line.
<code>></code>	Increase the indent level one unit.
<code><</code>	Decrease the indent level one unit.
<code>#n</code>	Unparse child <i>n</i> .
<code>?n{scheme1}{scheme2}</code>	If child <i>n</i> exists unparse with <i>scheme1</i> , otherwise use <i>scheme2</i> .
<code>@</code>	Display the literal value of the node.
<code>\x</code>	Display <i>x</i> as a literal without interpreting it as an unparsing symbol.

Figure 8. TextView unparsing operators

incorrect.

Another method of solving language ambiguities is to build a new tree of productions for use in the production where the ambiguity arises. This involves duplicating one or more productions appearing in the ambiguous production so that the ambiguous production has its own version that is not used in any other context. This effectively gives the parser more context information and allows it to make the correct decision. This approach was used to solve a problem in the `<discrete_range>` production used in array type declarations (illustrated in Figure 7).

The problem is that discrete range declarations consisting of just a `<subtype_indication>` (an identifier) were reduced to a `<simple_expression>` which then lead to a syntax error when no “to” or “downto” was found after the identifier. The solution was to create a new `<array_discrete_range>` production that was identical to the original `<discrete_range>`, but using new `<array_range>` and `<array_subtype_indication>` productions, which were identical to the originals without the *array* prefix. This local copy of the production tree allowed the parser to make the correct decision when it found an identifier in an `<array_range>` based on the more restricted set of choices now offered.

3.3 TextView unparsing

As mentioned in section 2.1, views interacting with the MultiView database server only have access to units in AST form. TextView requires a textual format both for its on-screen display and to provide hybrid editing features. The process of turning an AST into a pretty-printed textual form is termed *unparsing*, and is achieved by associating formatting templates with each operator in the LSL specification for the language. A particular set of such templates is termed an *unparsing scheme*. TextView allows more

than one scheme to be defined and the current scheme may be changed by the user as necessary.

The operations available within unparsing schemes control both the text that is produced and how the text is laid out and printed. For instance, keywords may be in boldface, placeholders in italics and normal text in the base font. The operations available in an unparsing scheme are shown in Figure 8.

Different styles can be defined for each of the \$, ', ! and % operators. Layout is controlled by the <, > and / operators which decrease/increase the indentation level and begin a new line respectively. The @ operator is generally only used for the leaves of the AST to display actual identifiers, numbers, string literals, etc. The ? operator allows the unparsing to select different outputs based on whether a child exists or not. This is often used for displaying optional children of an operator or to provide list separators.

As an example of an unparsing scheme, the scheme used to display a VHDL assignment statement with no placeholders will be given. The AST for this statement is shown in Figure 9. The unparsing begins at the

