

AFRL-IF-WP-TR-1999-1507

**PARALLEL VERY HIGH SPEED INTEGRATED
CIRCUITS (VHSIC) HARDWARE DESCRIPTION
LANGUAGE (VHDL) SIMULATION FOR
PERFORMANCE MODELING**

DR. MOON JUNG CHUNG

**MICHIGAN STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
EAST LANSING, MI 48824**

MARCH 1999

FINAL REPORT FOR 07/01/1996 – 02/28/1999

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

**INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE OH 45433-7334**



HPCMP

**High Performance Computing
Modernization Program**

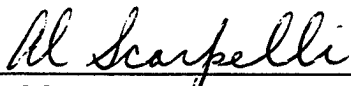
20000127 030

NOTICE

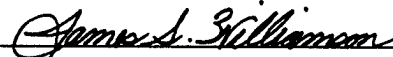
USING GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA INCLUDED IN THIS DOCUMENT FOR ANY PURPOSE OTHER THAN GOVERNMENT PROCUREMENT DOES NOT IN ANY WAY OBLIGATE THE US GOVERNMENT. THE FACT THAT THE GOVERNMENT FORMULATED OR SUPPLIED THE DRAWINGS, SPECIFICATIONS, OR OTHER DATA DOES NOT LICENSE THE HOLDER OR ANY OTHER PERSON OR CORPORATION; OR CONVEY ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY RELATE TO THEM.

THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

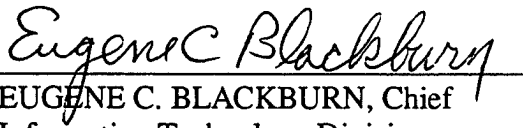
THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.



AL SCARPELLI, Electronics Engineer
Embedded Information Sys Eng Branch
Information Technology Division



JAMES S. WILLIAMSON, Chief
Embedded Information Sys Eng Branch
Information Technology Division



EUGENE C. BLACKBURN, Chief
Information Technology Division
Information Directorate

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE MARCH 1999	3. REPORT TYPE AND DATES COVERED FINAL REPORT FOR 07/01/1996 - 02/28/1999		
4. TITLE AND SUBTITLE PARALLEL VERY HIGH SPEED INTEGRATED CIRCUITS (VHSIC) HARDWARE DESCRIPTION LANGUAGE (VHDL) SIMULATION FOR PERFORMANCE MODELING		5. FUNDING NUMBERS C F33615-96-C-1913 PE 62204 PR 6096 TA 40 WU 35		
6. AUTHOR(S) DR. MOON JUNG CHUNG				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) MICHIGAN STATE UNIVERSITY DEPARTMENT OF COMPUTER SCIENCE EAST LANSING, MI 48824		8. PERFORMING ORGANIZATION REPORT NUMBER		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) INFORMATION DIRECTORATE AIR FORCE RESEARCH LABORATORY AIR FORCE MATERIEL COMMAND WRIGHT-PATTERSON AFB, OH 45433-7334 POC: AL SCARPELLI AFRL/IFTA. 937-255-7698 EXT. 3603		10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-WP-TR-1999-1507		
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED		12b. DISTRIBUTION CODE		
13. ABSTRACT (Maximum 200 words) As the complexity of micro-electronic systems continuously increases, it becomes critical to develop effective tools that can cut the design time and improve the quality of design. DoD needs to develop new tools to be able to simulate large complex systems, and to fully maximize the rapid progress in high performance computing technology occurring today. The goal of this project was to develop and implement efficient paradigms for VHDL simulation on massively parallel processor machines so that we can achieve speed-up of up to a hundred times compared to sequential simulation. Our research focus was on performance and behavioral level simulation. The performance model allows us to find the trade off between various hardware components and architectures. Behavioral simulations are used to prove the functional correctness of the system. The research issues involved in the project were: processor communications, synchronization, and event queue manipulation, deadlock handling, communication latency hiding, and granularity of computation. We have measured the performance of the proposed techniques on various platforms such as the IBM SP2 and SGI Origin 2000, and achieved speed-ups of 31 times.				
14. SUBJECT TERMS VHDL, Parallel Simulation, Time Warp, Performance Modeling		15. NUMBER OF PAGES 130		
		16. PRICE CODE		
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR	

