STEVE: A Syntax-Directed Editor for VHDL Based on SAVANT

Katrina E. Kerry
katrina@cs.adelaide.edu.au

Peter J. Ashenden
petera@cs.adelaide.edu.au

Michael J. Oudshoorn
michael@cs.adelaide.edu.au

Department of Computer Science
University of Adelaide
Adelaide, SA 5005
Australia

Abstract

This paper describes STEVE, a syntax-directed editor for VHDL, which uses auto-completion and automatic template insertion to accelerate the entry of syntactically correct VHDL models. It is part of a larger project, VIDE, which integrates textual and graphical tools for HDL design entry. STEVE extends the SAVANT intermediate representation to deal with representation of templates and the on-screen presentation of a VHDL model. STEVE demonstrates the advantages of SAVANT's extensible object-oriented structure, and shows how it facilitates the integration of a new tool into an existing CAD tool framework.

1. Introduction

STEVE is a syntax-directed editor for VHDL. It uses the SAVANT extensible object-oriented intermediate representation developed jointly by the University of Cincinnati and MTL Systems [6]. SAVANT has been designed as an extensible means for representing Hardware Description Languages (HDLs) and as a standard intermediate representation in hardware design tools. The use of a standard intermediate representation allows tools to be easily combined and integrated, producing more powerful and flexible design environments.

The aims of the SAVANT project are to build a suite of software tools to analyse VHDL programs, construct an intermediate representation and to output C++ suitable for execution with the TyVIS VHDL simulation kernel [10], also being developed at the University of Cincinnati and MTL Systems. The SAVANT project also aims to encourage research within the VHDL community.

The VIDE suite of tools [5] explores possible extensions of the SAVANT intermediate representation required to support the creation and manipulation of VHDL programs. STEVE [5], a syntax-directed editor within the VIDE framework, extends the SAVANT intermediate representation to support entry, editing and display of text. STEVE also explores the possibility of embedding semantic information into the intermediate representation, enabling programs to verify both their syntactic and their semantic correctness upon program entry.

2. VIDE

Hardware design of digital systems is becoming increasingly complex and so tools and design methods have been developed to overcome and reduce this complexity.

CAD tools traditionally used by designers of hardware components are graphically oriented. This allows designers to view their design structurally, which is more intuitive than a textual representation. This graphical notation represents the structural aspects of a design well at the expense of the behavioural aspects. HDLs, such as VHDL [2] and Verilog [9], attempt to rectify this inadequacy by providing a means of defining both the structure and behaviour of hardware designs through a textual language. Textual languages, however, abandon the intuitive graphical design of the structural aspects.

VIDE (VHDL Integrated Design Environment) is a hardware design environment for VHDL under development at the University of Adelaide. VIDE attempts to overcome the inadequacies of many hardware design tools by providing both graphical and textual design tools, which operate over a common intermediate representation. This combination of tools permits the user to select which method of de-

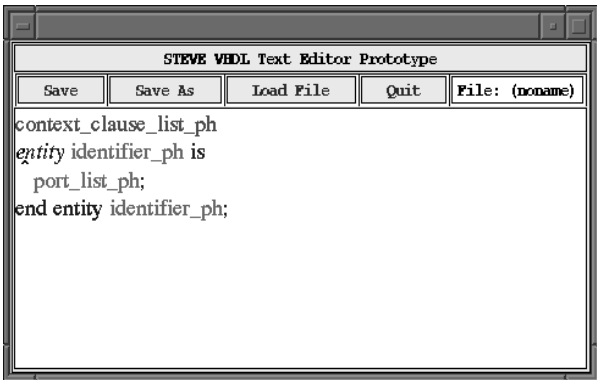

Figure 1. User interface of Steve.

sign entry they wish to use while operating on a single integrated design. The use of a common intermediate representation ensures that the user's program may be viewed consistently and concurrently through both tools.

STEVE is an integrated textual design tool for the development of VHDL descriptions. Textual design tools bring many benefits to the user. Such tools range from simple text editors to syntax-directed editors such as SED [1] and tools that have facilities for partial semantic analysis and incremental compilation [8]. STEVE is a syntax-directed editor ensuring syntactic correctness of VHDL programs as they are entered. An example of STEVE's user interface is shown in Figure 1.

