This is to certify that the

thesis entitled

Front End Interface for Parallel VHDL Simulations

presented by

Mai Yang

has been accepted towards fulfillment
of the requirements for

M. S. degree in Computer Science & Eng. Dept.

[illegible]
Major professor

Date 7/30/98

FRONT END INTERFACE FOR PARALLEL VHDL SIMULATIONS

By

Mai Yang

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Computer Science Department

1998

Abstract

Front End Interface for Parallel VHDL Simulations

By

Mai Yang

The Parallel VHDL Simulation Project conducted by the Computer Science Department of Michigan State University attempts to implement an efficient paradigm for VHDL simulations on massively parallel machines. The Front End Interface of the project is in charge of the translation from VHDL descriptions into C++ code which is used to run parallel simulations. This document describes how the SAVANT software package is utilized and modified to implement the Front End Interface.

ACKNOWLEDGMENTS

I want to express my gratitude towards my advisor, Professor Moon J. Chung, for his support and direction throughout my involvement in this project. I am deeply impressed by his tenacity and perseverance towards his goals. I thank him for his encouragement and advice which helped me overcome lots of difficulties. Without his support, this thesis would not have been completed. I am also grateful to Professor Matt W. Mutka and Professor Anthony S. Wojcik for serving in my guidance committee and providing comments on my thesis.

I owe special thanks to my partner Khashayar Rohanimanesh and Jinsheng Xu. I benefited a lot from our discussions and the helpful suggestions they gave me. It was a wonderful learning experience for me to have teamed up with these two nice fellows in this research project.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1

Introduction

1.1 VHDL

VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. It is an IEEE standard language used to describe the structure and behavior of digital electronic systems. It allows the designer to describe how the electronic system is decomposed into subsystems and how the subsystems are interconnected. It uses programming language forms to specify the functions of a system. VHDL makes it much easier to describe very large circuits and systems [1].

A big advantage of VHDL is that it allows testing and verification of designs using simulations. The designer can simulate the behavior of a system using test inputs then compare the resulting outputs to the requirement model of the system. The mismatch in the comparison usually indicates design problems, thus enabling the designer to reexamine the design and correct the errors.

1.2 Parallel VHDL Simulation

Many digital electronic systems used in current and future industrial and military systems are too large to be effectively simulated on even state-of-the-art workstations. Currently, performance simulation is extremely slow and is a major bottleneck for the development of micro-electronic systems. Research conducted at Lockheed Martin under DoD sponsorship has shown that simulation of only a portion of a digital signal processing system can take over 20 hours [4]. As should be expected, actually building systems or prototypes is prohibitively expensive. The power of parallel machines, along with appropriate software development, is therefore essential to the maintenance and upgrade of existing systems and the development of future systems.

As a result, parallel simulation has attracted a considerable amount of interest. However, most research on parallel simulation is restricted to symmetric multiprocessor machines or MIMD machines with a small number of processors. There are few benchmark results available based on actual simulations of large circuits. The few existing empirical results are based on MIMD machines with only a small number of processors, typically tens of processors. Thus, the speed-ups attained compared to sequential simulation are severely limited. To simulate a system with up to millions of processes, the computational power of massively parallel processors (MPP) with several hundred processors is necessary [4].

1.3 MSU Parallel VHDL Simulation Project

1.3.1 Goal

The goal of the Parallel VHDL Simulation Project at Michigan State University is to develop and implement a new and efficient paradigm for VHDL simulations on massively parallel machines (MPP) and enable simulation speed-ups of up to two orders of magnitude. The simulator will use MPI, a standard parallel communication protocol. The developed parallel program will be scalable and portable.

From a user's point of view, the output of this development effort will be a fast parallel program that will be able to simulate a large digital system with up-to 100,000 VHDL processes at the performance level. By using this simulator, designers may be able to prove the functional correctness of digital system, assess the performance of candidate architectures of a digital system, and select the best architecture that meets requirements. The design cycle thus can be shortened, and the number of real prototypes to be constructed can be reduced.

In this project, a subset of VHDL constructs is selected to describe the performance and behavioral models. These selected VHDL constructs are powerful enough to describe the behavior and function of any hardware systems, yet simple enough so that models written using these constructs can be efficiently simulated in parallel.

1.3.2 Extensibility

Another goal of this simulation project is to ensure the portability and extensibility of the simulations. The simulation kernel is completely separated from the simulation object model. The simulation kernel only contains simulation protocol objects, such as TimeWarp, SynchObj, and ChandyMisraObj, etc. It does not know what object model is used to simulate the target system. On the other hand, the simulation object model does not know the simulation protocols at all. It only defines how target system objects should be translated into simulation objects. Thus once the target system has been translated into simulation objects using the object model, the resulting C++ code man be run on any parallel platforms. The end user only needs to select the parallel simulation protocol (TimeWarp, SynchObj, etc.) and set up the initialization parameters. The simulation kernel will then talk to the selected simulation protocol and run the simulation.

1.3.3 Major Tasks

The following is a list of major tasks to implement the parallel VHDL simulator:

- Design and implement the Simulation Object Model. Simulation objects will inherit properties (such as member functions) from these objects.
- Select a subset of VHDL constructs that are critical to describe the performance and behavioral models.
- Develop the Front End Interface which generates C++ programs from VHDL descriptions using the object model.
- Design and develop the general structure of Cockpit, the main module of parallel simulator.

- Develop algorithms and data structures of event queue handling for parallel simulation.

- Implement the Baseline programs to measure the efficiency of parallel programs. Implement and perform benchmark on SP2.

1.3.4 Major Modules

The parallel VHDL simulator software suite is broken into five modules:

- **Front End Interface**, which generates C++ classes and object interconnection information from VHDL descriptions.

- **Partitioner**, which distributes C++ objects into processing elements of a parallel computer.

- **Cockpit**, which is the main program module of the parallel simulator. It reads input (test) vectors, initializes modules, starts simulation, orchestrates other modules, and detects the termination.

- **Event Scheduler**, which manages events and schedules them according to parallel simulation protocols. For event handling and scheduling, it adopts a Time-Warp mechanism.

- **Communication Module**, which sends and receive messages from other processors.

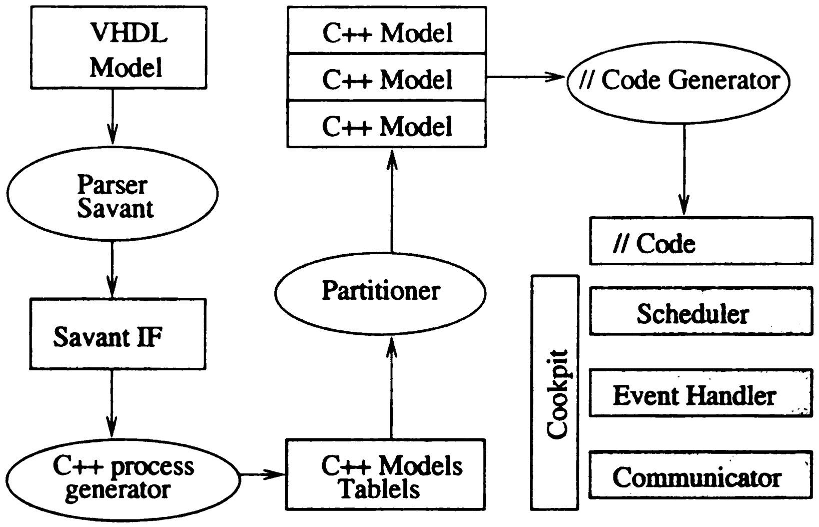

Figure 1.1: Structure of the Parallel Simulator

The Cockpit, Event Scheduler and Communication Module form the Simulation Kernel of the simulator. The structure of the simulator is shown by Figure 1.1 on page 6.

1.4 Front End Interface

The Front End Interface is the target of this thesis. The work I have done is only related to this part of this research project. To know more about other modules of the parallel simulator, please refer to [10] for details.

1.4.1 Function Description

The simulation kernel of this project is designed to run general parallel simulations. It is not designed only for VHDL simulations. The kernel is implemented in C++.

To simulate any real world system, a C++ description of the target system must be provided. This description must include domain objects and their interconnection information. The function of the Front End Interface is to provide such C++ descriptions for the system to be simulated. The Front End Interface has to use the pre-defined object model to generate the C++ code. The object model is discussed in details in Chapter 3.

When simulating VHDL systems, the Front End Interface will generate C++ code from VHDL source code. It will generate C++ classes corresponding to VHDL objects and their interconnection information. This is basically a translation from one language to another.

I am in charge of the C++ code generation part of the Front End Interface. Another research assistant of this project, Mr. Khashayar Rohanimanesh, implements the function of generating the interconnection information. Please refer to his research document [9] for details on this issue.

1.4.2 Approach

To translate VHDL into C++, the Front End Interface has to face the following problems:

- VHDL is a very complicated language. It is very hard to translate all its constructs and features into C++.

- There is no direct mapping between the two languages. Some intermediate form must be used to before the translation is performed.

- As a hardware description language, VHDL has some unique features, such as the "wait" statement. To keep the correct semantics of these features, the object model must implement mechanisms to support them.

The Parse Tree approach is a common method used to solve the translation problem. Generally, a parser is used to parse the source language and generate a parse tree. Translation is then performed by traversing the parse tree and taking publishing actions at each tree node. The parse tree can be constructed using either the source or the target language constructs, or some intermediate forms.

This approach is also used in this project. The SAVANT software package is used as the basis to implement the Front End Interface. SAVANT has developed a VHDL parser (SCRAM) and a set of intermediate forms (the AIRE specs), which is used to generate the parse tree. Chapter 2 describes the SAVANT software package in detail.

1.4.3 Selected VHDL construct subset

The following VHDL constructs have been selected as goals for the translation:

- Delay Mechanism : transport delay and inertial delay.

- Data Types: bit, integer, real, array types, record types, enumeration types, constants.

- Sequential Statements : signal assignment statement, variable assignment statement, multiple waveforms in one signal assignment, if-then-else, for loop, while loop, case statement, wait statement, logic/arithmetic operations.

- Concurrent Statements : process statement with sensitivity list, concurrent signal assignment statement, component instantiation statement, generate statement, conditional signal assignments.

These selected VHDL constructs are powerful enough to describe the behavior and function of very complicated hardware systems, yet they are simple enough so that model written using these constructs may be efficiently simulated in parallel. The task of the Front End Interface is implement these VHDL constructs.

Chapter 2

SAVANT

2.1 Overview

SAVANT stands for Standard Analyzer of VHDL Applications for Next-generation Technology. It is a joint effort between the University of Cincinnati and MTL Systems, Inc. to build an extensible, object-oriented intermediate form (IIR) for the hardware description language VHDL. The project is sponsored by the USAF Wright Laboratory and will produce a suite of software to analyze VHDL, build the IIR, and to output C++ code suitable for execution with the TyVIS VHDL simulation kernel developed by the University of Cincinnati.

The primary goal of SAVANT is to stimulate research among the VHDL community by providing an extensible, object-oriented, well-documented Intermediate Form (IIR) and a freely available analyzer to convert VHDL into the IIR. Because the IIR analyzer is released in source form, the additional derived classes can be inserted into the C++ class hierarchy. Thus, user actions can benefit fully from the fact that the

IIR is object-oriented. Consequently, no procedural interface is provided or needed [5].

2.1.1 Components of SAVANT

The SAVANT software suite contains the following components:

- **Scram: A VHDL Analyzer**

  The SCRAM analyzer inputs a VHDL description, check it for syntactic and static semantic correctness, and stores it in the intermediate form (IIR). The SCRAM parser is constructed as an LL(2) grammar and uses the Purdue Compiler Construction Tool Set (PCCTS) parser generator. SCRAM is written in C++ using g++ 2.7.2 for development.

- **The Intermediate Form**

  The intermediate forms are jointly developed with John Willis of FTL Systems, Inc. The intermediate form standard is called AIRE and includes definitions of two intermediates, a memory resident data structure called IIR and a machine-independent file data structure called FIR.

- **Transmute: Derived Classes that Implement Static Equivalence**

  The transmute method is a collection of derived classes that support rewriting of the IIR into a reduced form.

- **Archiver: Derived Classes for Library Management**

  The collection of archiver classes support library management. Within the derived classes two methods are defined, namely: record() and playback(). record() writes the IIR to an FIR file and playback() reads an FIR file into the IIR.

- **Publisher: Derived Classes for Output Generation**

  Output generation in the SAVANT software suite is supported by the collection of publisher classes. Two overloaded methods, _publish_vhdl() and _publish_cc(), are defined for all nodes of the reduced IIR. The _publish_vhdl() method simply regenerates VHDL from the internal IIR. The _publish_cc() method produces C++ simulation code to link with the TyVIS VHDL simulation kernel.

The relationship between the components of SAVANT is described by figure 2.1 on page 13. For more details about these individual components, please refer to [5].

2.1.2 How to Use SAVANT

SAVANT has implemented the VHDL Analyzer and the Intermediate Form, which are needed by our parallel VHDL simulation project. On the other hand, SAVANT has also implemented a C++ Publisher, which makes use of the Analyzer to produce C++ code for the TyVIS VHDL simulation kernel [6]. TyVIS is a completely different simulation kernel from our simulation kernel, thus the C++ code generated by SAVANT can not be used by our simulation kernel at all. But we can modify the SAVANT publisher to generate code for our own kernel. By avoiding rewriting a separate C++ publisher, we can reuse lots of SAVANT code and save lots of time.

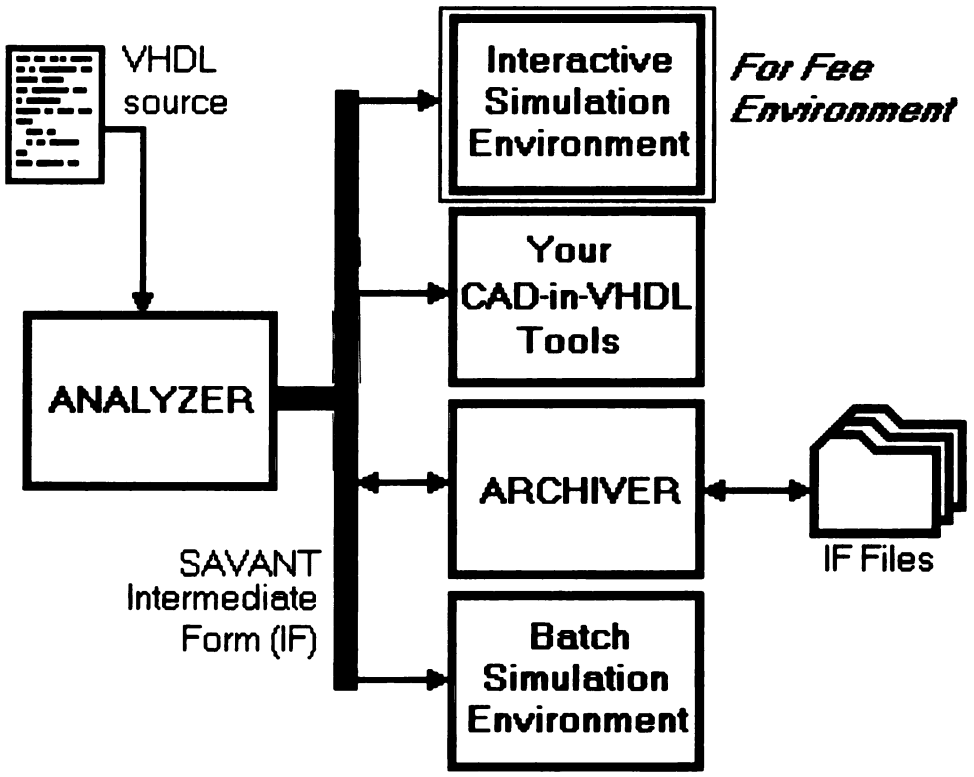

Figure 2.1: Components of SAVANT Project

The function of the Front End Interface can now be described as to modify the SAVANT Publisher to generate C++ code for our own parallel simulation kernel.

