A Method for Iterative Customization of Object-Oriented VHDL Intermediate

Praveen Chawla, Michael Shellhause, Jonathan Nocjar
MTL Systems, Inc.
3481 Dayton-Xenia Road
Dayton OH 45432
(513) 426-3111
{pchawla, mshell, jnocjar}@mtl.com

Jeffery Carter
GCD Software, Ltd.
PO Box 42165
Cincinnati, OH 45242
(513) 398-5620

Abstract

This paper describes a method to develop VHDL-based tools by iteratively and incrementally customizing an object-oriented VHDL internal intermediate representation. Such a method derives its need from the availability of commercial VHDL analyzers with advanced and open object-oriented interfaces which can potentially eliminate the need to develop or use proprietary analyzers and procedural interfaces, thereby resulting in considerable savings in time and money.

1.0 Introduction

The term internal intermediate representation (IIR) is used to describe design information that is present in a computer's random-access address space [1]. VHDL tools utilize IIR to share design information between tool components such as an Analyzer, an Elaborator, an Optimizer, and a Code Generator. Typically, an Analyzer receives a VHDL description as input and then produces an equivalent IIR as output after verifying the syntactic and static semantic correctness of the input description. The analyzer-produced IIR is utilized by back-end tools such as an Elaborator, an Optimizer and a Code Generator, to produce the desired output. In addition, IIR can also be saved on a secondary storage device in a file intermediate representation (FIR) and read back into random-access address space in IIR format when needed by a back-end tool.

The term object-oriented VHDL IIR is used to describe an IIR that uses a collection of class instances (objects) linked by pointers to represent design information contained in a VHDL description. For those familiar with the compiler terminology, the object-oriented IIR represents a very generalized abstract syntax tree (AST). In addition, the methods contained in classes used to create the AST represent an application programming interface (API) for such an object-oriented IIR. It should be noted that the object-oriented IIR is easy to extend because new functionality can be added to such an IIR by adding new methods to existing classes or adding new classes to the IIR class hierarchy.

Except the recent emergence of tools [2, 3] that utilize an open object-oriented IIR, namely Advanced Intermediate Representation with Extensibility (AIRE) [1], most of the commercial VHDL tools utilize proprietary intermediate representations which are accessed through proprietary procedural interfaces. Unfortunately, such proprietary representations and interfaces do not usually permit mixing and matching of tool components from different vendors and are usually very expensive. As a result, such analyzers and interfaces inhibit VHDL-based research and product development. On
the other hand, compliance with an open and extensible specification such as AIRE promotes interoperability between tool components from different vendors. Furthermore, it stimulates basic research in CAD-in-VHDL and encourages rapid insertion of new CAD-in-VHDL products.

Although availability of commercial VHDL analyzers with advanced and open object-oriented interfaces can potentially eliminate the need to develop proprietary analyzers, intermediate representations and procedural interfaces, and can result in considerable savings in development time and money, integration of such analyzers with back-end tools such as an elaborator, an optimizer or a code generator is not trivial. Integration of such an analyzer with back-end tools can be made easier with a well-defined method to customize the IIR and a support tool that implements the method. In this paper, we describe a method to iteratively and incrementally customize a VHDL object-oriented IIR. In addition, we describe a possible support tool for the described method.

Section 2 of this paper presents the method. Section 3 describes a possible support tool for the method. Section 4 illustrates the method with an example. Finally, Section 5 provides conclusions and a description of the future work.

2.0 Method for iterative and incremental customization of IIR

We first describe the general process by which an object-oriented VHDL IIR can be customized for back-end tools. We then describe an iterative and incremental method which can be used to ease the process of customization. Figure 1 shows a possible general form of the class hierarchy for an object-oriented VHDL IIR. In fact, the IIR class hierarchy shown in Figure 1 is quite similar in form to the AIRE hierarchy [1]. The figure shows four categories of classes, namely Base, Analyzer, User and Final classes. Base classes represent generic data and methods associated with VHDL language constructs whereas Final classes shown as double rectangles in this hierarchy (leaf nodes in the class hierarchy) are instantiated to build an AST for a given VHDL description. In addition, IIR is extended by adding user-defined classes and/or user-defined methods to existing user classes. User defined classes can be added anywhere between Base and Final classes. User and Analyzer classes shown in the figure are an example of such addition. In fact, SCRAM analyzer built as part of the SAVANT project [2] uses a similar technique. SCRAM is an AIRE compliant VHDL analyzer and its source code is available for non-commercial purposes on the WWW at the address specified in [2]. If SCRAM source structure and the form shown in Figure 1 are compared, it can be easily seen that Base and IIRBase are equivalent, Analyzer and IIRScram are equivalent, and Final and IIR are equivalent. In addition, SCRAM adds _publish_cc methods to the IIRScram classes to build a C++ code generator.
Figure 1: Class Hierarchy for an object-oriented IIR

Having described a general process for extending IIR for back-end tools, we now propose a method to ease the process of extension. Figure 2 shows our proposed approach for SCRAM (VHDLYzer is the name of commercially supported version of SCRAM). The same approach applies to any AIRE compliant analyzer as long as the source code for IIR class hierarchy is available. Note that although the figure shows analyzer source as input, we do not require such input for this method to be applicable. In addition, we assume that the source code is available in C++ in the figure although the method is not dependent on the source language of implementation.