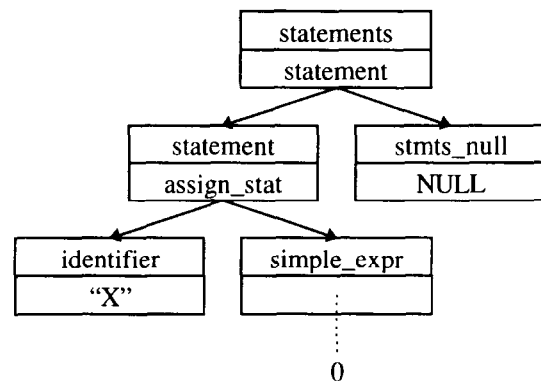


Figure 9. AST for a list of statements

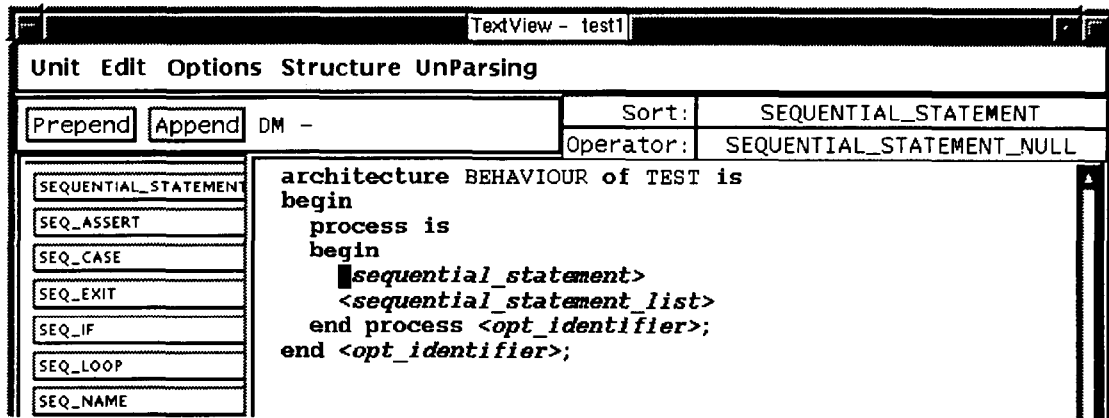


Figure 10. Expansions for *sequential_statement*.

root of the tree and looks for an unparsing scheme for *statements*. This scheme might look like:

```
statements -> ?1{ #1 } #2
```

Remember that this is a list of statements defined recursively: child 1 is a *statement* while child 2 is of sort *statements*. Therefore this scheme results in each non-null statement being prefixed by a new line. If no statements exist, then nothing will change. In the example, the unparser sees that child 1 exists, and uses the first part of the unparsing scheme in braces. This means that a new line is begun and child 1, an *assign_stat*, is unparsed.

```
assign_stat -> #1 ' := ' #2 ';
```

Child 1 is an identifier for which the literal value “P” is printed. Then a “:=” is displayed, followed by child 2, which is a *simple_expr* evaluating to “0”. After unparsing *assign_stat*’s children, the unparser goes up one level and finds the end of *statements*, marked by *stmts_null*. At this point, the unparser has finished parsing the root and terminates. The final output of this process is:

```
newline
X := 0;
```

3.4 TextView editing

As mentioned in the discussion of structure editors in section 1.2, one of the main text-entry features of a structure editor is intelligent template insertion. Template operations within TextView are controlled by the current unparsing scheme. The required “canonical” scheme simply shows all non-null nodes

in the AST in textual form. To allow the user to enter structures using a process of stepwise refinement, one or more unparsing schemes showing placeholders must be defined. Therefore it is not enough for the builder of a TextView for a new language to provide only a canonical view for units, since this would render one of the most powerful features of the editor effectively useless. Accordingly, as part of this project, a “placeholders” view that allows this form of “point-and-click” editing has been produced.

An example of a view with placeholders displayed is shown in Figure 10. The placeholders are printed as their production names in italics. Here all placeholders are optional—they do not need to be expanded in order for the unit to be parsed and they would all disappear if the canonical scheme were selected.

In a view that displays placeholders, a placeholder, or *handle*, appears wherever an optional or required subtree can be placed and does not currently exist. The user may select the placeholder and view a list of valid transformations at the left hand side of the screen. Clicking on a transformation will expand the selected placeholder to that construct. In order to provide useful and efficient editing for TextView, the designer must consider how the user would like to use the placeholder expansions and build the placeholder conventions accordingly. The users that wishes to personalise these schemes may do so by creating their own variant views.

As an example of how the user might edit a VHDL *process* statement using placeholder expansion, suppose the user wishes to add an *if* statement to the *process* construct shown in Figure 10. The user has already expanded the placeholders to get to *<sequential_statement>* and now clicks on the *seq_if*