To modify the SAVANT Publisher, it is crucial to understand the structure of the AIRE standard and its implementation in the SAVANT project.

2.2 The AIRE Standard

The AIRE Standard was initiated by the Wright Labs, FTL Systems, and the University of Cincinnati. It is a specification for a freely available, highly portable, extensible VHDL intermediate form. Its specification is defined by [7].

2.2.1 Purpose

The Advanced Intermediate Representation with Extensibility (AIRE) specifications address the HDL user's need for efficient mechanisms for sharing of design information between tool components. The AIRE specification includes coordinated Internal Intermediate Representation (IIR) and File Intermediate Representation (FIR) specifications.

The AIRE representation is useful following source code analysis, integration of separately analyzed units, elaboration, in-line expansion, machine-independent optimization, back-end code generation/synthesis and execution (simulation). Integration of component tools from different development groups should finally be realizable with modest effort using this intermediate representation. As a result both the research and design community benefit from stronger and more useful tools.

AIRE's purpose is to meet the evolving needs of advanced VHDL tool developers. Therefore, the specification is still evolving. Some changes to the core specification are still being made as a result of experience with early implementations. Implementation-specific extensions are being added to the core specification.

2.2.2 Approach

AIRE uses a collection of object (record or structure) instances linked by pointers to represent design information. These objects represent analyzed, elaborated, and executable instances of specific design information. In very general terms, the collection of objects represents a very generalized abstract syntax tree (AST), while methods associated with the classes (and thus objects) represent an integrated application programming interface (API).

2.2.3 Class Hierarchy

The IIR class hierarchy is shown abstractly in Figure 2.2 on page 16. A base class, called IIR, is located at the top of the class hierarchy. Derived classes, descended from the base IIR class, are instantiated in order to form a specific design representation. There are total 227 classes defined by the standard. All the classes are single rooted at class IIR. Figure 2.3 on page 17 is an example of the IIR class derivations.

An IIR-derived class may be instantiated by other IIR classes (such as declaration lists within a block declaration), by friends of an implementation-specific foundation (such as a source code analyzer intrinsic to the foundation), or by external application

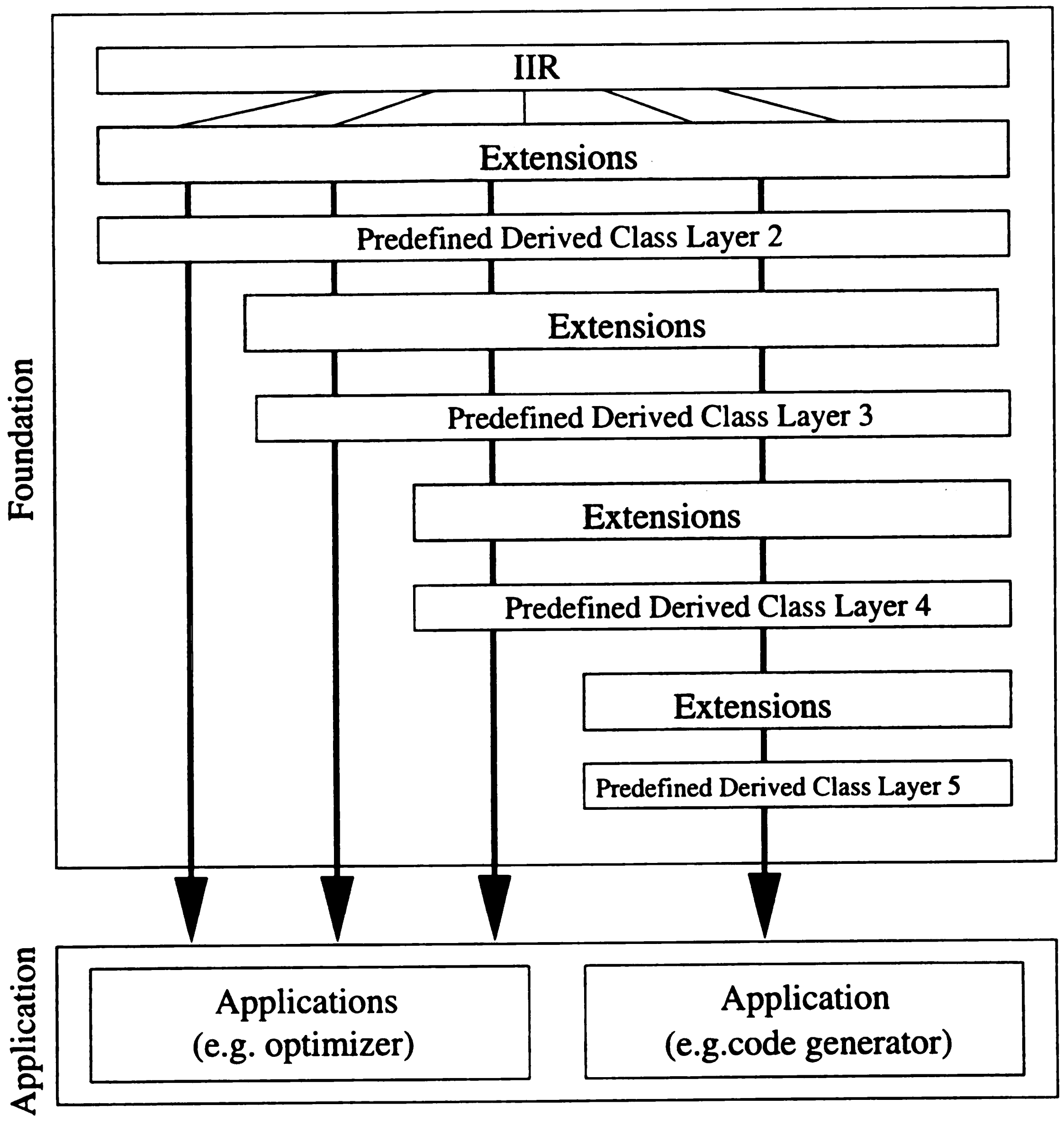
IIR
Extensions
Predefined Derived Class Layer 2
Extensions
Predefined Derived Class Layer 3
Extensions
Predefined Derived Class Layer 4
Extensions
Predefined Derived Class Layer 5
Foundation
Application
Applications (e.g. optimizer)
Application (e.g.code generator)

Figure 2.2: Structure of IIR and Applications

- IIR
  - IIR_DesignFile
  - IIR_Comment
  - ...
  - IIR_SequentialStatement
    - IIR_AssertionStatement
    - IIR_BreakStatement
    - ...
  - IIR_ConcurrentStatement

Figure 2.3: Example of the IIR Derivation Hierarchy

code (such as the optimizer and code generator shown in Figure 2.2). The public methods and public data elements are sufficient to construct, get and set all language functionality defined by the associated system design language.

2.2.4 Extensibility

Extension classes, shown in darker shading in Figure 2.2, may be interposed within the IIR inheritance hierarchy in order to add application-specific methods or (in instantiable classes) additional, application-specific data elements. Each extension layer begins derivation with a class named by pre-pending IIRBase_. The extension layer(s), if present, result in a final, predefined layer named by pre-pending IIR_ to the class name. The predefined properties of this layer are explicitly specified in this document; the predefined properties of the base layer for each class may be readily (even automatically) derived. The names of any intervening extension classes should assume the form IIR*ExtensionDesignator_SpecificClassName*. For example, a synthesis application might interpose an extension class IIRSyn_ProcessStatement between IIRBase_ProcessStatement and IIR_ProcessStatement.

Since a complete tool may be composed of more than one application, the darker shaded extension layers actual represent zero or more layers within the class hierarchy. For example, an elaborator application may add some extensions to the block and process declarations while a code generator adds its own extensions. In order to clearly delineate identifiers other than those predefined constructors, destructors or operators, such identifiers are prefaced by a leading underscore (_). Name conflicts are inevitable when linking two applications which both use the same extension designator, such as "Syn" in the example above.

All IIR class constructors don't have arguments, but there are no constraints on the destructors. Some intermediate classes in the IIR derivation hierarchy are not directly instantiable, such as the IIR_SequentialStatement class. These interior classes serve purely as parent classes for classes that process common characteristics. They are indirectly instantiated by the instantiation of their children classes.

Whereas the specification is generally based on the C++ language and single-inheritance C++ classes, implementations using C, Ada or Pascal are very feasible. However since AIRE takes advantage of modern, object-oriented programming techniques which may not be present in older programming languages, AIRE can be a somewhat less convenient when other languages are used.

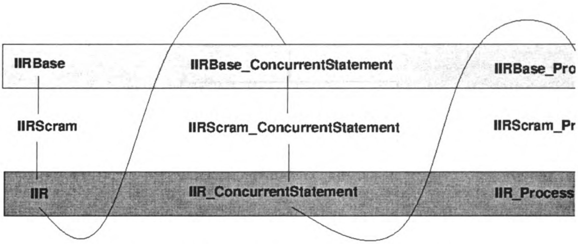

Figure 2.4: The SAVANT Derivation of AIRE

2.3 The SAVANT Implementation of AIRE

2.3.1 Class Derivation

The actual class derivation of AIRE in SAVANT is illustrated by Figure 2.4 on page 19. There are basically three layers in the derivation hierarchy: IIR layer, IIRBase layer, and the IIRScram layer.

The *IIRBase* layer classes contain public interface (data and methods) defined in the AIRE specification. It also contains protected or private data and methods that support the public interface.

The *IIRScram* layer classes implement the VHDL Analyzer (named "scram") and the C++ publisher. It resides below the IIRBase layer and above the IIR layer. The IIRScram layer classes inherit the public data and methods defined by the IIRBase layer classes. The classes in this layer contain type checking routines and symbol table management functions for scram. They also contain methods, such as **_publish_cc()**

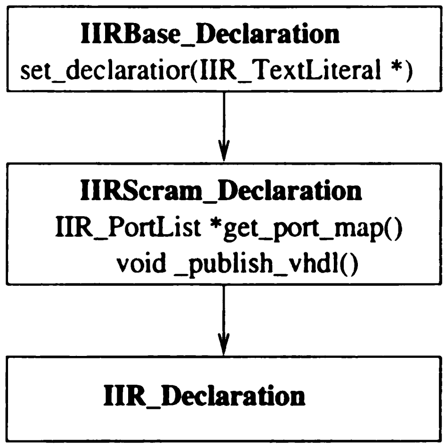

Figure 2.5: Implementation of IIR_Declaration

and `_transmute()`, to implement the analyzer and the publisher,

The *IIR* layer classes contain empty constructor and destructor. There is no other private, public and protected data or functions declared. The IIR layer is the bottom layer. The IIR layer classes inherit all the public data and methods declared by the classes in the other 2 layers.

All AIRE classes occur as 3 SAVANT Classes Definitions. Figure 2.5 on page 20 shows an example of this structure.

2.3.2 Extensibility in SAVANT

There are several reasons why SAVANT uses this structure. The most important one is to enable the extensibility without modifying code in the scram parser. Application specific functionality can be added by introducing extra layer(s) into the class hierarchy, thus avoiding changing code of the IIRScram layer completely.

The scram "parser" instantiates concrete classes in IIR. Essentially, the user-

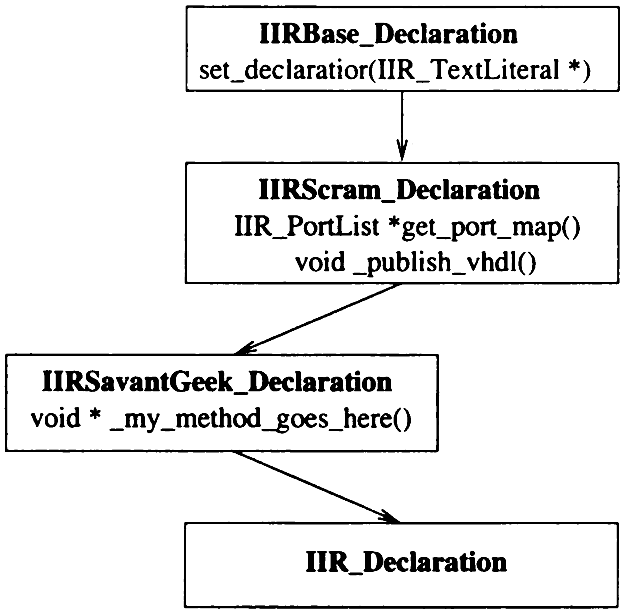
IIRBase_Declaration
set_declaratior(IIR_TextLiteral *)
IIRScram_Declaration
IIR_PortList *get_port_map()
void _publish_vhdl()
IIRSavantGeek_Declaration
void * _my_method_goes_here()
IIR_Declaration

Figure 2.6: Insert IIRSavantGeek_Declaration Class into the Hierarchy

defined class must be derived from the layer just "above" the IIR layer (the IIRScram layer). In addition, the derivation of the IIR level must be modified in such a way that it inherits directly from the user defined class.

Figure 2.6 on page 21 shows an example of how to insert a user defined class into this hierarchy. By inserting the IIRSavantGeek_Declaration class, the user now can define application-oriented functions of the IIR_Declaration class. If the user wants to shadow a function defined by the IIRScram layer, he can simply put a new definition of that function in the new class. For example, if the user defines the **_publish_cc()** function in IIRSavantGeek_Declaration, the same function in IIRScram_Declaration will be shadowed.