VGE [4], or the VIDE Graphical Editor, is the graphical editing tool for VHDL in the VIDE framework. It has been developed with the same design goals as STEVE.

3. Automatic Template Insertion

Syntax-directed editors help decrease the number of programming errors and reduce development time by preventing errors or by informing the user of mistakes as the program is entered. A syntax-directed editor performs this error detection by parsing the text and user's commands as it is entered, and by ensuring that an incomplete program conforms with the syntax rules of the language.

STEVE employs automatic template insertion as its method of program entry. This technique does not require the user to learn any complex editor commands (as is necessary in some syntax-directed editors), nor does it require the user to be fully conversant with the language syntax thereby assisting code development by novices. STEVE has knowledge of VHDL syntax and uses place-holders to identify insertion points. When a user selects a place-holder and commences typing, the input shown is parsed and once a valid, unique language construct is determined its template is inserted at the entry point. This process is termed automatic template insertion. If the user enters an identifier (eg. left hand side of an assignment statement) then auto-completion is used to insert the complete identifier once it can uniquely determined from the set of valid identifiers in scope.

Such techniques exploit knowledge of the syntax and the structure of the target language (VHDL) and increase the efficiency and ease of design of programs. The insertion of a template guarantees syntactic correctness and admits limited semantic checking (eg. declaration of identifiers and type consistency).

Each template in the language is derived from the syntax rule of a language construct. It contains all key-words, punctuation, non-terminal symbols (place-holders) and layout information including indentation levels and carriage returns. STEVE displays the distinction between tokens and place-holders by displaying their textual representations in different colours. This distinction identifies expansion positions clearly to the user. STEVE also provides support for optional tokens and for template positions in which a choice has to be made between tokens. Support is also provided for expandable lists of statements.

To illustrate the way in which templates are derived from syntax rules, Figure 2 shows the simplified syntax rule for an entity declaration used by STEVE. Figure 3 shows the template constructed for the syntax rule. It contains place-holders for an identifier and a port clause.

```
entity_declaration ::=
      entity identifier is
            port_clause;
      end entity identifier;
```

Figure 2. Syntax rule for an entity declaration.

```
entity identifier is
      port_clause;
end entity identifier;
```

Figure 3. Template for an entity declaration.

Syntax-directed editors use an internal representation of the user's program. This may be either a concrete syntax tree or an abstract syntax tree (AST). A con-

crete syntax tree is a hierarchal representation of the user's program corresponding to the expansion of the language's grammar rules. The concrete syntax tree is accordingly made up of parent nodes, which correspond to non-terminal symbols of the language, and leaf nodes, which correspond to terminal symbols. This representation is expensive as it requires the storage of redundant information (key-word and punctuation symbols) in the tree. An abstract syntax tree deletes this redundant information and stores only the non-terminal nodes, and identifier and literal information.

Syntax-directed editors require special techniques for text entry, editing and display. These operations are more complex than those required for a basic text editor due to the internal representation of a program and constraints placed on their manipulation.

Automatic template insertion works by inserting a valid template into the position previously occupied by a place-holder, ie., replacing a non-terminal symbol in the syntax rule. These insertions correspond to manipulations of the program's intermediate representation, the abstract syntax tree. Expanding a place-holder or modifying a key-word corresponds to expansions in the abstract syntax tree. Furthermore, traversal of the abstract syntax tree corresponds to traversal of the program.

The concept of auto-completion used in automatic template insertion can be extended beyond completion of templates and key-words to auto-completion of identifiers. Identifier auto-completion can be performed by matching the characters the user enters against declared identifiers that are visible. This matching requires a symbol tree to be maintained, holding information on all declared identifiers. Identifier nodes in the AST contain references into the symbol tree and are able to update and obtain information from their references.

Auto-completion applied to identifiers is useful in indicating undeclared identifiers so that the user is informed of any spelling mistakes or errors made.

3.1 Semantic Analysis

STEVE supports rudimentary semantic analysis of programs by embedding semantic information into the intermediate representation. Semantic aspects considered by STEVE include identifier declarations, semantic analysis for an entity declaration and cases where there are restrictions on token choice. STEVE performs symbol tree management to support this semantic analysis.