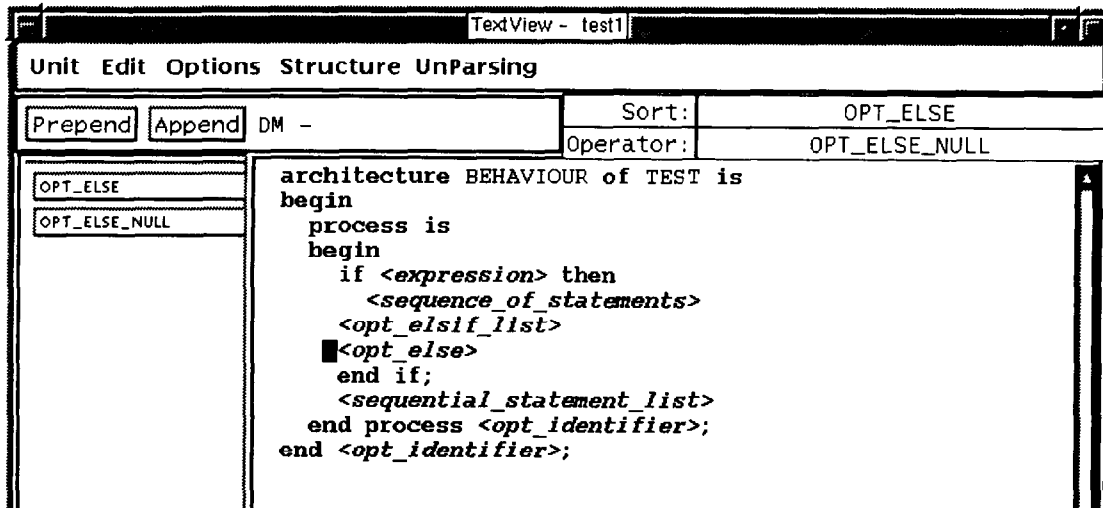


Figure 11. Expansion of *if_statement*

button to expand the production to the *if* statement template shown in Figure 11.

Now the user can fill in the *<expression>* and *<sequence_of_statements>* placeholders. The user may also add, for instance, an *else* statement by moving to the *<opt_else>* placeholder and then clicking on the *opt_else* button.

3.4.1 Creating placeholder schemes

Missing operators in the AST are represented by their *completing operator* which usually has the same name with “_null” appended. In order to have a placeholder displayed for the missing construct, an unparsing scheme for its completing operator needs to be defined. For example, in the case of an *if* statement (see section 3.3) the unparsing scheme for its completing operator might be as shown in Figure 12.

The *<expression>* placeholder is indicated as required by enclosing it in %’s, while the other placeholders are all marked as optional by !’s.

The view author needs to consider whether it might be easier to “skip” some connecting productions. For

```
if_statement_null ->
  $if $ %<expression>% $ then$ >/
  !<sequence_of_statements>!
  /!<opt_elsif_list>!
  /!<opt_else>!
  </ $end if$ ';
```

Figure 12. Unparsing scheme for an *if* statement completing operator

instance, the optional *<sequence_of_statements>* placeholder in the example above might be changed to the next production down in the tree, a *<labelled_sequential_statement>*, since it is unlikely the user wants to insert an *if* statement with no internal statements. By this sort of careful design, schemes can be optimised for VHDL.

4. Conclusion

The project has achieved its goal of providing a structure editor for VHDL designers. The editor can help designers enter and modify VHDL models, and provides assistance with handling VHDL’s complex syntax. The editor also catches syntax errors as they are entered, reducing development time. Two unparsing views are available, the standard view which shows no placeholders, and a view that displays all possible placeholders. This latter view allows the user to enter code via a “point and click” expansion technique.

The decision to build a VHDL TextView under the MultiView environment has been vindicated, since the time saved in development of the basic structure editor technology was used to implement an editor that can accept the full VHDL-93 syntax and provides two useful, customisable views of the code. Given the time constraints, it is likely that, had the editor been built from scratch, it would have been necessary to settle for a subset of VHDL.

4.1 Further work

4.1.1 MiniView

Although the unparsing operators provided by `TextView` can be combined to produce a wide range of layouts, there are a few limitations that become apparent for a language like VHDL. One of these limitations is that there is no provision for handling long lists of items in an intelligent manner. In VHDL, long lists of parameters occur frequently (see Figure 13) and it only makes these lengthy lists harder to read if they extend off the screen or wrap at a strange point. Since `TextView` does not allow the user to manually force new lines at appropriate points, some provision should be made for handling this automatically.