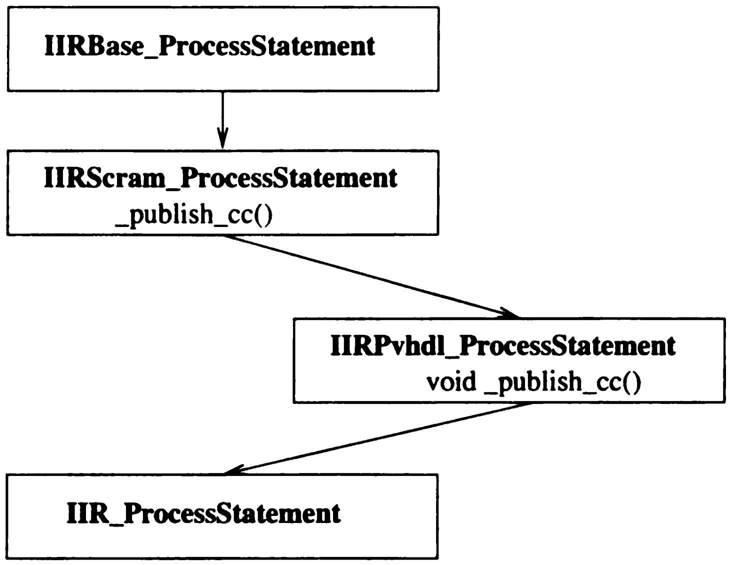

Figure 2.7: The New Pvhdl Layer

2.4 How to Modify the SAVANT Publisher

The SAVANT Publisher is implemented in the IIRScram layer. The *IIRScram class* is the root class of the IIRScram layer. The way SAVANT implements the Publisher is by defining a virtual function **_publish_cc()** in the *IIRScram class.* Since all other IIRScram layer classes are child classes of the *IIRScram class*, they inherit the **_publish_cc()** function. This function is then overloaded by each IIRScram layer class to publish C++ code according to different semantics.

After the scram parser generates the IIR parse tree, the C++ code is published by calling the **_publish_cc()** method from the root of the tree. The root will then call the **_publish_cc()** method of higher level nodes in the tree. This process is continued recursively until all the tree nodes have called the **_publish_cc()** function.

The reason SAVANT implements the publisher as a virtual function is to ensure polymorphism at run time. When a virtual function is invoked via a pointer, the actual function to be executed is determined by the type of the object that is stored at the memory location being accessed, not by the type of the pointer. The determination of which virtual function to use is thus made during run time [3, Chapter 15.6]. As a result, when an IIR layer object needs to call the _publish_cc() function of another IIR layer object, it can simply use the base IIR pointer to reference that object. The correct **`_publish_cc()`** function will be determined at run time. This process greatly simplifies the programming.

To modify the SAVANT Publisher, a new *IIRPvhdl* layer is introduced into the SAVANT hierarchy between the IIRScram layer and the IIR layer. The IIRPvhdl layer classes overload the **`_publish_cc()`** method to produce C++ code for the MSU parallel simulation kernel. Since the IIRPvhdl layer sits between the IIRScram layer and the IIR layer, the **`_publish_cc()`** function in the IIRScram layer is showed. When the nodes in the IIR parse tree call the **`_publish_cc()`** function, the function defined in the IIRPvhdl is executed. This approach is illustrated in Figure 2.7.

Chapter 3

Translate VHDL Constructs into C++

3.1 The Object Model

The Front End Interface needs to generate C++ according to the Simulation Object Model. This section describes the details about the Object Model.

The object model defines a "BasicObject" class which characterizes the common features of all simulation objects. When translating a target system into simulation classes, the BasicObject class should be used as the parent class of all resulting simulation classes. The BasicObject class defines input signals, output signals, states, and a method called `executeProcess()`. Through the input and output signals, BasicObject classes can interact with each other. States are used by BasicObject to keep private information. The `executeProcess()` method describes the behavior of the object (how the output signals should be changed according to the input signals).

The input signal, output signal and state are also classes defined by the object model. All signals and states must be "registered" before then can be used. Please refer to [10] for details about the BasicObject class, the signal classes, and the state class.

When translating a VHDL system into C++ code, each VHDL process is translated into a simulation class. The "in" signals in the VHDL process are translated into the input signals in the simulation class. The "out" signals are translated into output signals. The process variables are translated into states. The sequential statements of the VHDL process are translated line by line into the **`executeProcess()`** method. Generally, each C++ class has the following items:

- Declaration of input/output signals and states.
- Registration of input/output signals and states.
- Initialization of input/ output signals and states.
- The **`executeProcess()`** method.

Format of Object Class

Each simulation object class contains a data declaration section, a constructor, and a **`executeProcess()`** method. All data members and methods are defined as public (so that the simulation kernel can access them directly). In the constructor, signals and states are registered and initialized.

Figure 3.1 on page 26 shows an example C++ class generated using the object model. This example only shows a template of the object model class. It is not the exact code generated by the Front End Interface. Appendix B shows the VHDL

```
class DFF : public BasicObject
{ public:
    InSignal D, CLK;              // declarations
    OutSignal Q;
    State prev;

    DFF(): BasicObject() {        // registrations
          registerInSignal (&D);
          registerInSignal (&CLK);
          registerOutSignal (&Q);
          registerState (&prev);

          D=X; CLK=X; prev=X;     // initializations
          Q=Y;
    }

    void executeProcess () {      // actions
       int val;
       if (hasEvent(&D) && (CLK == '1')) {
           val = D;
           if (prev != val) {
                prev = val;

                if (prev == '0')  D = 0;
                else if (preve == '1') D =1;
                else D = val;
    } } }
};
```

Figure 3.1: An Example C++ Class of the Object Model

source code of an AND gate. The exact C++ code for this AND gate is shown in Appendix C.

Basic Data Types

VHDL has 3 basic data types: bit, real, integer. Table 3.1 on page 27 show their corresponding C++ implementation. Other complicated data types, such as array

VHDL Type	*C++ Type*	*Implementation*
bit	SavantbitType	typedef int SavantbitType
real	SavantrealType	typedef double SavantrealType
integer	SavantintegerType	typedef int SavantintegerType

Table 3.1: Basic VHDL Data Types and Their C++ Implementations

and record, are all derived from these basic types.

Declaration of Signals and Variables

In VHDL, signals are declared either in entity declaration or architecture declaration. In this project, the unit of publishing is neither entity nor architecture, but VHDL process. Each VHDL process is translated into a C++ class.

It is possible that a VHDL process could use only several of all the signals declared by the architecture body embracing it. It is also possible that one architecture body contains several processes and they share some signals together. The rule here is that each VHDL process class declares only those signals it uses. As a result, it is possible one signal is declared in different process classes. As how these signals are resolved, please refer to [9].

For each VHDL process, there are only two kinds of signals: in signals and out signals. When declaring these signals, prefix "`in_`" is added to each in signal, and prefix "`out_`" is added to each out signal. If a signal is an "in_out" signal, two separate signals are declared. For example, a signal assignment statement may look like this: `a <= not a after 3 ns`. Signal `a` is on both sides of this signal assignment statement and is an "in_out" signal. As a result, there will be two signals declared for signal `a`, namely `in_a` and `out_a`. When generating the interconnection information, these two

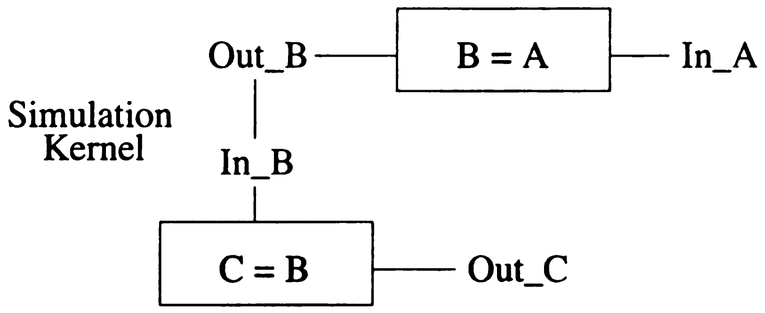

Figure 3.2: Naming Scheme of VHDL Singals

signals are connected directly [9]. The naming scheme of VHDL signals is shown in Figure 3.2 on page 28.

VHDL variables are declared within processes (global variable has not been implemented yet) and are translated into "states". In contrast to signals, states are declared without a prefix. For example, if a VHDL process declares a variable as: `variable a : bit`, its C++ declaration is `SavantbitType a`.

Signals and variables can have initial values. It is in the constructor that their initial values are granted. Later sections will address the issue on how to find which signals and variables are used by a process and how to retrieve their initial values.

The Constructor

The class constructor will register signals and variables and set their initial values. To register signals and states, the constructor will call functions declared in the `BasicObject` class. Table 3.2 shows these functions.

No special function needs to be called to initialize signals and states. The simple C++ *assignment operator* is used. For example, if the initial value of signal `a` is `0`,

input signal	`registerInSignal(&signal,sizeof(signal))`
output signal	`registerOutSignal(&signal,sizeof(signal))`
state	`registerState(&state,sizeof(state))`

Table 3.2: Functions for Signal/Variable Registration

signal/state	initial value	definition
input signal	X	#define X -1
output signal	Y	#define Y -1
state	X	as above

Table 3.3: System Defined Initial Values

a is initialized by **a** = 0. If the signal or state doesn't have an initial value, system defined initial values are used. Table 3.3 shows the predefined initial values.

These initial values are defined in file **SavantGlobal.h** shown in Appendix D.

The **executeProcess()** Method

The **executeProcess()** method of the object class describes the actions of the VHDL process. It is a line to line translation of VHDL statements to C++ code .

VHDL process is composed of VHDL sequential statements, such as signal assignment statement, variable assignment statement, if statement, etc. These VHDL language constructs all have corresponding IIR representations. The way to translate these VHDL language constructs into C++ code is to call the **_publish_cc()** function in their corresponding IIR nodes in the IIR parse tree. Since each IIR class has its own **_publish_cc()**, different semantics of different VHDL constructs can be translated by implementing the **_publish_cc()** method differently. Later sections will describe this approach in detail.

The **executeProcess()** method is defined in the BasicObject class. The simulation kernel simulates the VHDL description by calling the `executeProcess()` method of each VHDL simulation object.

Modification Guidelines

In the SAVANT project, the IIRBase layer and the IIR layer are well defined by the AIRE standard. They are well documented by the AIRE standard [7]. The IIRScram layer implemented the VHDL Analyzer and the Publisher. This layer contains most of the programming tasks. However, this layer is very poorly documented. Actually their is no documentation at all which describe the programming details of the IIRScram layer.

The C++ code is generated by traversing the IIR parse tree. The information collected by the Analyzer has to be used to perform the publication. Since the IIRScram layer has no documentation, the only way to find out all this information is to use a debugger to trace through the program. In this project, the GNU gdb is used to debug the SAVANT executable file, Scram. The basic debugging process is to first set a break point at the **`_publish_cc()`** method of the IIRScram class being modified , then trace into all pertinent sub-routines and relevant data members. There are 227 IIR classes defined and the size of the Scram file is about 35MB. Thus the process of using a debugger to debug the file to find out some information can be extremely time consuming and painful.

The following are the general steps taken to modify the SAVANT publisher:

- Look at the AIRE standard to find out what public data member and public functions are declared by the target IIRBase class. These data and functions are will be used in generating the C++ code.

- Debug the `_publish_cc()` function of the target IIRScram class to see how this information is used.

- Create a new IIRPvhdl layer class which implements a new `_publish_cc()` method to shadow its corresponding IIRScram layer class.

3.2 Implementation Details

This section discusses the details of how the IIRPvhdl layer classes are generated. As mentioned earlier, this project tries to handle only those VHDL constructs that are deemed as essential to VHDL simulation. It does not try to handle all VHDL language constructs. This section will use the VHDL construct as the unit of discussion.

3.2.1 How to Publish

SAVANT defined a utility class called "switch_file". This class deals with input/output streams. It defines a method `void set_file(char *name, char *ext)`, which is used to set the name of the output file for the Publisher. If a file with the name "name.ext"already exists, future outputs of the Publisher will be appended to the end of this file. Otherwise, a new file with that name is created and future outputs of the Publisher will be written to the new file.

In IIRScram.hh, two global variables are declared as follows:

```
extern switch_file _vhdl_out;  //file for vhdl output
extern switch_file _cc_out;    //file for c++ output
```

The IIRScram class is the root of the IIRScram layer classes. Thus the above two global variables could be accessed by any IIRScram class. The `_vhdl_out` is only used to publish VHDL source code, while the `_cc_out` is only used to publish C++ code. For example, to publish code " i += 1;" to file "test.cc", you only need to do the following:

```
_cc_out.set_file(''test'', ''.cc'');

_cc_out <<  '' i += 1;'';
```

Any IIRPvhdl layer class is derived directly from a IIRScram layer class. As a result, all IIRPvhdl layer classes can also access `_cc_out` and publish C++ code to a named file.

3.2.2 Where to Publish

In SAVANT, each class corresponding to a VHDL process will generate two files, a ".hh" file and a ".cc" file. When all the C++ files have been generated, a "Makefile" is created to tell the TyVIS simulation kernel how to link all the files together.

In this project, all C++ code is published in the file "Classes.h". The constructor and the **executeProcess()** method are all inline functions. Thus there is no need to create a Makefile. The simulation kernel can simply include the "Classes.h" file to get all class definitions.

3.2.3 How to Insert an IIRPvhdl Class

The sole purpose of adding an IIRPvhdl layer is to shadow the _publish_cc() functions of the IIRScram layer classes. To do this, an IIRPvhdl layer class has to be derived directly from the IIRScram layer class. The IIR layer class will then be derived from the IIRPvhdl layer class instead. Thus when an IIR layer node calls the _publish_cc() function, it will call the _publish_cc() declared in the IIRPvhdl layer class, not in the IIRScram layer class. Figure 2.6 on page 21 illustrated this scheme.

As an example, let's look at how to insert the IIRPvhdl_ProcessStatement class between the IIRScram_ProcessStatement and the IIR_ProcessStatement class. The original IIR_ProcessStatement.hh is like this:

```
#include "IIRScram_ProcessStatement.hh"
class IIR_ProcessStatement : public IIRScram_ProcessStatement {
    ...
}
```

After adding the IIRPvhdl_ProcessStatement class, the IIR_ProcessStatement needs to be derived from the IIRPvhdl_ProcessStatement class. The IIR_ProcessStatement.hh is then modified like this:

```
#include "IIRPvhdl_ProcessStatement.hh"
class IIR_ProcessStatement : public IIRPvhdl_ProcessStatement {
    ...
}
```

On the other hand, the IIRPvhdl_ProcessStatement has to be derived directly from the IIRScram_ProcessStatement class. The IIRPvhdl_ProcessStatement.hh looks like this:

```
#include "IIRScram_ProcessStatement.hh"
class IIRPvhdl_ProcessStatement : public IIRScram_ProcessStatement {
    ...
}
```

Other IIRPvhdl layer classes should be added to the class hierarchy in similar manner.

3.2.4 The IIRPvhdl_DesignFile Class

The predefined IIR_DesignFile class represents the textual contents of a design file. These contents may include one or more IIR_LibraryUnits and/or one or more IIR_Comments. The IIR_DesignFile class defines a public data member **IIR_LibraryUnitList library_units**. This data member is a list of entity declarations and architecture declarations which cluster all VHDL library units into a design file together, so that they could be accessed one by one.

The IIRPvhdl_DesignFile class has basically 3 functions:

1. Use preprocessor **#ifndef** ... **#define** to protect the "Classes.h" file.

2. Include some ".h" files and use "typedef" to define some data types, like **SavantbitType**, **SavantintegerType**, and **SavantrealType**.

3. Call `library_units._publish_cc()` to let the lower level tree node publish C++ code.

The implementation of this class is pretty straightforward. Please refer to [13, IIRPvhdl_DesignFile.hh, IIRPvhdl_DesignFile.cc] for details.