STEVE maintains references to all declarations within a symbol tree. This symbol tree contains references to all identifiers declared within the VHDL program and also all identifiers that have been included in the program using 'library' statements and 'use clauses'.

A library statement contains a list of library names that are to be included within the program. Visible library names must be pre-loaded into the symbol tree when STEVE is initialised. By including a library into a VHDL program, subsequent 'use clauses' can include packages and design units contained within the library. Hence, including a library requires that its contents must be loaded into the symbol tree. Use clauses are able to include specific contents of a design unit contained within a library.

The symbol tree maintains a list of pointers to identifiers that reference a declaration, so if a user changes a declaration, references to identifiers can be easily highlighted as invalid references.

4. The SAVANT IR

The AST structure internal to SAVANT is known as its Internal Intermediate Representation (IIR). SAVANT also provides methods to produce a File Intermediate Representation (FIR) of a program. The development of a common intermediate form, utilised by different VHDL development tools, ensures portability between different tools.

The SAVANT intermediate representation is implemented through a C++ [7] class hierarchy, which is derived from the hierarchy of the VHDL grammar. Figure 4 illustrates the class hierarchy used by SAVANT. The base class in SAVANT is the class Base. It contains basic information that is common to all nodes in the intermediate representation. Every subsequent class inherits from the Final_Base class. Each node in the tree is identified by an enumerated value, which states the type of the node, and a string, which duplicates this information in a printable format.

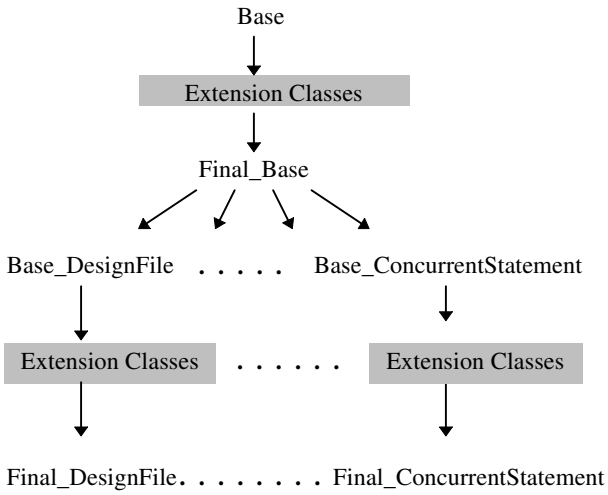

Figure 4. SAVANT IR design.

Each node in the intermediate representation has a base class and a final class which serves to encapsulate additional classes required by a CAD tool based on SAVANT. The base class for each node, for example class Base_Entity_declaration, is similar in purpose to mixin classes as described by Booch [3]. A mixin class is a non-instantiable class which defines attributes and behaviour which may be inherited by a class to extend another class. The base classes in the SAVANT hierarchy have a similar purpose but are not implemented through multiple inheritance mechanisms. Base classes form part of the SAVANT hierarchy, defining relevant attributes for each node.

Base classes provide the mechanism for granting extension classes access to node attributes, whilst a standard method for instantiation of a node is obtained through the use of the final class. This technique defines clearly the tool-specific class insertion points. Each node is modelled by an object instantiation of the node's corresponding final class.

This mechanism is illustrated in Figure 5. The base class for a design unit node contains a pointer to the instantiable final class for an entity declaration. This structure allows any extension classes for a design unit node to have access to the entity node attribute whilst other classes that contain design unit nodes as attributes can do so by instantiating its corresponding final class. Instantiation of this final class provides access, through inheritance, to any attributes added to the node through extension classes.

The class hierarchy follows VHDL's grammar closely. For example, an entity declaration inherits from a library unit, which in turn inherits from a declaration (see Figure 5). As can also be seen in Figure 5, a design unit node contains pointers to object instances of its children nodes, in this case, an entity declaration node. As an identifier value is common to all declaration nodes, this class member has been moved upwards within the hierarchy to reside in the declaration class, taking full advantage of the object-oriented nature of the intermediate representation.

4.1 Extending the SAVANT IR

The SAVANT hierarchy design is the basis of STEVE's design. The main concept of the SAVANT design is that nodes are modelled by objects; any extensions made by STEVE to SAVANT must maintain this concept and implant any extra information required, such as template information and the knowledge to manipulate these templates, into each node. Figure 6 shows the design of STEVE integrated with SAVANT.