TABLE OF CONTENTS

SUMMARY	1
BACKGROUND	2
1. INTRODUCTION	4
1.1 VHDL	4
1.2 PARALLEL VHDL	4
1.3 GOAL	5
1.4 OBJECTIVE	6
1.5 EXTENSIBILITY	7
1.6 MAJOR TASKS	8
2. PARALLEL SIMULATOR SYSTEM ARCHITECTURE	10
2.1 MAJOR MODULES	11
3. OBJECT MODELING	12
3.1 OBJECT MODEL OF THE SIMULATION ENGINE	14
3.2 SIMULATION ENGINE: LOGICAL PROCESS AND SIMULATION OBJECT	17
3.3 PARTITIONING	18
4. FRONT END INTERFACE	19
4.1 THE OBJECT MODEL	20
4.2 IMPLEMENTATION DETAILS	24
4.3 FUTURE WORK	67
5. ELABORATION AND INTERCONNECTION OBJECTS	68
5.1 INTRODUCTION	68
5.2 OBJECTS IN VHDL	70
5.3 ALGORITHM OVERVIEW	75

5.4 IMPLEMENTATION OVERVIEW	81
6. BENCHMARKS	85
7. CONCLUSIONS	90
APPENDICES	91
APPENDIX A	91
APPENDIX B	93
APPENDIX C	94
APPENDIX D	97
APPENDIX E	98
APPENDIX F	99
APPENDIX G	102
APPENDIX H	108
REFERENCES	121

Figures

2.1 PARALLEL SIMULATOR BLOCK DIAGRAM	10
3.1 OBJECT MODELS	14
3.2 LOGICAL PROCESS	17
4.1 DEFAULT PUBLISHING FORMAT OF SIGNAL ASSIGNMENT	43
4.2 EXAMPLE ON HOW TO PUBLISH EXPRESIONS	46
4.3 THE _PUBLISH_CC_OPERATOR_NAME() METHOD	47
4.4 SYNTAX OF WAIT STATEMENT	55
4.5 WAIT FOR TIMEOUT CLAUSE	59
4.6 CONSTANT DECLARATION SYNTAX	60
4.7 CODE OF CONSTANT DECLARATION	62
4.8 OVERLOADING ASSIGNMENT OPERATOR	65
5.1 INTERCONNECTION OBJECTS	69
5.2 THE ENCODING OF THE NETWORK	69
5.3 DELAYED AND ARCHITECTURE	74
5.4 DECOMPOSITION HIERARCHY	78
6.1 SPEED-UP OF S35932X2	85
6.2 COMPARISON OF TIME WARP AND SYNCHRONOUS PROTOCOLS	86
6.3 SPEED-UP WITH ISCAS S953 FOR GRAIN SIZE 100	87
6.4 SPEED-UP WITH ISCAS S953 FOR GRAIN SIZE 1000	87
6.5 S35932X1 SPEEDUP	88
6.6 SPEEDUP OF S15850	88
6.7 ODD-EVEN SORTER	89

Tables

4.1 FUNCTIONS FOR SIGNAL VARIABLE/REGISTRATION	22
4.2 SYSTEM DEFINED INITIAL VALUES	23
4.3 PREDEFINED PUBLIC DATA ELEMENTS	29

SUMMARY

As the complexity of micro-electronic systems continuously increases, it becomes critical to develop effective tools that can cut the design time and improve the quality of design. DoD needs to develop new tools to be able to simulate large complex systems, and to fully maximize the rapid progress in high performance computing technology occurring today.

The goal of this project was to develop and implement efficient paradigms for VHDL simulation on massively parallel processor machines so that we can achieve speed-up of up to a hundred times compared to sequential simulation. Our research focus was on performance and behavioral level simulation. The performance model allows us to find the trade off between various hardware components and architectures. Behavioral simulations are used to prove the functional correctness of the system.

The research issues involved in the project were: processor communication, synchronization, and event queue manipulation, deadlock handling, communication latency hiding, and granularity of computation. We have measured the performance of the proposed techniques on various platforms such as the IBM SP2 and SGI Origin 2000, and achieved speed-ups of 31 times.

BACKGROUND

With the increasing complexity of micro-electronics systems, validating various models (or virtual prototypes) becomes critically important. There are two approaches to validating the correctness of VLSI design: verification techniques and simulation. Of these two approaches, simulation is still the primary tool. Simulation can be used in a hierarchical fashion. At the top level, performance models are used to explore the trade off between various hardware components and architectures. Behavioral simulations are used to prove the correctness of the system specification. At a lower level, timing simulation is used to validate the correct timing information such as set-up hold times, and is the only practical tool available at this level. For large micro-electronics systems, simulation has become a very time-consuming and critical part of VLSI design. Typically, simulation constitutes about 80% of the design cycle.