The `_publish_cc()` method of IIRScram_LibraryUnitList is not overloaded because it has already done the correct things. This method goes through each library unit in the list and calls their `_publish_cc()` method. This is the desired behavior of the `_publish_cc()` for IIR_LibraryUnitList . As a result, there is no need to define an IIRPvhdl_LibraryUnitList class to shadow the IIRScram_LibraryUnitList class. For implementation detail of the IIRScram_LibraryUnitList class, please refer to [12, IIRScram_LibraryUnitList.hh, IIRScram_LibraryUnitList.cc].

From here we can see that if the `_publish_cc()` function of an IIRScram layer class has implemented the desired functions, then there is no need to defined its corresponding IIRPvhdl layer class to shadow it. As a result, not every IIRScram layer class has a corresponding IIRPvhdl layer class.

3.2.5 The IIRPvhdl_EntityDeclaration Class

The predefined IIR_EntityDeclaration class represents VHDL entities. It is a child class of the IIR_LibraryUnit class and contains several predefined public data elements shown in Table 3.4.

These data elements are all lists which keep entity related information, such as generic declarations, ports, etc. The IIRScram_EntityDeclaration class uses all of

Data Member Type	*Data Member Name*
IIR_GenericList	generic_clause
IIR_PortList	port_clause
IIR_DeclarationList	entity_declarative_part
IIR_ConcurrentStatementList	entity_statement_part
IIR_DesignUnitList	architectures

Table 3.4: Predefined Public Data Elements of IIR_EntityDeclaration

them in its `_publish_cc()` method. None of them have been used in this project so far. The desired behavior of the IIR_EntityDeclaration is to publish nothing at all. Thus the `_publish_cc()` method of IIRPvhdl_EntityDeclaration class is just empty.

It is possible to make changes to this class if new features need to be implemented in the future, such as generic constants. It probably would require corresponding changes in other part of the front end interface. This is left to the decision of future participates of this project.

3.2.6 The IIRPvhdl_ArchitectureDeclaration Class

The predefined IIR_ArchitectureDeclaration class represents one of several potential implementations of an entity. Like the IIR_EntityDeclaration class, it is also a child class of IIR_LibraryUnit.

The IIR_ArchitectureDeclaration Class has several predefined public methods and public data elements. Among them, the **`IIR_EntityDeclaration * get_entity()`** method retrieves the pointer to the entity to which the architecture is associated. The **`IIR_ConcurrentStatementList architecture_statement_part`** data member is a list pointing to all the current statements in the architecture body.

The IIRPvhdl_ArchitetureDeclaration class does two things. First, it prints a message to the standard output showing the name of the architecture being processed. Second, it calls the `_publish_cc()` method of `architecture_statement_part` which causes the higher lever nodes in the parse tree to publish. The first task is performed by printing the names to standard output "cerr". To get the name of the architecture, the `_get_declarator()` function is called. This function is defined as a virtual function in the IIRScram class. It can be used by any node in the parse tree to get its declarator string.

The implementation of this class is straightforward. For details please refer to [13, IIRPvhdl_ArchitectureDeclaration.cc].

3.2.7 The Process Statement

The SAVANT predefined IIR_ProcessStatement class represents a sequential declarative region and single thread of execution. Such processes must appear within an architecture, concurrent block statement or concurrent generate statement.

The VHDL process is the translation unit of the C++ publisher. Each process will be translated into a simulation class. In the simulation class, signals and variables that are used in the VHDL process are translated into C++ data elements and registered to the simulation kernel. This section discusses how to translate VHDL process into simulation class.

Data Elements

There are two predefined public data elements in IIR_ProcesssStatement class:

- **IIR_DeclarationList process_declarative_part**, which is a list of declaration items, such as local variables.

- **IIR_SequentialStatementList process_statement_part**, which is a list of all sequential statement in the process.

The IIRPvhdl_ProcessStatement.hh declares several public and private data elements. There are two public data elements:

- **IIR_Char *class_name**, which is a character pointer used to keep the name of this class. This information will be used when generating the interconnection information.

- **IIR_Int32 class_id**, which is a integer used to keep the ID of the class. In the **_publish_cc()** method of IIRPvhdl_ProcessStatement class, a variable **static int type_id=0** is declared to generate an unique ID for each simulation class. The **class_id** uses the value of **type_id**. It will be used in generating interconnection information.

The IIRPvhdl_ProcessStatement.hh declares five private data elements. They are all used as local variables. These data elements are:

- **set<IIR_Declaration> sig_in_list**, which is the set for input signals.

- **set<IIR_Declaration> sig_out_list**, which is the set for output signals.

- **int in_sig**, which is used to count the total number of input signals.

- **int out_sig**, which is used to count the total number of output signals.

- `int state_num`, which is used to count the total number of states.

These variables will be used when publishing the constructor of the BasicObject class, which needs to know the number of input/output signals and states in the VHDL process.

Input/Output Signals

Before further discussion, it is necessary to define "input" signal and "output" signal first. In a VHDL process, if the value of a signal is referenced, this signal is an *input* signal. For example, signals appear on the right hand side of signal assignment statements or "if" statements are input signals. If a signal's value is changed in the VHDL process, it is an *output* signal. For example, signals appear on the right hand side of signal assignment statements are output signals. It is possible that a signal is both an input signal and an output signal. For example, signal a in VHDL statement `''a <= not a''` is both an input signal and an output signal.

In VHDL, both entity declaration statement and architecture declaration statement declare signals. With in an architecture body, there could be several processes. This means a VHDL process may not cover all signals declared by the architecture. Only those signals that are covered by the process should be translated when translating this process into a simulation class. This presents the problem of finding what signals are used by the process.

The SAVANT VHDL Analyzer has solved this problem. In IIRScram.hh, a virtual function

```
void _get_list_of_input_signals(set<IIR_Declaration>* list)
```

is declared. Its function is to add pointers of all input signals associated with an IIR node to "list". Since "list" is defined as a set object, the same signal pointer will be added to the set only once (please refer to [12, set.hh] for details). This function is overloaded by all other IIRScram layer classes.

The IIRScram_ProcessStatement class defines this function like this:

```
void IIRScram_ProcessStatement::
_get_list_of_input_signals(set<IIR_Declaration>* list) {
  process_statement_part._get_list_of_input_signals(list);
}
```

The process_statement_part data element is of type IIR_SequentialStatementList. In the IIRScram_SequentialStatementList class, the above function is defined as follows:

```
void IIRScram_SequentialStatementList::
_get_list_of_input_signals(set<IIR_Declaration>* list) {
  IIR_SequentialStatement* stmt = first();
  for(; stmt != NULL; ) {
    stmt->_get_list_of_input_signals(list);
    stmt = successor(stmt);
  }
}
```

As a result, each sequential statement in the VHDL process will call its `_get_list_of_input_signals()` to put its input signals into "list".

Similarly, another virtual function

```
void _get_signal_source_info(set<IIR_Declaration>* siginfo)
```

is defined in IIRScram.hh to collect output signals using set.

The set utility class defines a function named `make_list()`. This function first creates a list that contains all data in the set then returns the pointer of the list. After the input signal set and the output signal set have been obtained, the two sets call the `make_list()` function to create two signal list.

In file IIRPvhdl.hh, two global variables are declared to keep the two list:

```
extern dl_list<IIR_Declaration> *_proc_in_sig_list;

extern dl_list<IIR_Declaration> *_proc_out_sig_list;
```

These two signal lists will be used by other IIRPvhdl layer classes, such as IIR-Pvhdl_IndexedName.

In SAVANT, it is possible that the same array element appears more than once in the input or output signal list. The reason is that SAVANT generates an IIR_IndexedName node each time it encounters an array element signal. For other of signals, SAVANT generates only one IIR node, no matter how many times this signal appears. For example, for the following VHDL process:

```
 process begin

   a <= c and arr(1);
```

```
    b <= c or arr(1);

  end process;
```

SAVANT generates only one IIR node for "c", but two IIR nodes for "arr(1)". When generating the input signal set, their is only one IIR node for "c" added to the set. But there are two "arr(1)" IIR nodes added. This seems strange, but this is SAVANT does handle this problem this way. It took me lots of effort to find out this problem. The reason is that TyVIS requires different object for different array element. Since the SAVANT publisher is TyVIS oriented, thus it handles this situation in this strange way.

In our project, only one array element should be put in the signal list. As a result, a function named

```
void
_clean_sig_list_for_identical_elements(dl_list<IIR_Declaration> *list)
```

is defined in IIRPvhdl.hh. The function of this method is to remove multiple copies of array elements from a signal list. Its implementation will be discussed in later sections. After calling `make_list()` to get the input signal list and the output signal list, the `_clean_sig_list_for_identical_elements()` function is called on both signal lists.

Wait Statement List

VHDL process usually contains wait statement. There is no language construct in C++ which directly corresponds to the semantics of the VHDL wait statement. As a result, wait statement has to be implemented according to its semantics.

In SAVANT, for each process, it is necessary to find out the number of wait statements and keep them in a list. To achieve this, SAVANT defines a virtual function **void _build_wait_list(dl_list<IIRScram_WaitStatement> *)** in the IIRScram_SequentialStatement class. Since IIRScram_SequentialStatement class is the root class of all sequential statement classes in the IIRScram layer, each sequential statement class will inherit then overload this function. For example, both IIRScram_IfStatement class and IIRScram_WhileStatement class are child class of the IIRScram_SequentialStatement class. They may all contain wait statement. As a result, they all overload the **_build_wait_list()** function.

The IIRScram_ProcessStatement class defines a public data element named **dl_list<IIR_WaitStatement> _wait_stmt_list** to keep the wait statement list. To build the list, a while loop is used to call the **_build_wait_list()** function of each sequential statement node in the **process_statement_part** data element. The code is as following:

```
stmt = process_statement_part.first();
while (stmt != NULL) {
  stmt->_build_wait_list((dl_list<IIRScram_WaitStatement>*)&_wait_stmt_list)
  stmt = process_statement_part.successor(stmt);
}
```

Details on how to handle wait statement will be discussed in later sections.

Name of Class

Since each VHDL process is translated into a class, it is important that they all have different names. To achieve this, the `_publish_cc()` method of IIRPvhdl_ProcessStatement class defines a static integer **`type_id`**. The initial value of **`type_id`** is 0. Each time after publishing a class, the value of **`type_id`** is increased by one. Since **`type_id`** is used as part of the class's name, each class will have a different name.

The naming scheme of a class is like this : "current architecture name" plus "_of_" plus "current entity name" plus "_class" plus "type_id". The name of the class is stored by the **`class_name`** data element described above. Also, each class name and its corresponding "type_id" is written to a file named "Classes.id" for examination purpose.

Publishing Class Header

As discussed in previous sections, **`_cc_out`** is used exclusively to publish C++ code. The first thing in publishing the code is to set the output file name to "Classes.h". The first line of code should be the class header, like: **`class class_name : public BasicObject`** This could be done easily by the following code:

```
_cc_out.set_file("Classes", ".h");
_cc_out << "class " << class_name << " : public BasicObject" << "\n";
```

Other C++ code could be published in the same way. It is not necessary to set the publishing file to "Classes.h" each time new code is written to that file. It is only when the publishing file name is changed, does it necessary to change the file name back to "Classes.h". Also, don't forget to write "endl" to start a new line in "Classes.h".

Signal Declarations

Signal declarations in the class are not different from the common "type + name" format. There are two kinds of signals, input signals and output signals. As described above, global variable `_proc_in_sig_list` is the list of input signals and `_proc_out_sig_list` is the list of output signals of the VHDL process being translated. To publish these signals, it is necessary to go through the two lists to publish the type and name of each signal.

The IIRPvhdl_ProcessStatement class has defined two functions, `_publish_cc_in_sig_decl()` and `_publish_cc_out_sig_decl()`, to publish input signals and output signals respectively. For input signals, a prefix "in_" is put to the name of each signal. For output signals, a prefix "out_" is put to the name of each signal. The reason for adding a prefix is that one signal could be both an input signal and an output signal. In the simulation object model, these two types of signals are implemented differently. As a result, this signal has to be described by one input signal and one output signal. The prefix is used to distinguish the two signals.

To publish the type of the signal, IIRScram.hh has defined a virtual function `void _publish_cc_type_name()`. For each signal declaration, this function will publish the type of the signal using `_cc_out`. In general, SAVANT puts a prefix "Savant" and suffix "Type" to each VHDL type. For example, the VHDL "bit" type will be published as "SavantbitType". Previous sections have discussed how to implement three basic VHDL data types in C++.

Each time an input signal is published, it is necessary to increase the input signal counter **in_sig** by one. For array types, it is necessary to add the total number of elements in the array to the counter. To get the size of the array, the IIRPvhdl_ArraySubtypeDefinition class has defined a function **int _get_array_total_element_num()**. This function simply multiplies each dimension of the array and returns the product. Details on implementing VHDL array types will be discussed later.

SAVANT has defined a virtual function **IIR_Boolean _is_array_type()** in IIRScram.hh. Its function is to determine whether a IIR node is of array type. IIR nodes of array type will return a boolean value true, all other IIR nodes will return false. By calling this function for each signal, it can be determined if this signal is an array type.

The last thing to mention is using single array elements as input signals. In SAVANT, if the value of an array element is referenced as an input signal, the *whole* array is put into the input signal list (not that particular array element). This is not the desired behavior in this project. As a result, the IIRPvhdl_IndexedName class has overloaded the **_get_list_of_input_signals()** function so that *single* array element is put into the input signal list. Details about the IIRPvhdl_IndexedName class will be covered in later sections. The problem here is how to publish name of the signal array element. The index of the array element is used as part of the name for that signal. For example, if array element arr(1,2) is in the input signal list, the name of this signal will be "in_arr_1_2". To determine whether a signal is an array element, the IIR predefined **get_kind()** function can be used. If the return value of this function is "IIR_INDEXED_NAME", the signal is an array element. We can then use the above naming scheme to publish it.

Variable Declarations

SAVANT did not implement a way to get a "variable list". To find out what variables have been declared in a VHDL process, the **process_declarative_part** data element of IIR_ProcessStatement must be used. The type of "process_declarative_part" is "IIR_DeclarationList". It is the list of declaration items in the process. It may contain type declarations and variable declarations. To see if an declaration item is a variable declaration, we can use the predefined "get_kind()" function to get its type. If the return value is "IIR_VariableDeclaration", this item is a variable declaration. We can then publish this declaration item as a variable.

Publishing variable declaration is similar to publishing signal declarations. The only difference is that no prefix is add to variable names. Also, each time after publishing a variable, the counter "state_num" must be increased. The same rule about array size applies for variable arrays.

The Constructor

After publishing signal and variable declarations, the next step is to publish the constructor. Since "BasicObject" is the parent class of each simulation class, it is necessary to call the constructor of "BasicObject". The constructor of BasicObject uses the number of input signal, output signal, and state in the VHDL process as parameters (this is why counters "in_sig", "out_sig" and "state_num" are declared).

The constructor basically has two tasks: to register signals and variables, and to assign initial values to them. The way to publish signal registrations is similar to publishing the signal declarations. The **_proc_in_sig_list** data element is used to publish input signal registration, and the "_proc_out_sig_list" is used to publish output signal registrations. The **process_declarative_part** is used to publish state (variable) registrations.