There are two problems to be solved in order to implement automatic line-wrapping in `TextView`. One is that, at this time, `TextView` is still undergoing extensive development, and has not been written to be easily read by others. The second problem is that the `TextView` unparsing description has no support for this sort of extension. This means that, at least initially, any parameters for automatic line-wrapping would have to be “hard-wired” or stored in a configuration file separate to the unparsing scheme.

It seems that the best solution to the first problem would be to incorporate the `TextView` unparsing unit into an experimental “MiniView” that simply displays the results of experimental unparsing schemes in an X Windows text pane. This would be feasible because the unparsing unit is connected into `TextView` via a narrow and simple interface: the unparsing unit takes an AST and an unparsing scheme as inputs and outputs a series of tokens which can then be used to generate the display. Once the unparsing scheme is made to operate within MiniView, work can be done to extend the unparser and the results can be viewed within MiniView. The new unparsing engine could be incorporated back into `TextView` at a later date.

The list-handling abilities incorporated into MiniView would probably need to be customisable for each particular list type. In general however, when automatic wrapping occurs, the user would probably like the next line to automatically indent so it aligns with

```
entity coeff_ram is
  port (rd, wr : in bit; addr : in
        coeff_ram_address; d_in: in real; d_out :
        out real);
end entity coeff_ram;
```

Figure 13. Example of a long list in VHDL

the first element of the list. For instance, with this rule in place, the port map list in Figure 13 might be displayed as in Figure 14.

4.1.2 Unparsing schemes

As mentioned in section 3.4.1, two unparsing schemes currently exist for VHDL `TextView`, a canonical scheme and a scheme that displays all possible placeholders allowing the user to perform template insertion and replacement. The placeholders view achieves its purpose, but at the cost of introducing a large amount of clutter on the screen, most of which the user will not be interested in. This is because VHDL has a large number of optional constructs, many of which are not normally used (such as statement labels). Therefore, in order to be less confusing to the user, it would be helpful to create new placeholder schemes displaying only required placeholders and “useful” optional placeholders. What defines a useful placeholder in a given context would need to be discussed with designers as they use the editor.

Another use for placeholder schemes is to provide *elided* views of designs. Elision is a way of removing low-level detail from code to provide an overall view of its structure, similar to the way an outline program is used to collapse and expand headings at different levels. An elision facility would be particularly useful in VHDL `TextView`, since models in VHDL can become very large, and high-level views can help the user “see the forest for the trees”.

4.1.3 Schematic view

The VHDL structure editor built for this project was intended from the beginning to be one of a number of possible tools for VHDL designers. In particular, it is envisioned that a complementary schematic view for VHDL would be extremely useful.

The schematic view would present a graphical representation of the connections between entities within a model. It might also show processes within entities and their interconnections via signals if this level of detail were required. In Figure 15 a simple schematic

```
entity coeff_ram is
  port ( rd, wr : in bit;
        addr : in coeff_ram_address;
        d_in: in real;
        d_out : out real);
end entity coeff_ram;
```

Figure 14. A long list with automatic wrapping

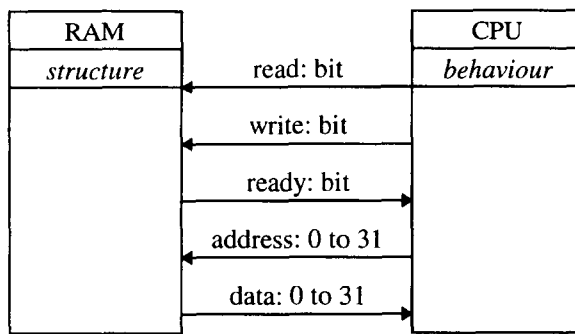


Figure 15. Simple schematic diagram

of a CPU connected to a random access memory (RAM) unit is shown.

The schematic view could be connected to a VHDL simulator so that it can display the results of a simulation in schematic form. Editing operations might include changing the way entities connect, adding and deleting entities and changing the architecture of selected entities.