Extension classes added by STEVE are prefixed by "MIR_". Symbols are represented by a MIR_symbol class, and are instantiated as either a MIR_token, a MIR_placeholder or as any of the child classes of MIR_symbol. Templates are represented as a list of

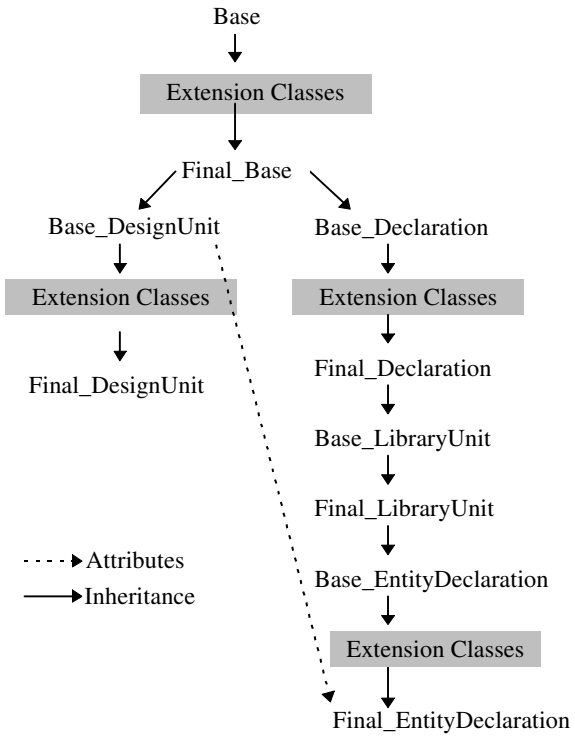

Figure 5. SAVANT IR for an entity declaration.

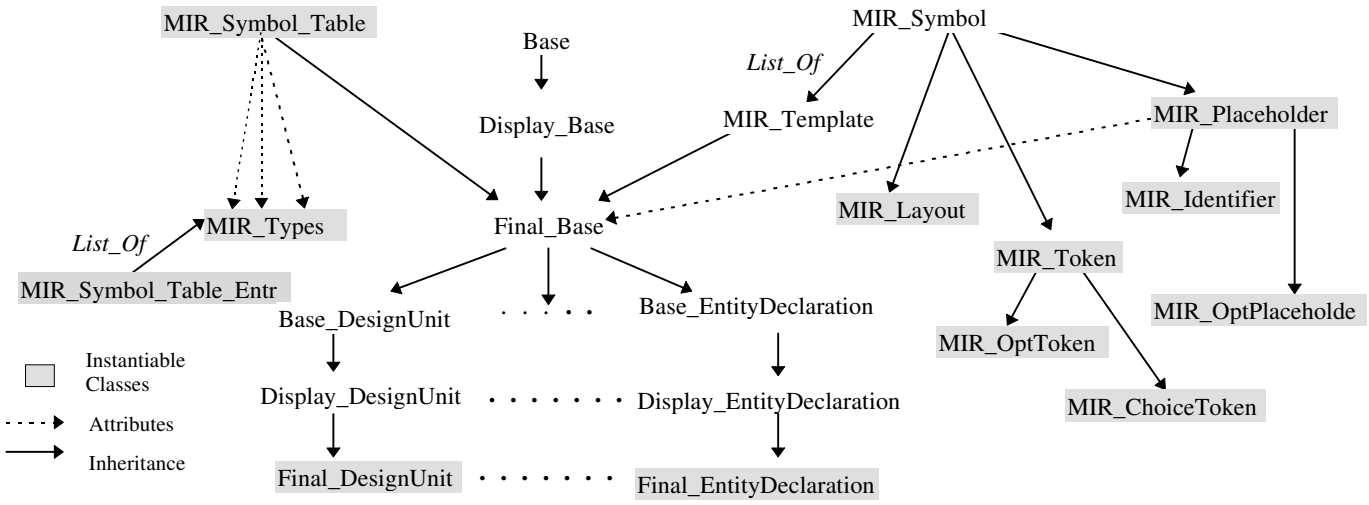

Figure 6. SAVANT hierarchy with STEVE extensions.