If there is an input signal **in_a**, an output signal **out_b**, and a variable **c**, then **registerInSignal(&in_a)**, **registerOutSignal(&out_b)** and **registerState(&c)** should be published to register them respectively. These three functions are declared by the "BasicObject" class.

To get the initial value of a signal, the predefined "get_value()" function is called. This function returns an IIR type pointer pointing to the initial value expression. If the returned pointer is not NULL, its **_publish_cc()** function can be called to publish the signal's initial value. Otherwise, the rules discussed in previous sections to set the initial value of the signal. The same approach applies to setting variable initial values.

The "wait for" signals

Two extra signals, **in_wait_for_signal** and **out_wait_for_signal** are declared exclusively to handle the VHDL wait statement. The two signals are of type

SavantbitType. The first signal is an input signal, the second signal is an output signal. Their initial values are both 0. These two signals are not signals declared by the VHDL process. They are extra add on signals for each simulation class. The reason to add them will be cleared up in later sections.

It is important to know the registration of these two signals should be put to the last part of the signal registration section. As a result, they will be put to the end of the input and the output signal list of the simulation object. On the other hand, these two signals are connected directly when generating the interconnection information [9].

The executeProcess() Function

The **executeProcess()** function is a line to line translation of the sequential statements in the VHDL process. To publish all the sequential statements, the _publish_cc() function of **_process_statement_part** is called.

Before calling the **process_statement_part._publish_cc()**, the system-publishing prefix is set to "in_" by calling the **void _set_publish_prefix_string()** function defined in IIRScram.hh. Once the system publishing prefix is set, all signals or variables names will contain this prefix when they are published. In most cases, the sequential statements in a process will reference an input signal, thus the prefix is set to "in_". When publishing the output signals, the system-publishing prefix need to be changed temporarily to "out_". After calling **process_statement_part._publish_cc()**, the system publishing prefix needs to be restored to its original value.

In addition, a static integer "P" is also declared in `executeProcess()`. The purpose of this variable is to handle the VHDL "wait" statement. It will be covered in detail in the section discussing the VHDL wait statement.

The IIRPvhdl Class

The IIRPvhdl class is the root class of the IIRPvhdl layer. The reason to add this class is to declare some global data elements and public functions for the IIRPvhdl layer classes. The following data and functions are declared in the IIRPvhdl class:

- **`dl_list<IIR_Declaration> *_proc_in_sig_list`**, which is a global variable used to keep the input signal list of the current VHDL process being published.

- **`dl_list<IIR_Declaration> *_proc_out_sig_list`** which is a global variable used to keep the output signal list of the current VHDL process being published. The way how this and the above variable is used has been discussed in previous sections.

- **`IIR_Boolean _is_identical_with(IIR *obj)`**, which is a function used to test whether `obj` and the current IIR node (*this) have the same name. The way to determine this is to use **`strstream`** to print out the names of these two objects. If the names are the same, then the function returns "true". Otherwise, the function returns "false".

- **`int _get_position_in_list(dl_list<IIR_Declaration> *list,`** **`IIR *obj)`**, which is a function used to find the position of `obj` in `list`. If `obj` is found in `list`, then its position is returned. Otherwise, "-1" is returned.

This function will call the above `_is_identical_with(IIR *)` function to test if `obj` has the same name of any node in `list`.

- **`void _clean_sig_list_for_identical_elements(dl_list<IIR_Declaration> *list)`**, which is a function used to get rid of redundant array entry element from `list`. In SAVANT, if an array entry appears several times in a VHDL process, there will be multiple IIR_IndexedName objects created for that array entry. Thus after the input or output signal list of the VHDL process is created, it is possible the same array entry could appear more than once in the two lists. Because each array entry object has a different memory address, the only way to tell two arrays elements are the same is to examine their names. This is the reason to define the **`_is_identical_with()`** function. It is also possible that the whole array is in the signal list, and one of its entry element appears in the list. If this happens, this array entry also must be deleted from the signal list. The **`_get_position_in_list()`** function is used to find out the position of the whole array signal(the prefix of the array entry). If the return value is "-1", the whole array is not in the list. Otherwise, the whole array has already been put into the list and the array entry is deleted.

For coding details of the IIRPvhdl class, please refer to [13, IIRPvhdl.cc].

3.2.8 Signal Assignment Statement

The predefined IIR_SignalAssignmentStatement class updates the projected waveform output of one or more signal drivers. It is a child class of the IIR_SequentialStatement class. The signal assignment may appear anywhere a sequential statement may appear. The IIRPvhdl_SignalAssignmentStatement class is defined to overload the **`_publish_cc()`** function of IIRScram_SignalAssignmentStatement class.

Predefined Public Method and Data

The IIR_SignalAssignmentStatement has a predefined target method **`IIR* get_target()`**. Target method refers to the target of a signal assignment statement. After getting the IIR pointer of the target, its **`_publish_cc()`** method can be called to publish its name. The target is an output signal. Since the system publishing prefix is "in_", and all output signals should have prefix "out_", we need to change the system publishing prefix temporarily to "out_". After the **`_publish_cc()`** method of the target is called, the system publishing prefix should be changed back to "in_".

Another useful predefined public method is IIR_DelayMechanism get_delay_mechanism(). A signal assignment statement either uses transport or inertial delay. The the return value of this method should be either "IIR_TRANSPORT_DELAY" or "IIR_INERTIAL_DELAY".

The IIR_SignalAssignmentStatement has a predefined data element **`IIR_WaveformList waveform`**. It is the list of signal drivers (waveforms) associated with this signal assignment. We need to go through this list to publish all

```
 a <= inertial b after 1 ns, c after 2 ns;
```

Figure 3.3: VHDL Source Code

```
out_a = in_b;
assignDelay(&out_a, 1 NS, INERTIAL);

out_a = in_c;
assignDelay(&out_a, 2 NS, INERTIAL);
```

Figure 3.4: Published C++ Code

Figure 3.5: Default Publishing Format of Signal Assignment

waveforms.

Default Publishing Format

The default format of signal assignment is to publish an C++ assignment statement and a **assignDelay()** function for each individual waveform. Figure 3.5 is an example of the publishing format.

To publish the assigned value, the **_publish_cc()** method of the IIR_WaveformElement class should be called.

Both "INERTIAL" and "TRANSPORT" are defined in SavantGlobal.h file. They are published according to the result of the **get_delay_mechanism()** function. If the return value is "IIR_INERTIAL_DELAY", then "INERTIAL" is published. If the return value is "IIR_TRANSPORT_DELAY", then "TRANSPORT" is published.

The delay of the signal assignment can be retrieved by calling the predefined **IIR* get_time()** method of the IIR_WaveformElement class. The returned IIR pointer will point to the assignment time expression. If the pointer is not NULL, then its **_publish_cc()** method is called to publish the time expression. If the pointer is NULL, this means "delta"delay and the SavantGlobal.h defined "DELTA" string should be published.

The IIRPvhdl_WaveformElement Class

The IIR_WaveformElement has a predefined public method **''IIR* get_value()''**. It returns an IIR pointer pointing to the value expression being assigned to the output signal. The IIRPvhdl_WaveformElement class is defined to overload the **_publish_cc()** of the IIRScram_WaveformElement class. There is simply one line in the new **_publish_cc()** function: **get_value()->_publish_cc()**.

Whole Array Assignment

It is valid in VHDL to assign the value of one array to another. For example, if both **a** and **b** are two dimensional array objects, "**a <= b**" means to do one to one copy of the array element from **b** to **a**. In our project, array is not registered to the simulation kernel as single object but as a group of discrete elements, thus whole array assignment should be handled differently. Details on how to handle VHDL arrays will discussed in later sections.

Assignment of Record Field

In contrast to array, record is registered to the simulation kernel as a single object. Thus when a filed of the record object is assigned a new value, the delay should be assigned to the whole record object. To get the name of the record object, the predefined `_get_prefix()` method of IIR_SelectedName class is called. This method will return the pointer of the record object. To publish its name, we can simply call its `_publish_cc()` function.

3.2.9 Variable Assignment Statement

The IIR_VariableAssignmentStatement class updates the value of a variable with the value specified in an expression. It is a child class of the IIR_SequentialStatement class. Variable assignment statement may appear anywhere a sequential statement may appear.

The IIR_VariableAssignmentStatement also has a predefined public target method, the **`''IIR* get_target()''`**. It will return an IIR pointer to the target of the assignment. It defines another public method **`''IIR* get_expression()''`** to get the value of the assignment. To publish the assignment equation, the **`get_target()->_publish_cc()`** is called first, then an "=" is written out to "Classes.h", then **`get_expression()->_publish_cc()`** is called.

Variable Name

When declaring variables, no prefix is added to their names. Now that the system-publishing prefix string is set, variable names will have the system prefix as well. This is not desired in this project. To overcome this problem, a IIRPvhdl_VariableDeclaration class is added. This class overloads the **`_publish_cc()`** of IIRScram_VariableDeclaration class. Actually the new function is almost the same as the old one, it only comments out the line `IIRScram::_publish_cc_prefix_string()`, which publish the system-publishing prefix.

3.2.10 Expressions

In signal assignment and variable assignment statements, expressions are most commonly used as the new value to be assigned. As mentioned above, the **`get_value()`** method of the IIR_WaveformElement class and the **`get_expression()`** of the IIR_VariableAssignmentStatement class will return an IIR pointer to the expression. To publish the expression, the **`_publish_cc()`** method of the IIR pointer is called. Most of the time, the IIR pointer returned will point to an IIR_DyadicOperator node or an IIR_MonadicOperator node.

Dyadic operator is an operator with two operands, like **`''add''`**. Monadic operator is operator with only one operand, like **`''not''`**.

The Dyadic Operator Classes

The predefined IIR_DyadicOperator classes include logical, relational, shift, adding, multiplying and miscellaneous operators. Derivatives of this class represent both language predefined dyadic operators and subprograms defining overloading of these operators.

The parent class of IIR_DyadicOperator is IIR_Expression, which has lots of predefined child classes, such as the IIR_NandOperator, the IIR_EqulityOperator class, etc. Actually all operator classes with two operands are its child classes.

The IIR_DyadicOperator class has two useful predefined methods, the ``IIR* get_left_operand()'' and the ``IIR* get_right_operand()''. The first method returns an IIR pointer to the left operand, the second method returns an IIR pointer to the right operand. By calling the _publish_cc() method, the two operands can be published easily. It is possible that either of the IIR pointers points to an IIR_DyadicOperator node itself. This is the situation where the expression contains more than one dyadic operators. The expression tree is thus more than one level. The whole expression can be published recursively.

The predefined ``get_kind()'' method can be used to determine the name of the operator. For example, if the return value is "IIR_EQULITY_OPERATOR", it means the current operator is an "equality" operator and we can publish the corresponding " == "operator in C++. Figure 3.6 is a code segment of the _publish_cc() method of the IIRPvhdl_DyadicOperator class.

Several operators, such as "and", "or", and "not", can be either logical operators or bitwise operators. To publish the correct operator, their type should be determined. To solve this problem, the IIRScram_DyadicOperator class defines a virtual function ``void _publish_cc_operator_name()''. All its child classes must override this class to print out the correct operator. For example, Figure 3.7 show this function of the IIRPvhdl_AndOperator class.

If the return value of get_subtype()->_is_bit_type() is true, it means the operation is a bitwise operation and the bitwise operator should be published. Otherwise, the logic operator should be published. The way to publish these operations is shown in the default section of Figure 3.6 on page 58. Please refer to [13, IIR-

```
switch(get_kind())
  case IIR_NAND_OPERATOR :
    _cc_out << "~(";
    get_left_operand()->_publish_cc();
    _cc_out << " & ";
    get_right_operand()->_publish_cc();
    _cc_out << ")";
    break;
  ...
  default:  // for and/or/not, both logical and bitwise
    _cc_out << "(";
    get_left_operand()->_publish_cc();
    _cc_out << " ";
    _publish_cc_operator_name();
    _cc_out << " ";
    get_right_operand()->_publish_cc();
    _cc_out << ")";
```

Figure 3.6: Example on How to Publish Expressions

```
void IIRPvhdl_AndOperator::_publish_cc_operator_name() {
  if(get_subtype()->_is_bit_type()){
    _cc_out << "&";
  } else {
    _cc_out << "&&";
  }
}
```

Figure 3.7: The `_publish_cc_operator_name()` Method of IIRPvhdl_AndOperator

Pvhdl_DyadicOperator.cc] for coding detail.

The Monadic Operator Classes

The predefined IIR_Monadic operators include identity, negation, absolute value and not. Derivatives of this class represent both language predefined monadic operators and subprograms defining overloading of these operators.

The IIR_MonadicOperator class has a predefined function ``IIR* get_operand'' which returns an IIR pointer of the operand. The operand can be either a dyadic operator or a monadic operator itself. To publish the operand, its ``_publish_cc()'' method is called. The IIRScram_MonadicOperator class also defined a virtual function ``void _publish_cc_operator_name()''. This function also must be overloaded by its child class. Up to now, only the NOT monadic operator has been implemented. For coding details, please refer to [13, IIRPvhdl_MonadicOperator.cc].

3.2.11 The If Statement

Like C++, VHDL has a ``if..then..else'' statement which evaluates an condtion then executes different branches accordingly. The goal here is to translate the VHDL If statement into the C++ If statement.

If statement usually contains a test condition, a "then" branch, a cluster of "elsif" branches, and a final "else" branch. AIRE defines the IIR_IfStatement class and the IIR_Elsif class to implement the If statement. Correspondingly, an IIRPvhdl_Ifstatement class and an IIRPvhdl_Elsif class have been implemented to perform the translation work.

The IIRPvhdl_IfStatement Class

The AIRE predefined IIR_IfStatement class provides for the optional, selective execution of one or more sequential statement lists. It is a child class of IIR_SequentialStatement and may appear anywhere sequential statements are allowed.

The IIR_IfStatement uses a chain of IIR_Elsif tuples to contain the elsif parts of the If statement. The IIR_Elsif tuple combines a test condition and a sequence of statements, which are to be executed if the test condition is true. If the recursion does not encounter a "true", the final else sequence of statements (the else_sequence in IIR_IfStatement) is reached.

The IIR_IfStatement has the following predefined public data and method:

- **`IIR* get_condition()`**, which returns an IIR pointer to the boolean "condition" which is evaluated in order to determine which sequential statements are to be executed.

- **`IIR_SequentialStatementList then_sequence`** is the "then" statement branch which is to be executed when the condition is true.

- **`IIR_Elsif* get_elsif()`**, which returns an IIR_Elsif pointer pointing to the "elsif" sequences.

- **`IIR_SequentialStatementList else_sequence`**, which is the "else" statement branch.

To publish the If statement, we only need to call the **`_publish_cc()`** method of the above public data or pointers returned by the public methods. Coding is rather straightforward. For details, please refer to [13, IIRPvhdl_IfStatement.cc].

The IIRPvhdl_Elsif Class

The predefined IIR_Elsif class represents one step within a recursive if-then-else statement. It is the "elsif" branch which may contain more than one "elsif" statement sequences.