VHDL is an IEEE standard hardware description language developed by the DoD. It can be used to describe systems at both behavioral and structural levels. It can also be used to describe a performance model. It is expected that we need to simulate a system up to 100,000 VHDL processes to accurately model a start-of-the-art system. Simulating such a large system can be extremely slow in a sequential machine.

Parallel logic simulation has recently attracted a considerable amount of interest. However, most research is restricted to symmetric multi-processor machines or a MIMD machine with a small number of processors. Moreover, there are only a few benchmark results available based on actual simulation 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. With the hardware resources available for High Performance Computing, such as the 336 node Intel Paragon, and the 400 node IBM SP2, parallel simulation on Massively Parallel Processors (MPP) is now feasible which can achieve a speed-up of up to

several hundred times compared to sequential simulation.

NECESSITY

The research will result in fast parallel simulation techniques which can achieve speed-up of up to hundreds of times compared to sequential simulation. The proposed parallel simulation techniques will provide an attractive solution to better throughput of the overall design environment. Fast parallel simulation of large VLSI systems enables the designer to validate the correctness of the design. The usage of behavioral simulation in an earlier stage of the VLSI system design can prove the functional correctness. By speeding up the performance model simulation, designer can carry out extensive HW/SW trade-off analysis, and allow designers to select the best hardware architecture. The structural level simulation can be used to validate the correctness in logic level and timing. The fast simulator developed in this research will allow designers to validate various virtual prototypes and to test complete fault coverage analysis. Thus, the design cycle can be shortened, and the number of real prototypes to be constructed can be reduced. Parallel simulation tools will allow 4X productivity improvement of RASSP goal.

PROGRAM

This program was funded by the DoD High Performance Computing Modernization Office (HPCMO) under the Common High Performance Computing Software Support Initiative (CHSSI), Computational Electronics and Nanoelectronics area. This area was led by Dr. Barry S Perlman, U.S. Army CECOM, Fort Monmouth NJ. Preliminary research was performed by the Army Research Laboratory (ARL) at Fort Monmouth as a joint research effort. Several team members at ARL were involved in parallel simulation, VHDL, and testing.

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.

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

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. 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 multi-processor 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.

1.3 GOAL

The goal of the Parallel VHDL Simulation Project at Michigan State University was 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 uses MPI, a standard parallel communication protocol. The developed parallel program is scalable and portable. From a user's point of view, the output of this development effort is a fast parallel program that is 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.

1.4 OBJECTIVE

The required software system is a behavioral simulator for digital micro-electronic systems. The simulator accepts a VHDL description of a digital system to be simulated and the test vectors. The output of the software is the values of signals and the event times when such changes occur. VHDL is used widely to describe models of digital electronic systems. The designer can simulate and validate the various levels of VHDL models from the system level architecture to detailed design. Performance model is a term used to denote the modeling in which the focus is on the behavior of the system in terms of available resources and computational requirements (such as input rate, network queues, and computational resources).

From a user 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. Using the 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. Thus, the design cycle can be shortened, and the number of real prototypes to be constructed can be reduced.

A subset of VHDL constructs was selected to describe the performance and behavioral models. The VHDL constructs selected are powerful enough to describe the behavior and function of any hardware systems, yet simple enough so that models written using the constructs can be efficiently simulated in parallel.

The following issues have been addressed in the proposed research.

- Processor communication, synchronization
- Event queue manipulation
- Deadlock handling in conservative mechanisms

- Granularity of each computation between synchronization points

Using VHDL benchmark suites, the performance of the developed techniques on the SP2 and Origin 2000 has been compared with that of other simulation techniques. The performance of the proposed techniques was evaluated in terms of the following criteria: simulation cycles, parallelism, maximum event queue size, speed-up, and memory space reduction.

1.5 EXTENSIBILITY

Another goal of this simulation project was 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 may be run on any parallel platform.

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.

1.6 MAJOR TASKS

The following is a list of major tasks performed 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.
- **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, and 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 a model written using these constructs may be efficiently simulated in parallel.

- **Develop the Front End Interface** which generates C++ programs from VHDL descriptions using the object model. The task of the Front End Interface is implementing these VHDL constructs. To translate VHDL into C++, the Front End Interface has to face the following problems:
 - VHDL is a very complex and rich language. Translating its constructs requires

full understanding of its semantics and compiler techniques.