The design of a schematic view poses a number of problems. Some of these are:

- How to automatically lay out the connecting signals between entities. Should the user be able to override this layout? Some work on layout schemes has already been done in the context of FlowView, a MultiView flowchart view [5].
- How to present information regarding the internal structure of each entity to the user. Some sort of elision and/or zoom mechanism would be required.
- How to handle multiple interconnected units that form a model. Within the MultiView system each unit is self contained, so if a model in one unit references an external entity, a way needs to be found for the schematic view to easily find the correct unit.

If the schematic view were to be interfaced to a VHDL simulation system, a number of technical problems arise, including the handling of asynchronous messages from both the simulator and the MultiView kernel concurrently.

References

[1] A. Aho, J. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts, 1977.

- [2] Lloyd Allison, "Syntax Directed Program Editing", *Software – Practice and Experience*, May 1983, **13** (5) pp. 454–465.
- [3] J. Brook, "TextView: a textual editing view for MultiView interfaced to the X Window System", Honours Report, Department of Computer Science, University of Adelaide, Adelaide, South Australia, Nov. 1990.
- [4] C.N. Fischer, G.F. Johnson, J. Mauney, A. Pal, D.L. Stock, "The Poe Language-Based Editor Project", May 1984, *SIGPLAN Notices*, **19** (5), pp. 21–29.
- [5] D. Jacobs, "FlowView: A Flowchart Editing System for the MultiView Programming Environment", Honours Report, Department of Computer Science, University of Adelaide, Adelaide, South Australia, Nov. 1991.
- [6] G.E. Kaiser, P.H. Feiler, F. Jalili, J.H. Schlichter, "A Retrospective on DOSE: An Interpretive Approach to Structure Editor Generation", August 1988, *Software – Practice and Experience*, **18** (8), pp. 733–748.
- [7] C. Lee, "A Textual View for the MultiView Programming Environment", Honours Report, Department of Computer Science, University of Adelaide, Adelaide, South Australia, Oct. 1987.
- [8] C. D. Marlin, "A Distributed Implementation of a Multiple View Software Development Environment", *Proceedings of the 5th Conference on Knowledge-Based Software Assistance*, Syracuse, NY, Sep. 1990, pp. 388–402.
- [9] M. McCarthy, C. Marlin, "Interprocess Communication Protocol Support in a Distributed Integrated Software Development Environment", *Proceedings Seventeenth Annual Computer Science Conference*, Christchurch, New Zealand, Jan 1994.
- [10] Douglas L. Perry, *VHDL: second edition*, McGraw-Hill, New York, NY, 1994.
- [11] M. Read, "TextView: a Textual View for the MultiView Environment", Honours Report, Department of Computer Science, Flinders University, Adelaide, South Australia, Nov. 1993.
- [12] S.P. Reiss, "Graphical Program Development With PECAN Program Development Systems", May 1984, *SIGPLAN Notices*, **19** (5), pp. 30–41.
- [13] T. Reps, T. Teitelbaum, "The Synthesizer Generator", May 1984, *SIGPLAN Notices*, **19** (5), pp. 42–48.

- [14] R.M. Stallman, "EMACS The Extensible, Customisable, Self Documenting Display Editor", *SIGPLAN Notices*, June 1981, **16** (6) pp.147–156.
- [15] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A syntax-directed programming environment", *CACM*, 1981, **24** (9) pp. 563–573.
- [16] T. Teitelbaum, T. Reps, S. Horowitz, "The Why and Wherefore of the Cornell Program Synthesizer", June 1981, *SIGPLAN Notices*, **16** (6), pp. 8–16.
- [17] *IEEE Standard VHDL Language Reference Manual*, IEEE/ANSI Standard 1076-1987.
- [18] *IEEE Standard VHDL Language Reference Manual*, IEEE/ANSI Standard 1076-1993.
- [19] T.R. Wilcox, A.M. Davis and M.H. Tindall, "The design and implementation of a table driven interactive diagnostic programming system", 1976, *CACM*, **19** (11) pp. 609–616.
- [20] S.R. Wood, "Z – the 95% Program Editor", June 1981, *SIGPLAN Notices*, **16** (6), pp. 1–7.