The IIR_Elsif class has the following predefined public data and methods:

- **`IIR* get_condition()`**, which returns an IIR pointer pointing to the boolean expression to be evaluated.
- **`IIR_SequentialStatementList then_sequence_of_statements`**, which is the sequence of statement to be executed when the condition is true.
- **`IIR_Elsif* get_else_clause()`**, which returns the next "elsif" sequence if it exists. This makes the "elsif" sequence a linked list.

Similarly, the **`_publish_cc()`** method of the above data and returned pointers are called to publish the "elsif" statement. Details of coding please refer to [13, IIRPvhdl_Elsif.cc].

3.2.12 The Case Statement

VHDL has a "Case" statement in which the behavior is to depend on the value of a single expression. Different evaluation of the expression will lead to the execution of different sequential statement sequences. This is similar to the "Switch" statement in

C++. But the VHDL "Case" can not be converted to the C++ "Switch" statement directly. The reason is that in C++, the evaluation values of "Switch" can not be expressions, they can only be constants. Thus the VHDL "Case" statement is also translated into C++ "if..then..else" statement.

The IIRPvhdl_CaseStatement Class

The AIRE predefined IIR_CaseStatement provides for execution of at most one sequential statement list from a set of alternatives. It is a child class of the IIR_SequentialStatement class and may appear anywhere sequential statements are allowed.

The IIR_CaseStatement class has a predefined `''IIR* get_expression()''` public method. The returned IIR pointer of this method points to an expression whose value is evaluated in order to select one choice and the implied sequence of statements to execute.

The IIR_CaseStatement class has a predefined public data named **`case_statement_alternatives`**. It is of type IIR_CaseStatementAlternativeList. It is a list of the alternatives of the "Case"statement.

The `_publish_cc()` function of IIRPvhdl_CaseStatement basically does 4 things:

1. Save the returned IIR pointer of **`get_expression()`** to **`_current_publish_node`**. The **`_current_publish_node`** is an IIR* type global variable defined in IIRScram.hh. The reason to save the pointer to this variable is to be able to publish the expression later when out of the IIR_CaseStatement node.

2. Publish "if(false)" as the "then" branch of the C++ If statement, so that all "Case" alternatives can be published as "elsif" branches.

3. Call the `_publish_cc()` method of the `case_statement_alternatives` to publish the "Case" alternatives.

4. Restore the old `_current_publish_node` value.

For programming details please refer to [13, IIRPvhdl_CaseStatement.cc].

The IIRPvhdl_CaseStatementAlternativeByExpression Class

The predefined IIR_CaseStatementAlternativeByExpression represents a case statement alternative in which the choice is a simple expression, discrete range (range type), or element simple name (the choice). It is a child class of the predefined IIR_CaseStatementAlternative class.

The IIR_CaseStatementAlternativeByExpression has a predefined public method ``IIR* get_choice()'' which returns an IIR pointer pointing to the choice expression of this alternative.

The IIR_CaseStatementAlternative class has a predefined public data ``IIR_SequentialStatementList sequence_of_statements'' which is inherited by the IIR_CaseStatementAlternativeByExpression class. This data element is basically a list of sequential statements which are to be executed when the "Case" expression is evaluated to match the choice expression of this alternative.

The `_publish_cc()` method of IIRPvhdl_CaseStatementAlternativeByExpression publish the "Case" alternative as a "elsif" branch of the If statement. It does the following things:

1. Publish the "else if(" string.

2. Call **_current_publish_node->_publish_cc()** to publish the expression. In the **_publish_cc()** method of IIRPvhdl_CaseStatement, the IIR pointer to the expression is stored in **_current_publish_node**.

3. Publish the "==" string.

4. Call **get_choic()->_publish_cc()** to publish the choice expression.

5. Publish the ")".

6. Call **sequence_of_statements._publish_cc()** to publish the sequential statement of this "Case" alternative.

Please refer to [13, IIRPvhdl_CaseStatementAlternativeByExpression.cc] for coding details.

The IIRPvhdl_CaseStatementAlternativeByOthers Class

The predefined IIR_CaseStatementAlternativeByOthers represents a case statement alternative in which the choice implicitly denotes other elements of the case's composite subtype not previously explicit within an IIR_CaseStatementAlternativeList. It is similar to the "else" branch of the If statement or the "default" branch of the Switch statement in C++.

The **_publish_cc()** method of IIRPvhdl_CaseStatementAlternativeByOthers simply calls **_publish_cc()** of **sequence_of_statements** to publish the code. Since IIR_CaseStatementAlternativeByOthers is a child class of IIR_CaseStatementAlternative class, it inherits this data element too. For coding details, please refer to [13, IIRPvhdl_CaseStatementAlternativeByOthers.cc].

3.2.13 The For Loop Statement

VHDL has a For loop statement which resembles the C++ For loop statement. The predefined IIR_ForLoopStatement executes a sequences of statements zero or more times, advancing the value of an iterator constant once before each execution of the loop body. The IIR_ForLoopStatement class is the child class of IIR_SequentialStatement and may appear anywhere a sequential statement is allowed.

The IIR_ForLoopStatement has the following predefined public data and method:

- **`IIR_ConstantDeclaration* get_iteration_scheme()`**, which returns an pointer pointing to the iteration scheme. The iteration scheme, a constant declaration, is the For loop iterator. The declaration's subtype determines the iteration direction and range.

- **`IIR_SequentialStatementList sequence_of_statements`**, which is the list of sequential statements within the For loop.

The IIRPvhdl_ForLoopStatement class is defined to overload the **`_publish_cc()`** method of the IIRScram_ForLoopStatement class. To publish the For loop, the following functions are used:

- **`get_iteration_scheme()->_is_ascending_range()`**, which is called to determine whether the iteration scheme is ascending. If this is true, its left bound is its lower bound. Otherwise, its right bound is its lower bound. The lower bound is assigned to C++ iteration variable.

- `get_iteration_scheme()->_publish_cc_left()`, which is called to publish the left bound of the iteration scheme.

- `get_iteration_scheme()->_publish_cc_right()`, which is called to publish the right bound of the iteration scheme.

The above functions are called to publish the head of the For loop. Inside the For loop, `sequence_of_statements._publish_cc()` is called to publish the sequential statements of the For loop.

In VHDL, the iteration variable is meaningful only within the For loop. Its value can't be reference outside the For loop. In SAVANT, an iteration variable is named by its memory address. As a result, different iteration variables have different names. In the published C++ class, each iteration variable is declared within the For loop. For coding details, please refer to [13, IIRPvhdl_ForLoopStatement.cc].

3.2.14 The While Loop

VHDL also has a While Loop statement which is similar to the While Loop statement in C++. The predefined IIR_WhileLoopStatement executes a sequential statement list zero or more times. A boolean condition is evaluated before each iteration. If the condition evaluates true, the enclosed statement sequence is executed. Otherwise, the statement following the While loop statement is executed.

The IIR_WhileLoopStatement is a child class of IIR_SequentialStatement class and may appear anywhere a sequential statement is allowed.

The IIR_WhileLoopStatement has the following predefined public data and method:

- **`IIR* get_while_condition()`**, which returns an IIR pointer pointing to the loop condition. The While condition is evaluated at the beginning of each iteration through the loop statement's body. When the While condition evaluates False, the loop execution terminates.

- **`IIR_SequentialStatementList sequence_of_statements`**,which is the list of sequential statement that will be executed when the condition is true.

To publish the While Loop, we simply need to call the **`''_publish_cc()''`** function of the above data and the IIR pointer returned by the method. Coding is straightforward. For coding details please refer to [13, IIRPvhdl_WhileLoopStatement.cc].

3.2.15 The Wait Statement

VHDL has a Wait statement which does not have corresponding language construct in C++. A VHDL process (with no sensitivity list) executes from the beginning of the process to the first occurrence of a Wait statement, then suspends until the condition specified in the Wait statement is satisfied. If the process only includes a single Wait statement, the process reactivates when the condition is satisfied and continues to the "end process" statement, then begins executing again from the beginning. If there

```
wait_statement <=
   [label:] wait [on signal_name {,...}]
                 [until boolean_expression]
                 [for time_expression]
```

Figure 3.8: Syntax of Wait Statement

are multiple Wait statements in the process, the process executes only until the next Wait statement is encountered [1, page168-169].

Syntax

Wait statement is a sequential statement with syntax rule shown in Figure 3.8 on page 68.

The *sensitivity* clause, *condition* clause, and *timeout* clause specify when the process is subsequently to resume execution. They can be combined together to use in the VHDL process.

Starting with the word **on**, the sensitivity clause specifies a list of signals to which the process responds. If the Wait statement contains only a sensitivity list, the process will resume whenever any one of the listed signals has an event. The condition clause starts with the word **until**. It specify a condition that must be true for the process to resume. The timeout clause starts with the word **for**. It specifies a maximum interval of simulation time for which the process should be suspended [2, page114-116].

The IIR_WaitStatement Class

The IIR_WaitStatement suspends execution pending a signal event, boolean condition and/or time out interval. It is a child class of the IIR_SequentialStatement class and may appear almost anywhere a sequential statement may appear (some restrictions in subprograms). It has the following predefined public data and methods:

- **IIR_SignalNameList sensitivity_list**, which is the sensitivity list.

- **IIR* get_condition_clause()**, which returns an IIR pointer to the condition clause. If no condition clause is associated with the Wait statement, the pointer to condition clause returns NIL.

- **IIR***

  get_timeout_clause(), which returns an IIR pointer to the timeout clause. A NIL value for the clause denotes timeout at STD.STANDARD.TIME'HIGH.

The IIRPvhdl_WaitStatement class overloads the **_publish_cc()** method of the IIRScram_WaitStatement class. The three Wait clauses all have been implemented.

The executeProcess() Function

Before discussing details on how to implement the three Wait clauses, it is necessary to go back to the structure of the **executeProcess()** method of the simulation object class.

The way the **executeProcess()** implements the suspension semantics of the Wait statement is simple. It uses a If statement to test the Wait conditions(an condition expression or event of sensitivity list signals). If the test returns true, then execution of the function goes to the next statement. Otherwise, the function simply returns.

The semantics of Wait statement requires that VHDL process resume from the last suspended Wait statement, not the beginning of the process. In C++, each time an executeProcess() function is resumed, it is always resumed from the beginning. Thus it requires a jump from the beginning of the function to the Wait statement where the function was last suspended.

To solve this problem, two things have to be done. First, each Wait statement has to be labeled so that a jump can reach it directly. Second, a record has to be kept when the **executeProcess()** is suspended. The next time the function is resumed, the record will tell the function where to jump to. A local static integer "P" is defined to serve this purpose. Its initial value is "0". Each time a Wait statement is reached, "P" increases by "1". The first Wait statement is labeled "BLOCK1", the second is labeled "BLOCK2", and so forth. A switch statement is publish at the beginning of each **executeProcess()** function using the value of "P" as branching factor. Thus when a **executeProcess()** is suspended, "P" keeps the sequence number of that Wait statement. Since "P" is a static variable, its value is not lost when the function returns. When the function is resumed next, the switch statement will jump to the Wait label according to the sequence number kept by "P". This is how things work.

Another issue is how each Wait statement could know its sequence number in the process. Wait statement will be published in the IIR_WaitStatement node, not the IIR_ProcessStatement node. Thus each IIR_WaitStatement node should already know its sequence number in the VHDL process when it is published. This problem is solved by SAVANT. The IIRScram_WaitStatement class has defined an public data **IIR_Int32 wait_id**. This integer is used to keep the Wait sequence number. Remember in the **_publish_cc()** method of IIRPvhdl_ProcessStatement, the Wait list is built before any publishing work. It is during this building of the Wait list that each IIRScram_WaitStatement node is assigned an wait_id. For details please refer to [13, IIRPvhdl_ProcessStatement.cc].

The Wait On Sensitivity List Clause

The BasicObject class defines a function `bool hasEvent(*)` to test if a signal has an event on it. To implement the sensitivity list semantics, this function is called upon each signal in the sensitivity list of the Wait sensitivity clause. If all **`hasEvent()`** function returns false, this means no event at all and the **`executeProcess()`** function should return. Otherwise, the function should start executing the following statement. The names of signals in the sensitivity list can be easily retrieved from the predefined **`sensitivity_list`** data element.

The Wait Until Condition Clause

The Wait condition clause is easy to handle. The **`get_condition_clause()`** will return the IIR pointer to the condition clause. The condition is published as the test expression in an C++ If statement. If the condition is false, the **`executeProcess()`** function should return. Otherwise, the function should proceed from the next statement.

The Wait For Timeout Clause

As mentioned earlier, each C++ class has defined two signals exclusively to handle the Wait For clause, the **`in_wait_for_signal`** and the **`out_wait_for_signal`** of type **`SavantbitType`**. The first signal is registered as an input signal, the second is regis-

```
        P++;
 BLOCK3:
        out_wait_for_signal = ! in_wait_for_signal;
        assignDelay(&out_wait_for_signal,3 NS,TRANSPORT);

        if(!hasEvent(&in_wait_for_signal)) return;
```

Figure 3.9: Example of Wait For Timeout Clause

tered as an output signal. These two signals are connected directly when generating the interconnection information.

The time out clause is actually handled as a signal assignment statement with transport delay. The delay time equals the timeout value. For example, if the third Wait statement in a VHDL process is "wait for 3 ns", then the published C++ code is shown in Figure 3.9 on page 72.

Thus the Wait For timeout clause is basically translated into the Wait On sensitivity clause.

Combination of Clauses

All the Wait clauses are implemented using the C++ If statement. The combination of Wait clauses is only a matter of publishing which If statement first. Since the Wait For clause is transformed into the Wait On sensitivity clause, the problem is simplified to publishing the combination of sensitivity and condition clauses.

If a Wait statement includes a sensitivity clause as well as a condition clause, the condition is only tested when an event occurs on any of the signals in the sensitivity clause [2, page 116]. This means the sensitivity clause has higher priority than the condition clause. The If condtion generated by the sensitivity clause should be tested first. If there is an event on the sensitivity list, then the If condition generated by the Wait condition clause should be tested. For coding details, please refer to [13, IIRPvhdl_WaitStatement.cc].

Process with Sensitivity List

SAVANT transforms process with a sensitivity list into process with a Wait statement which uses the same sensitivity list. This Wait statement is put as the last statement in the VHDL process. Thus there is no need to spend extra effort on this issue. To test this, simple type ``scram -publish-vhdl test.vhd'' to see the VHDL code generated by SAVANT (test.vhd is the testing VHDL file containing process with sensitivity list statement).

Appendix E shows the code of the IIRPvhdl_WaitStatement.hh and appendix F shows the IIRPvhdl_WaitStatement.cc file.

3.2.16 Constant Declaration

In VHDL, constants can be declared anywhere declarations can appear. Due to limited time, the declarations of constants are restricted only to package declaration in this project. It is also possible to implement the declaration of constant in other declaration bodies.