MIR_symbol objects.

Each node has access to its own template through the inheritance of the template class at Final_Base. The template class provides methods for creating, accessing and manipulating templates. The classes inserted at the predefined extension positions for each node in the intermediate representation, for example Display_Entity_declaration, use these methods to create their own templates and perform any template operations that the node requires. The extension classes also contain methods to respond to requests by the user, passed on by the user interface.

Many of the methods required in the extension classes are very similar in behaviour and content. Nodes can be classed into groups according to their template contents. For example, templates that contain lists form one group; similarly, templates that contain identifiers may form another group. STEVE uses template frameworks to enable easy class development for a node. A template framework is a class template which contains many of the methods required by a specific type of node. A framework will typically require minimal modification to make it suitable for a particular node. An example template framework for a basic node is shown in Figure 7. This framework includes methods for display, a template method for creation, methods for each child, and methods for editing and automatic template insertion.

```
/* ------------------------------------------------------------------
*/
/* Framework : base */
/** Specification for the base framework. Framework includes
 ** general methods required by any node that has a template.
 ** Many templates require only this framework. These templates
 ** contain only tokens, placeholder, terminal and layout symbols.
 ** Child constructs do not start with an identifier.
 **/

/* constructor function */
Display_<class_name>(Final_base*, long);

/* destructor function */
~Display_<class_name>();

/* create children functions*/
void create_<child_name>(Final_base*, long);

/* return automatic template insertion information
       using template method */
ati_return_data* get_ati(MIR_symbol*);

/* remove and cut function for children */
void remove(Final_base*);
/* ------------------------------------------------------------------ */
```

Figure 7. Base framework.

The use of frameworks supports the concept of extensibility, hence, it is easy to modify existing nodes and to add nodes to the intermediate representation. Any special requirements for a node can be implemented as modifications to the framework that it uses. Frameworks can be organised into a hierarchy showing the additional methods required for each new level of complexity in the hierarchy. An example hierarchy used by STEVE is illustrated in Figure 8.

In many cases a node belongs to many framework groups and hence requires methods from multiple

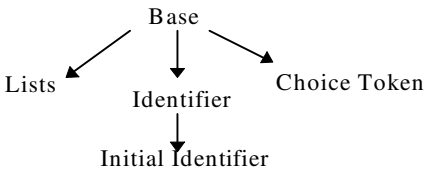

Figure 8. Example framework hierarchy.

frameworks. For example, a node whose template consists of a list of interface declarations, such as a port_interface_list, may also start each of its interface declarations with an identifier. A port_interface_list extension class requires methods from the basic framework, the list framework and the initial identifier framework.

The MIR_symbol class contains information and methods common to all symbols. This information includes a position number in the graphical display, a text value and a name. The text value is used for displaying the symbol in the user interface. Methods provided by the MIR_symbol class include methods to create a symbol, to modify a symbol's information and to gain access to a symbol's information. A MIR_Placeholder object has as one of its attributes a pointer to an object of type Final_Base. This attribute points to a child node in the intermediate representation, which in turn has its own template.

Figure 9 shows an implementation of the STEVE design for a subset of VHDL including an entity declaration and a design unit. Figure 9 illustrates the use of the extension class insertion positions in SAVANT for the Display extension classes.

Figures 6 and 9 also show symbol table management classes that are used by STEVE. These classes provide simple functions to add, search and remove entries from the symbol table. The classes are included in the design to allow some semantic analysis to be performed by STEVE.

Each node in the intermediate representation inherits from the symbol table class through the inheritance of the Final_Base class. This inheritance gives each node access to methods to check that identifiers have been declared, and to insert and remove objects from the symbol table. The symbol table class also provides a method which takes a string parameter, used to search for partial matches. This method provides the mechanism for performing auto-completion of identifiers.

The symbol table groups the symbols into pools sorted by the type of identifier. For example, there are separate pools for types, package names, reserved words and general identifiers. This concept can be extended to sort the general identifier pool into separate pools for entity identifiers, architecture identifiers, constants and variables.