The method works iteratively and incrementally as follows. Each iteration consists of four steps, namely compilation, analysis, visualization and extension. In the compilation step, we begin by partitioning the Analyzer source into two parts, namely the IIR class hierarchy and the remaining
Analyzer code (such as the lexical analyzer and the parser). Given the IIR class hierarchy in source form, we compile the IIR source during this step and then compile and link it with the remaining analyzer code. The result of this compilation step is an executable analyzer, shown in the figure as the VHDLyzer. In the *analysis* step, we use the VHDLyzer to analyze a VHDL description. To reduce the complexity of an iteration, we use a VHDL description that contains a small subset of the language although VHDL descriptions analyzed during all iterations collectively contain all relevant language constructs that require extension. The analysis step produces IIR which is saved in a file (shown as Node Log in the figure). In the *visualization* step, we view a visual representation of the IIR tree utilizing the file produced in the previous step. In addition, we view visual representations of the class hierarchies, including inheritance and aggregation, for selected nodes in the tree. This step enables a tool developer to easily and intuitively identify the classes in the IIR class hierarchy where new methods should be inserted or places in the class hierarchy where new classes should be inserted to extend the IIR functionality. In the *extension* step, we add new methods or classes to accomplish the desired extension. The iteration is repeated with new VHDL descriptions till all the language constructs [5] that require extension have been addressed, to completely customize the IIR class hierarchy.

![Diagram](image)

*Figure 2: A method for iterative and Incremental customization*

The method proposed above defines a process for extending an object-oriented VHDL IIR. Although it is possible for a tool developer to extend an object-oriented IIR for his/her needs without following the proposed method, following such method will ease the task of identifying places in the class hierarchy where new additions are needed., thereby reducing development time needed for extending IIR. In the next section, we describe an implementation that supports the proposed method.
3.0 Implementation of the Method

We now describe a support tool for the proposed method. MTL Systems has implemented a support tool for the proposed method as VHDLyzer Tool Kit [4]. Without getting into a detailed product description, we now describe through screen shots of the Tool Kit how it has implemented the proposed method. Figure 3 shows the main screen that appears when VHDLyzer Tool kit is invoked. The main screen provides icons to initiate any one of the four steps of an iteration i.e., compilation, analysis, visualization and extension. In addition, the steps can also be initiated through menu selection. Figure 4 shows the IIR Viewer which comes up when the visualization step is initiated. The IIR Viewer presents the AST for the most recently analyzed VHDL description in form of a tree. In addition, due to size of the tree for most VHDL descriptions, it is usually not possible to view the whole tree in one screen. The IIR Viewer allows a user to view a portion of the tree that includes a particular node of the tree on the screen by selecting it from a list of nodes which is displayed on the left side of the tree. Figure 5 shows the Tree Node Inspector which comes up when the user desires to examine a particular node in the tree. It displays various details about the node such as node data, predecessors and descendants. In addition, it provides the user with an opportunity to view the IIR class hierarchy for the selected node. Figure 6 shows the Class Hierarchy Viewer which comes up when a user decides to view the IIR class hierarchy of a node. In addition to viewing the class hierarchy, user can also spawn off their favorite editor to edit the source code for a selected class from the class hierarchy.

![VHDLyzer Tool Kit Main Screen](image)

*Figure 3: VHDLyzer Tool Kit Main Screen*
Figure 4: IIR Viewer
<table>
<thead>
<tr>
<th>Node Type</th>
<th>SignalDeclaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique ID</td>
<td>59</td>
</tr>
<tr>
<td>Parent Node Type</td>
<td>SignalNameList</td>
</tr>
<tr>
<td>Parent Node ID</td>
<td>56</td>
</tr>
</tbody>
</table>

**Figure 5: Node Inspector**
4.0 Example: A Pretty Printer for VHDL

We now provide a brief example of how the proposed method can be utilized to implement a pretty printer for VHDL. A pretty printer will be required to reformat an input VHDL description and output VHDL in a format adhering to certain style and format guidelines without any modification to the syntax or semantics of the input VHDL. For example, a pretty printer may be required to produce VHDL in TeX format adhering to certain guidelines regarding boldfacing and tabs so that it can be printed using TeX typesetting software.

A pretty printer, as described above, can be created using the method proposed in this paper given the source code for an analyzer and an object-oriented IIR class hierarchy. Simply put, if the class hierarchy has the general form as shown in Figure 1, the pretty printer is created by adding new methods, say _publish_pp, to the Analyzer classes. The new methods can be added incrementally by repeating the iterative process as described in Section 2.0. Iterations can utilize a group of VHDL descriptions which include all relevant VHDL language constructs collectively. During each iteration, methods will be added to Analyzer classes corresponding to unique language constructs analyzed during the first step of the iteration. Note that if the analyzer source code is not available, the same
extension can be accomplished by adding new user-defined classes to the IIR class hierarchy, say UserABC classes, containing methods to publish output for pretty printing software. In fact, the later method is preferable since it is independent of any analyzer implementation and therefore will not require any changes to pretty printing classes even when analyzer implementation classes change as new analyzer versions are released.

5.0 Conclusions and Future Work

We have described a method to iteratively and incrementally extend and customize an object-oriented IIR for back-end tools. The method provides for visualization support to ease the task of IIR class hierarchy traversal and extension. Following the same theme, future work will focus on providing automated support for easing the task of IIR extension. For example, such support can include automatic generation of VHDL code to serve as inputs to various iterations.

Acknowledgments

This work was partially supported by an SBIR contract (# F33615-95-C-1638) from the US Air Force, Wright Laboratory, WPAFB, Ohio.

References