Since C++ also allows constant declarations, the translation of VHDL constant declaration into C++ constant declaration is straightforward. After being declared, constants can be used just like signals and variables.

```
VHDL Constant Declaration <=
   constant identifier{,...}: subtype_indication [:=expression];

C++ Constant Declaration <=
   const type identifier = expression;
```

Figure 3.10: Constant Declaration Syntax

The syntax of VHDL constant declaration and the C++ constant declaration syntax is shown in Figure 3.10 on page 74.

The IIR_PackageDeclaration Class

Package declarations are usually at the beginning of VHDL source code, thus packages will be published first by the SAVANT publisher. As a result, declarations in the package will be published at the beginning part of the "Classes.h" file. Since all C++ classes are published to "Classes.h", these constants will become global constants. They can be accessed by any classes in the Classes.h file.

The predefined IIR_PackageDeclaration class represents collections of declarations which are elaborated at most once, as a collection. This class has a predefined public data element IIR_DeclarationList package_declarative_part. It is a list of declarations appearing in the package.

When processing the package declaration, the IIRScram_PackageDeclaration calls the **_publish_cc()** method to publish the C++ code. The **_publish_cc()** method in turn calls the locally defined **_publish_cc_header()** function. This function then calls the **_publish_cc_package_declarations()** method of the **package_declarative_part**, which is of type IIR_DeclarationList. This sequence

```
case IIR_CONSTANT_DECLARATION:
  _cc_out.set_file("Classes.h");
  _cc_out << "const ";
  decl->_get_subtype()->_publish_cc();
  _cc_out << " ";
  decl->_publish_cc();
  _cc_out << " = ";
  decl->get_value()->_publish_cc();
  _cc_out << ";\n" << endl;
  break;
```

Figure 3.11: Code of Constant Declaration

is correct thus there is no need to change it. There is no IIRPvhdl layer class, namely IIRPvhdl_PackageDeclaration, added to shadow the `_publish_cc()` method of the IIRScram_PackageDeclaration class.

The IIRPvhdl_DeclarationList Class

The purpose to add the IIRPvhdl_DeclarationList class is simply to overload the

`_publish_cc_package_declaration()`

function declared by the IIRScram_DeclarationList class. The new function is basically a copy of the old function, only with the changes shown in Figure 3.11 on page 75.

Coding is quite straightforward. For details, please refer to [13, IIRPvhdl_DeclarationList.cc].

3.2.17 Enumeration Types

Like C++, VHDL also supports enumeration types. Thus the translation from the VHDL enumeration type declaration to C++ enumeration type is straightforward. Enumeration types can be declared anywhere, but in this project, only enumeration types declared in package are implemented.

When processing a package declaration, the IIRScram_PackageDeclaration calls the **_publish_cc()** method to publish the C++ code. The **_publish_cc()** method in turn calls the locally defined **_publish_cc_header()** function. This function then calls the **_publish_cc_package_declarations()** method of the **package_declarative_part**, which is of type IIR_DeclarationList. A IIRPvhdl_DeclarationList class has been added to overload the function. For enumeration declaration types, the "get_kind()" method will return "IIR_TYPE_DECLARATION". The "_publish_cc_decl()" method is then called to publish the declarations (coding details please refer to [13, IIR-Pvhdl_DeclarationList.cc]).

The **void _publish_cc_decl()** function is defined at IIRScram.hh as a virtual function. An IIRPvhdl_TypeDeclaration class is defined to overload this function. There is only one line in the new function, **''get_type()->_publish_cc_decl()''**.

The **get_type()** method is a predefined public method of the IIR_TypeDeclaration class. It returns an IIR_TypeDefinition pointer to the new type definition node. In the context of enumeration type declaration, the returning pointer will be an IIR_EnumerationTypeDefinition pointer.

An IIRPvhdl_EnumerationTypeDefinition class is defined to overload the `_publish_cc_decl()` method. In the AIRE standard, the predefined IIR_EnumerationTypeDefinition represents its value domain by a set of enumeration literals. It has a predefined public data **`IIR_EnumerationLiteralList enumeration_literals`**, which is the list of enumeration literals associated with the type definition.

The `_publish_cc_decl()` function of IIRPvhdl_EnumerationTypeDefinition does the following things:

- Set the output file name to "Classes.h".
- Call `_publish_cc_type_name()` to publish the name of the new type.
- Use a for loop to go through `enumeration_literals` and publish each enumeration literal in the list.

After an enumeration type has been declared, there is no difference in using a variable of this enumeration type and variables of other types. Thus declaration is the only thing needed to considered for enumeration types. For programming details, please refer to [13, IIRPvhdl_TypeDeclaration.cc, IIRPvhdl_EnumerationTypeDefinition.cc].

3.2.18 Array Types

VHDL supports array types. Unlike C++, array type have to be defined first. This new type can then be used to declare array objects. This section discusses issues on how to publish VHDL array into C++ arrays.

The IIRPvhdl_IndexedName Class

The predefined IIR_IndexedName denotes a single element of an array. It has a predefined public method `IIR* get_suffix()` which returns an IIR pointer to the name's suffix (an expression which evaluates to a single integer).

The IIRPvhdl_IndexedName Class is defined to handle array entry related problems. This class has overloaded or defined the following functions:

- **`void _get_signal_source_info(set<IIR_Declaration> *siginfo)`**
  This function is defined at IIRScram.hh as a virtual function. The purpose of this function is to put the current array entry into the output signal list **`siginfo`**. In SAVANT, the whole array will be put into the output signal list, not the single element. Thus this function is overloaded to put the array entry to the list.

- **`void _get_list_of_input_signals(set<IIR_Declaration>* list)`** This function is also defined in IIRScram.hh as a virtual function. Its purpose is to put the current array entry into the input signal list denoted by **`list`**. In SAVANT, the whole array is put into the input signal list, not the single entry. Thus this function is overloaded to put the array entry into the list.

- **`void _publish_cc()`** This function is overloaded to publish the whole name of the array entry. It will use the two variables **`_proc_in_sig_list`** and **`_proc_out_sig_list`** defined by IIRPvhdl.hh to determine whether the current array entry is an output signal or an input signal, then put prefix **`''in_''`** or **`''out_''`** accordingly.

- `void _publish_cc_array_entry_location()` This function is defined to publish the suffix of the array entry using the underscore format. For example, array entry `(1,2)` will be published as _1_2. This function is to handle the naming of an array entry where the whole array is not in either the input or the output signal list. Thus this entry is declared as a separate signal and its name will use the underscore format. This function uses the **`get_suffix()`** to get the IIR pointer to the suffix expression.

Please refer to [13, IIRPvhdl_IndexedName.cc] for programming details.

Template Array Classes

VHDL supports direct array operations. That is, array objects can be assigned, added, or multiplied as simple type objects. The result is that each array entry element will perform the operation. This feature is not supported in C++ directly. To implement this feature, VHDL array types have to be declared as C++ classes. These array classes must use operator overloading to implement the whole array operations.

Template array classes have been developed to solve this problem. Right now, template array classes have been developed for one, two, and three dimensional array types. The reason to use template is that all array classes are almost identical except for the data types of their entries. Thus there is no need to generate a separate class for each array type. Figure 3.12 on page 80 show how the assignment operator is overloaded by the one dimensional template array class. For now, the following operators have been overloaded: `+,-,*,/, ==,!=,&,|,¬=`. The "=(int)" operation

```
// overload =
Pvhdl1DArray<Dtype>& operator=(Pvhdl1DArray<Dtype> &obj) {
  for( int i=0; i<size; i++)
    array[i] = obj.array[i];
  return *this;
}
```

Figure 3.12: Overloading Assignment Operator for One Dimensional Array Class

is overloaded for each array class to ensure the initialization of the class object using a single integer value. The code of template array classes implemented in file "PvhdlArray.h". Appendix G shows the portion of one dimensional array template class.

Right now, SAVANT only supports direct array assignment operation. Other array operations are not supported. There is a compiling error if the source VHDL code contains other array operations. These array operations are overloaded for future versions of SAVANT, which is supposed support them.

The IIRPvhdl_ArraySubtypeDefinition Class

The IIRPvhdl_ArraySubtypeDefintion class is added to handle the declaration of array types. Since array classes are implemented as template classes, it is necessary to know two things about the VHDL array type: its dimension and the data type of its entry. The `void _publish_cc()` function is overloaded to publish array types using the array template classes described above.

To get the dimension of the array type, the **`IIR_Int32 _get_num_indexes()`** function is called. This function is defined by the IIRScram_ArrayTypeDefinition class, which is the parent class of IIRScram_ArraySubtypeDefinition class. If the return value is "1", then the "Pvhdl1DArray" template class is used; if the return value is "2", the "Pvhdl2DArray" template class is used; if the return value is "3", the "Pvhdl3DArray" template class is used. No higher dimension array types are supported right now.

To get the data type of the array entry, a for loop is used to call the **_get_element_subtype()** function as many times as the dimension. The final subtype will be the type of the entry element. By calling its **_publish_cc_type_name()** function, the array entry data type can be published. To see how this is done, please refer to [13, IIRPvhdl_ArraySubtypeDefinition.cc].

3.2.19 Record Types

VHDL also supports record types. VHDL record types are translated into C++ record types in a straightforward way.

The IIRPvhdl_RecordTypeDefinition Class

The predefined IIR_RecordTypeDefinition class represents a record type having zero or more element declarations. It has a predefined public data element **element_declarations** of type "IIR_ElementDeclarationList". This is the list of all the fields of this record.

The IIRPvhdl_RecordTypeDefinition class is defined to overload the **_publish_cc_decl()** function. In this function, **element_declarations** is used several times. By going through this list, each of the record is published by its type and name. Also, some operators are overloaded for the record, such as addition, etc. The reason is to support array of record. Since the template array classes have overloaded some operators, thus each defined record type has to overload the same operators. For coding details, please refer to [13, IIRPvhdl_RecordTypeDefinition.cc].

Chapter 4

Conclusions

4.1 Summary

This thesis focuses on the Front End Interface part of the Parallel VHDL Simulation research project conducted by the Computer Science Department of Michigan State University. The major purpose of the Front End Interface is to translate VHDL descriptions into C++ code according to the simulation object model. The Simulation Kernel will then compile and link with the C++ code to simulate the VHDL system.

The SAVANT software package is used as the basis for the Front End Interface. SAVANT has implemented a VHDL analyzer to parse the VHDL source code and generate a parse tree using the Intermediate Forms defined by the AIRE standard. SAVANT has also implemented a C++ Publisher which uses the parse tree to generate C++ code for the TyVIS VHDL simulation kernel. TyVIS is not compatible with the simulation kernel of this project. As a result, the C++ code can not be used. The SAVANT C++ Publisher was then modified to generate C++ for this project.

SAVANT implements the AIRE standard using a layered approach. The C++ Publisher is implemented at the IIRScram layer as a virtual function `_publish_cc()`. To modify the publisher, an IIRPvhdl layer is added to the SAVANT class hierarchy. The `_publish_cc()` function is overloaded in this layer so that the `_publish_cc()` of the IIRScram layer is shadowed. When the publisher is called again, the `_publish_cc()` function in the IIRPvhdl layer is called to generate C++ code for our own simulation kernel.

A subset of critical VHDL constructs have been selected and implemented by modifying the SAVANT Publisher. Many tests have been conducted on these constructs to ensure the their semantic correctness. Up to now, 47 IIRPvhdl layer classes have been added to the SAVANT class hierarchy and bout 6000 lines of C++ code have been developed.

In this project, simulation objects are separated from the simulation kernel, which handles different parallel simulation protocols. This ensures the extensibility and portability of simulations. On the other hand, using Intermediate Forms (AIRE) enables the translation between languages with no direct mapping (VHDL and C++). Also, adding an extra layer (IIRPvhdl) into the SAVANT class hierarchy modifies the behavior of the SAVANT Publisher without changing the SAVANT source code. This makes upgrading to new version of SAVANT much easier.

A big drawback of this approach is that each time a change is made to the source code, the program has to be recompiled and linked again to generate the new executable file. Because of the size of the executable file is quite large (35MB), this process usually takes a very long time (tens of minutes). This greatly reduces the efficiency of user modifications.

SAVANT is also an on-going project. It still does not support some VHDL features, such as bus signals. This software package also contains some bugs.

4.2 Future Work

To handle more complicated VHDL descriptions, more VHDL constructs need to be supported in the future, such as functions and procedures, bus resolution, generic constant, etc. On the other hand, as time goes by, SAVANT will be improved and it can be used to serve this project better.

APPENDICES

Appendix A

IIRPvhdl Layer Class List

```
IIRPvhdl
IIRPvhdl_AdditionOperator
IIRPvhdl_AndOperator
IIRPvhdl_ArchitectureDeclaration
IIRPvhdl_ArraySubtypeDefinition
IIRPvhdl_CaseStatement
IIRPvhdl_CaseStatementAlternative
IIRPvhdl_CaseStatementAlternativeByExpression
IIRPvhdl_CaseStatementAlternativeByOthers
IIRPvhdl_CaseStatementAlternativeList
IIRPvhdl_Choice
IIRPvhdl_Declaration
IIRPvhdl_DeclarationList
IIRPvhdl_DesignFile
IIRPvhdl_DivisionOperator
IIRPvhdl_DyadicOperator
IIRPvhdl_Elsif
IIRPvhdl_EntityDeclaration
IIRPvhdl_EnumerationLiteral
IIRPvhdl_EnumerationTypeDefinition
IIRPvhdl_EqualityOperator
IIRPvhdl_FloatingPointLiteral
IIRPvhdl_ForLoopStatement
IIRPvhdl_IfStatement
IIRPvhdl_IndexedName
IIRPvhdl_IntegerLiteral
IIRPvhdl_MonadicOperator
IIRPvhdl_MultiplicationOperator
IIRPvhdl_Name
IIRPvhdl_NotOperator
IIRPvhdl_OrOperator
IIRPvhdl_PhysicalLiteral
```

```
IIRPvhdl_ProcessStatement
IIRPvhdl_RecordTypeDefinition
IIRPvhdl_ScalarTypeDefinition
IIRPvhdl_SelectedName
IIRPvhdl_SequentialStatement
IIRPvhdl_SequentialStatementList
IIRPvhdl_SignalAssignmentStatement
IIRPvhdl_SubtractionOperator
IIRPvhdl_TypeDeclaration
IIRPvhdl_VariableAssignmentStatement
IIRPvhdl_VariableDeclaration
IIRPvhdl_WaitStatement
IIRPvhdl_WaveformElement
IIRPvhdl_WhileLoopStatement
IIRPvhdl_XorOperator
```

Appendix B

VHDL Source of an AND Gate

```
entity and2 is
    port(A, B: in bit;
         Y: out bit);
end entity and2;

architecture behav of and2 is
begin
gate: process
   variable N : bit := '0';
  begin
    Y <=  N;
    wait on A, B;
    N :=  A and B;
  end process gate;
end architecture behav;
```