The pools maintained by the symbol table are static members of the MIR_symbol_table class, which means that each instantiated object accesses the same data structures. This restriction means that the symbol table must be rebuilt whenever a new program is loaded into STEVE. Further design and implementa-

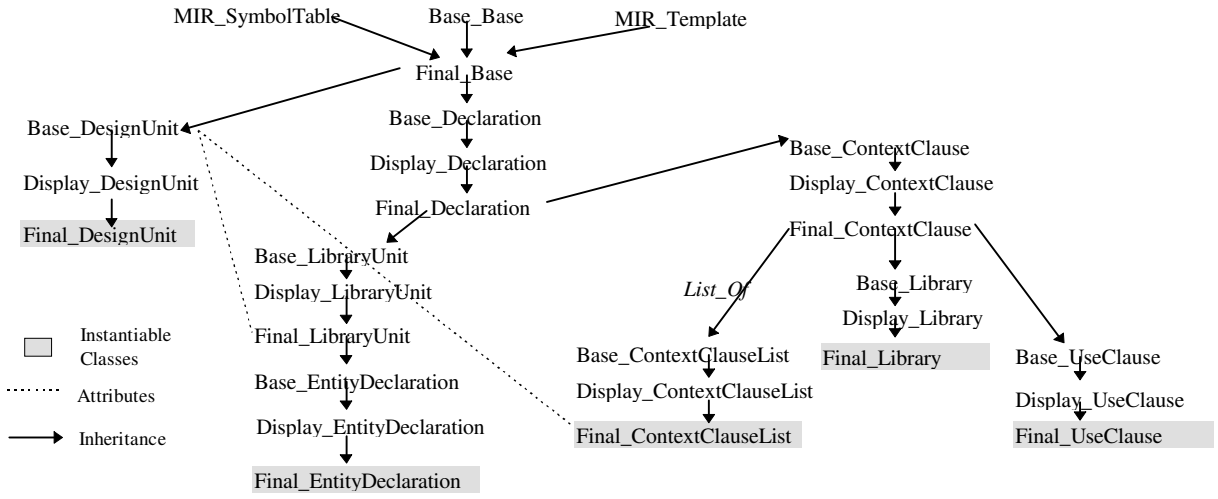

Figure 9. STEVE design for an entity declaration.

tion will address this restriction.

5. Implementation

STEVE has been implemented in prototype form for a subset of VHDL. This subset includes simplified versions of VHDL syntax rules, including a design unit, which consists of a list of context clause statements and an entity declaration.

The high-level system structure of STEVE is shown in Figure 10. It describes the communication routes between the intermediate representation and the user through the user interface. Each action performed by the user corresponds to a user interface event. Each event invokes the execution of a procedure or function in the user interface; these procedures and functions may in turn invoke methods in the intermediate representation.

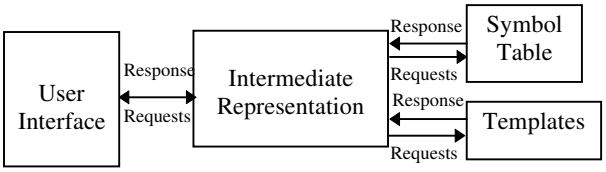

Figure 10. System level design of STEVE.

Communication between the intermediate representation and the template for a node takes place in order to access any information that the user interface requires or to perform any operations that the user interface requests. Communication takes place between the intermediate representation and the symbol table to perform semantic analysis.

5.1 Integration and Interaction with Other Tools

SAVANT has been designed to aid integration of CAD tools that extend the SAVANT intermediate representation.

Integration of multiple tools requires only that the extension classes be inserted with the appropriate inheritance hierarchy and that no naming conflicts occur within the extension classes. As long as the extension classes use the attributes of the Base and Final classes in their designated manner, then no problems arise.

One of the aims of the SAVANT project is to provide a common intermediate representation that can be used by multiple CAD tools. This requires that CAD tools output VHDL programs in a way that other tools will be able to understand. SAVANT provides a file intermediate form that can be used to port VHDL designs between tools. Unfortunately, this form of output loses any information that is specific to a CAD tool. Hence, if a design is to be reloaded into a CAD tool, it must regenerate any additional information that it requires; potentially an expensive process.