Appendix C

C++ Code of the AND Gate

```
#ifndef CLASSES_PVHDL_H
#define CLASSES_PVHDL_H

#include "BasicObject.h"
#include "SavantGlobals.h"

typedef int SavantbitType;
typedef int SavantintegerType;
typedef int SavanttimeType;
typedef double SavantrealType;
typedef Pvhdl1DArray<SavantbitType> Savantbit_vectorType;

class work_Dand2_Dbehav_of_work_Dand2_class0 : public BasicObject
{
 public:
    // input signals
    SavantbitType in_work_Dand2_0a;
    SavantbitType in_work_Dand2_0b;

    // output signals
    SavantbitType out_work_Dand2_0y;

    // signals used to handle wait for statement only
    SavantbitType in_wait_for_signal;
    SavantbitType out_wait_for_signal;

    // local vairables - states
    SavantbitType gatework_Dand2_Dbehav_0n;

 public:

    work_Dand2_Dbehav_of_work_Dand2_class0()  : BasicObject()
    {
        // register input signals
        registerInSignal(&in_work_Dand2_0a,sizeof(in_work_Dand2_0a));
        registerInSignal(&in_work_Dand2_0b,sizeof(in_work_Dand2_0b));

        // register output signals
        registerOutSignal(&out_work_Dand2_0y,sizeof(out_work_Dand2_0y));

        // register the wait for signals
        registerInSignal(&in_wait_for_signal,sizeof(in_wait_for_signal));
        registerOutSignal(&out_wait_for_signal,sizeof(out_wait_for_signal));

        // register states
        registerState(&gatework_Dand2_Dbehav_On, \
                      sizeof(gatework_Dand2_Dbehav_On));

        // input signal initial values
        in_work_Dand2_0a = X;
        in_work_Dand2_0b = X;

        // output signal initial values
        out_work_Dand2_0y = Y;

        // wait for singal initial values
        in_wait_for_signal = 0;
        out_wait_for_signal = 0;

        // state initial values
        gatework_Dand2_Dbehav_On = 0;
    } // end of constructor

    void executeProcess()
    {
        static int P=0;

        // resume to the last wait statement
        switch(P) {
        case 0 : goto BLOCK0;
        case 1 : goto BLOCK1;
        }

BLOCK0:
        // line 11 "and.tex"
        out_work_Dand2_0y = gatework_Dand2_Dbehav_On;
        assignDelay(&out_work_Dand2_0y,DELTA,TRANSPORT);
        // line 12 "and.tex"
        P++;
BLOCK1:
        if( !hasEvent(&in_work_Dand2_0a) &&
            !hasEvent(&in_work_Dand2_0b) ) return;

        // line 13 "and.tex"
        gatework_Dand2_Dbehav_On = (in_work_Dand2_0a & in_work_Dand2_0b);

        // reset P value then return, or resume from beginning
        P = 0;
    }
};

#endif
```

Appendix D

SavantGlobals.h

```
#ifndef SAVANT_GLOBALS_HH
#define SAVANT_GLOBALS_HH

#define X -1
#define Y -1

#define DELTA 1
#define NS *100
#define US *100000
#define MS *100000000

#define TRANSPORT 0
#define INERTIAL 1

#include "PvhdlArray.h"

#endif
```

Appendix E

IIRPvhdl_WaitStatement.hh

```
#ifndef IIRPVHDL_WAITSTATEMENT_HH
#define IIRPVHDL_WAITSTATEMENT_HH

#include "IIRScram_WaitStatement.hh"

class IIRPvhdl_WaitStatement : public IIRScram_WaitStatement {
 public:
  void _publish_cc();
 private:
  void _publish_cc_wait_on();

 protected:
  IIRPvhdl_WaitStatement(){};
  ~IIRPvhdl_WaitStatement(){};
};
#endif
```

Appendix F

IIRPvhdl_WaitStatement.cc

```
#include "IIRPvhdl_WaitStatement.hh"
#include "IIR_Designator.hh"
#include "IIR_Declaration.hh"
#include "IIR_DesignatorExplicit.hh"

void
IIRPvhdl_WaitStatement::_publish_cc(){
  IIR_Designator *desig;
  IIR *cond_clause = get_condition_clause();
  IIR *time_clause = get_timeout_clause();

  _cc_out << "\t" << "P++;\n";
  _cc_out << "BLOCK" << wait_id +1 << ":\n";

  if(time_clause != NULL){
    _cc_out << "\t" << "out_wait_for_signal = ! in_wait_for_signal;\n";
    _cc_out << "\t" << "assignDelay(&out_wait_for_signal,";
    time_clause->_publish_cc();
    _cc_out << ",TRANSPORT);\n" << endl;
  }

  desig = sensitivity_list.first();

  if(desig != NULL) _publish_cc_wait_on();
  else if(cond_clause != NULL) {
    _cc_out << "\t" << "if(!(";
    cond_clause->_publish_cc();
    _cc_out << ")";
    if(time_clause != NULL) {
      _cc_out << " && !hasEvent(&in_wait_for_signal)";
    }
    _cc_out << ") return;\n\n";
  } else if(time_clause != NULL) {
    _cc_out << "\t" << "if(!hasEvent(&in_wait_for_signal)) return;\n\n";
  } else {
    _cc_out << "\t" << "return;\n\n";
  }
}

void
IIRPvhdl_WaitStatement::_publish_cc_wait_on(){
  IIR_Designator *desig;
  IIR_Declaration *sens_sig;

  IIR *cond_clause = get_condition_clause();
  IIR *time_clause = get_timeout_clause();

  desig = sensitivity_list.first();

  _cc_out << "\t" << "if( ";

  while(desig != NULL) {
    ASSERT(desig->get_kind()==IIR_DESIGNATOR_EXPLICIT);
    sens_sig = (IIR_Declaration *) \
               ((IIR_DesignatorExplicit *)desig)->get_name();

    if(sens_sig->_is_array_type()){
      _cc_out << "!";
      sens_sig->_publish_cc();
      _cc_out << ".arrayHasEvent() ";
    } else {
      _cc_out << "!hasEvent(&";
      sens_sig->_publish_cc();
      _cc_out << ") ";
    }

    // move to the next
    desig = sensitivity_list.successor(desig);
    if(desig !=NULL) {
      _cc_out << "&&" << "\n" << "\t" << "    ";
    }
  } // while

  if(time_clause != NULL){
    _cc_out << "&&" << "\n" << "\t" << "    ";
    _cc_out << "!hasEvent(&in_wait_for_signal)";
  }
  _cc_out << ") return;\n";

  if( cond_clause != NULL) {
    _cc_out << "\t" << "else if(!(";
    cond_clause->_publish_cc();
    _cc_out << ")";

    if(time_clause != NULL) {
      _cc_out << " && !hasEvent(&in_wait_for_signal)";
    }

    _cc_out << ") return;\n";
  }

  _cc_out << "\n";
}
```

Appendix G

PvhdlArray.h

```
#ifndef PVHDL_ARRAY_H
#define PVHDL_ARRAY_H

#include <stdio.h>
#include <iostream.h>
#include "BasicObject.h"

template <class Dtype> class Pvhdl1DArray {
 public:
  int size;
  Dtype *array;
  BasicObject *owner;

  // constructor
  Pvhdl1DArray(){
    size = 0;
    array = NULL;
    owner = NULL;
  }

  // distructor
  ~Pvhdl1DArray(){
    if (array != NULL) delete [] array;
  }

  // allocate array
  void allocateArray(int s){
    if(s<=0) return;

    size = s;
    array = new Dtype [s];
  }
```

```
// set the owner BasicObject of this array object
void setBasicObject(BasicObject *b){
  owner = b;
}

// register array as input signal
void registerInSignalArray(){
  for(int i=0;i<size;i++){
    owner->registerInSignal(&array[i],sizeof(array[i]));
  }
}

// register array as output signals
void registerOutSignalArray(){
  for(int i=0;i<size;i++){
    owner->registerOutSignal(&array[i],sizeof(array[i]));
  }
}

// register array as states
void registerStateArray(){
  for(int i=0;i<size;i++){
    owner->registerState(&array[i],sizeof(array[i]));
  }
}

// test if array values have changed
int arrayHasEvent(){
  for(int i=0;i<size;i++){
    if(owner->hasEvent(&array[i])) return 1;
  }

  return 0;
}

// copy array value from obj with delays
void assignArray(Pvhdl1DArray<Dtype> &obj, int delay, int delay_type){
  for(int i=0;i<size;i++){
    array[i] = obj.array[i];
    owner->assignDelay(&array[i],delay,delay_type);
  }
}

// overload ==
int operator == (Pvhdl1DArray<Dtype> &obj) {
  for(int i=0;i<size;i++)
     if(array[i] != obj.array[i]) return 0;
  return 1;
}

// overload !=
int operator != (Pvhdl1DArray<Dtype> &obj) {
  if (*this == obj) return 0;
  else return 1;
}

// overload == (int)
int operator == (int val) {
  for(int i=0;i<size;i++)
     if(array[i] != val) return 0;
  return 1;
}

// overload != (int)
int operator != (int val) {
  if (*this == val) return 0;
  else return 1;
}

// overload =
Pvhdl1DArray<Dtype>& operator=(Pvhdl1DArray<Dtype> &obj) {
  for( int i=0; i<size; i++)
    array[i] = obj.array[i];
  return *this;
}

//////////////// testing
// overload = (int) value, used to initialize the array
Pvhdl1DArray<Dtype>& operator=(int value) {
  for( int i=0; i<size; i++)
    array[i] = value;
  return *this;
}

// overload +
Pvhdl1DArray<Dtype>& operator+(Pvhdl1DArray<Dtype> &obj) {
  Pvhdl1DArray<Dtype> *temp = new Pvhdl1DArray<Dtype>;
  temp->allocateArray(obj.size);
  for(int i=0; i<size; i++)
    temp->array[i] = array[i]+obj.array[i];
  return *temp;
}

// overload -
Pvhdl1DArray<Dtype>& operator-(Pvhdl1DArray<Dtype> &obj) {
  Pvhdl1DArray<Dtype> *temp = new Pvhdl1DArray<Dtype>;
  temp->allocateArray(obj.size);

  for(int i=0; i<size; i++)
    temp->array[i] = array[i]-obj.array[i];
  return *temp;
}

// overload *
Pvhdl1DArray<Dtype>& operator*(Pvhdl1DArray<Dtype> &obj) {
  Pvhdl1DArray<Dtype> *temp = new Pvhdl1DArray<Dtype>;
  temp->allocateArray(obj.size);

  for(int i=0; i<size; i++)
    temp->array[i] = array[i]*obj.array[i];
  return *temp;
}

// overload /
Pvhdl1DArray<Dtype>& operator/(Pvhdl1DArray<Dtype> &obj) {
  Pvhdl1DArray<Dtype> *temp = new Pvhdl1DArray<Dtype>;
  temp->allocateArray(obj.size);

  for(int i=0; i<size; i++)
    temp->array[i] = array[i]/obj.array[i];
  return *temp;
}

// overload &
Pvhdl1DArray<Dtype>& operator&(Pvhdl1DArray<Dtype> &obj) {
  Pvhdl1DArray<Dtype> *temp = new Pvhdl1DArray<Dtype>;
  temp->allocateArray(obj.size);

  for(int i=0; i<size; i++) {
    if(array[i]!=0 && obj.array[i]!=0) temp->array[i] = 1;
    else temp->array[i] = 0;
  }
```

```
    return *temp;
  }

  // overload |
  Pvhdl1DArray<Dtype>& operator|(Pvhdl1DArray<Dtype> &obj) {
    Pvhdl1DArray<Dtype> *temp = new Pvhdl1DArray<Dtype>;
    temp->allocateArray(obj.size);

    for(int i=0; i<size; i++) {
      if(array[i]!=0 || obj.array[i]!=0) temp->array[i] = 1;
      else temp->array[i] = 0;
    }

    return *temp;
  }

  // overload ^
  Pvhdl1DArray<Dtype>& operator^(Pvhdl1DArray<Dtype> &obj) {
    Pvhdl1DArray<Dtype> *temp = new Pvhdl1DArray<Dtype>;
    temp->allocateArray(obj.size);

    for(int i=0; i<size; i++) {
      if(array[i] != obj.array[i]) temp->array[i] = 1;
      else temp->array[i] = 0;
    }

    return *temp;
  }

  // overload ~
  Pvhdl1DArray<Dtype>& operator~() {
    Pvhdl1DArray<Dtype> *temp = new Pvhdl1DArray<Dtype>;
    temp->allocateArray(size);

    for(int i=0; i<size; i++) {
      if(array[i] == 0) temp->array[i] = 1;
      else temp->array[i] = 0;
    }

    return *temp;
  }
}; // class Pvhdl1DArray

#endif
```

Bibliography

[1] David Perllerin and Douglas Taylor, “VHDL Made Easy”, Printice Hall, Upper Saddle River, 1997.

[2] Peter J. Ashenden, “The Designer’ Guide to VHDL”, Morgan Kaufmann, San Francisco, 1996.

[3] James P. Cohoon and Jack W. Davidson, "C++ Program Design - An Introduction to Programming and Object-Oriented Design”, Times Mirror Higher Education Group, 1997.

[4] Moon J. Chung, Department of Computer Science, Michigan State University. Parallel VHDL Performance Simulation Cost and Schedule Report. [Online] Available http://chung-res1.cps.msu.edu/pvhdl/sep.htm, Oct. 13, 1997

[5] Dept. of ECECS, University of Cincinnati.SAVANT: An Extensible Intermediate for VHDL. [Online] Available http://www.ececs.uc.edu/ paw/savant/index.html, July 11, 1998.

[6] Department of ECECS, University of Cincinnati. TyVIS: A VHDL Simulation Kernel. [Online] Available http://www.ececs.uc.edu/ paw/tyvis/index.html, May 25, 1998.

[7] Department of ECECS, University of Cincinnati. AIRE Home Page - Advanced Intermediate Representation with Extensibility (AIRE). [Online] Available http://www.ececs.uc.edu/ paw/aire/Index.html.

[8] Dept. of ECECS, University of Cincinnati. SAVANT Programmer's Manual. [Online] Available http://www.ececs.uc.edu/ paw/savant/doc/programers.html, Aug. 25, 1997

[9] Khashayar Rohanimanesh, "Generating Map File by SAVANT", Technical Document of Parallel VHDL Simulation Project, Department of Computer Science, Michigan State University, July, 1998.

[10] Jinsheng Xu, Department of Computer Science, Michigan State University. An Object Oriented Model for Developing Event-Driven Systems. [Online] Available ftp://chung-res1.cps.msu.edu/pvhdl/paper/OOMODEL.doc, April, 1998.

[11] Jinsheng Xu, Department of Computer Science, Michigan State University, Parallel VHDL Simulation Result Report. [Online] Available ftp://chung-res1.cps.msu.edu/o2k_sp1.xls, March, 1997.

[12] Department of ECECS, University of Cincinnati. IIRScram Layer Source Code. [Online] Available telnet://chung-pvhdl.cps.msu.edu, /home/savant/savant/src/aire/iir/IIRScram, June, 1998.

[13] Mai Yang, Department of Computer Science, Michignan State University. IIR-Pvhdl Layer Source Code. [Online] Available telnet://chung-pvhdl.cps.msu.edu, /home/savant/savant/src/aire/iir/IIRPvhdl, July, 1998.