STEVE provides mechanisms to output an annotated file format of a VHDL program. This is implemented in STEVE by providing a method in the MIR_template class, which, when invoked, traverses the intermediate representation from the invoking node and prints the contents to a file. To save an entire VHDL program, this method is simply invoked from the root node of the intermediate representation. The annotations to the file include information about which nodes have been accepted, and, for each node, which place-holders have been expanded and information about any of its children nodes.

The information is printed to a shadow file for each VHDL file that is produced. The use of a shadow file allows STEVE to reload incomplete VHDL programs and easily reconstruct all display information that it requires.

5.2 Summary

STEVE has been implemented as a prototype for a subset of VHDL. It has been designed to work with the SAVANT intermediate representation and to be easily extensible and maintainable. STEVE consists of two main components: the user interface, which uses a purpose-built editor widget, and the extended intermediate representation.

STEVE operates through communication between the user interface and nodes in the intermediate representation. Nodes in the intermediate representation have access to their templates and methods for operating on the templates and also have access to a symbol table, which records all declarations within a VHDL program. Frameworks have been developed for STEVE-specific classes that allow alterations and additional nodes to be implemented easily.

6. Conclusions

STEVE is a textual programming tool for VHDL which helps the user to construct programs more easily and extends the SAVANT intermediate representation. The tool is syntax directed, and uses automatic

template insertion and the concept of auto-completion wherever possible to enforce correct syntax and to simplify program entry.

The implemented prototype of STEVE is a fully functional syntax-directed editor for a subset of VHDL. It implements automatic template insertion, auto-completion and performs some semantic analysis, including maintaining and referencing a symbol table. The extensibility of the design means that information can be added to each node in the intermediate representation in any order; so that testing and implementation can be performed in an incremental manner.

STEVE also supports extensibility by providing frameworks for STEVE extension classes. These frameworks can be extended and combined to produce a class definition for a specific node.

Automatic template insertion is a concept used successfully in STEVE. The implementation of STEVE illustrates that this concept is intuitive, considering the nature and structure of programming languages, and helps the user. Auto-completion is used to complete identifier names and tokens. When combined with semantic knowledge, auto-completion is even more successful, as it enforces selection of appropriate attributes, for example, selection of an appropriate identifier for a type place-holder.

STEVE does not perform complete semantic analysis. However, STEVE does examine the key concepts of a syntax-directed editor for VHDL and explores how successful these techniques are; producing a successful preliminary design and prototype for a complete editor.

SAVANT proved to be a good basis for STEVE. The design is easily extensible due to its object-oriented nature, and lent itself to the implementation of full functionality for a subset of VHDL. Different implementation ideas are tested easily with no major changes required in other sections of the intermediate representation. The only difficulties encountered whilst working with SAVANT were involved with lack of detailed specification and explanatory comments for the intermediate representation's design. These problems have been addressed in the successor to the SAVANT intermediate representation, AIRE [11].

Acknowledgments

The authors would like to thank Philip Wilsey of the University of Cincinnati and Praveen Chawla of MTL Systems for making the SAVANT specification and implementation available for use in the VIDE project.

References

[1] L. Allison, "Syntax Directed Editing", *Software - Practice and Experience*, Vol **13**, pp 453-465, 1983.

[2] P. Ashenden, *The Designers Guide to VHDL*, Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, 1996.

[3] G. Booch, *Object-Oriented Analysis And Design 2nd Edition*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, USA, 1994.

[4] M. Clarke, *VGE - The VIDE Graphical Editor for VHDL*, Honours Thesis, Department of Computer Science, University of Adelaide, 1996.

[5] K. Kerry, *STEVE: A Syntax Directed Editor for VHDL*, Honours Thesis, Department of Computer Science, University of Adelaide, November 1996.

[6] SAVANT home page, http://www.mtl.com/projects/savant.

[7] B. Stroustrop, *The C++ Programming Language*, 2nd Edition, Addison-Wesley Publishing Company, 1991.

[8] T. Teitelbaum & T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM*, September Vol **24** (9), pp 563-573, 1981.

[9] D.E. Thomas & P. Moorby, *The Verilog Hardware Description Language*, Kluwer, USA, 1991.

[10] TyVIS home page: http://www.ece.uc.edu/~paw/tyvis.

[11] J. Willis, P. Wilsey & D. Martin, *AIRE IIR SPEC*, http://www.cs.adelaide.edu.au/~vide/aire.