- There is no direct mapping between the two languages. Some intermediate form must be used 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 by 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. The intermediate form approach is also used in this project. The SAVANT [31,34] 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 specification [33]), which is used to generate the parse tree.
- **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 Benchmark programs** to measure the efficiency of parallel programs. Perform benchmarking on SP2 and Origin 2000.

2. PARALLEL SIMULATOR SYSTEM ARCHITECTURE

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.

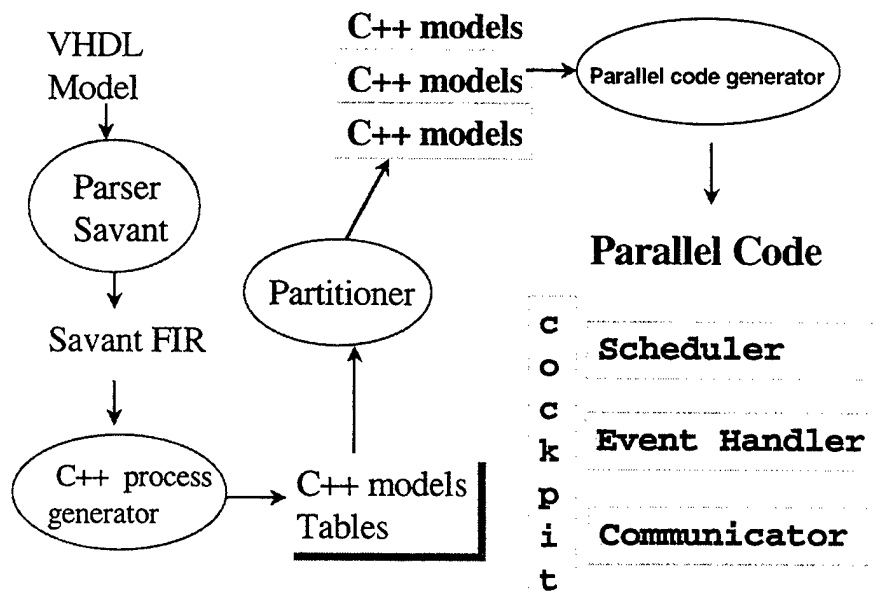


Figure 2.1 Parallel Simulator Block Diagram

2.1 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, using the SAVANT scam VHDL analyzer.
- **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.

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.

3. OBJECT MODELING

In practice, object modeling aims to be suitable for the representation and manipulation of complex objects of engineering design. In most current simulation systems, users must know the parallel platforms and the simulation technique to develop the model. As a new simulation technique is invented and new hardware introduced, the models they developed are not reusable, and must be modified. Moreover, the modeling techniques are different for each application domain.

There has been much research work in modeling simulation. Several simulation languages have been proposed including Simula, GASP, GPSS, CSIM, and Sim++. In general, they can be classified into three different approaches:

- an approach which is limited into a specific application domain,
- a fixed model for discrete event simulation
- a new simulation language to model the objects

In these approaches, the application domain is limited and the system is not extensible. Modifying simulation models frequently requires changing the simulation scheme employed. Moreover, adding a new simulation mechanism or modifying data structures may affect the models already developed. For example, the Tyvis simulator [32] is specifically targeted for parallel VHDL simulation using Time Warp. In this approach, all objects are a subclass of Time Warp objects. Therefore, adding a new simulation scheme requires changes of all models already developed. DEVS [29] has been proposed by Ziegler as a model of distributed event simulation. All objects must be modeled under DEVS formalism. Even though this approach may have the advantage of introducing the formalism, it puts a burden on the modeler in a certain application domain. Simulation languages such as Yaddes[21], Maisie[3], SIMA[23] and MOOSE[12,14] have been proposed to aid the user to develop models. These languages are also associated with a fixed set of pre-selected set of simulation mechanisms. Yaddes is a distributed event driven specification language that resembles Yacc and Lex. A Yaddes program is translated into a

C program which is later linked together with a run time support library. SPEEDES [25] is a C++ based simulation environment developed at the Jet Propulsion Lab which supports sequential, Time Warp [15], and Time Bucket Algorithms. In SPEEDES, end users may adjust a predefined set of parameters to improve the speed.

Only a few research results have been reported on the modeling and the performance of parallel simulators. Maisie is developed for efficient execution of the simulator. Depending on the hardware platforms, the complexity of the model, and the frequency of messages generated, the performance of parallel simulation varies greatly. Most of the parallel simulators discussed above require the end user to provide their own models as the simulation scheme changes. The only exception is Maisie. In Maisie, objects must be defined using Maisie language.

Experience has shown that simulation is evolutionary in nature. While requirements change, the system being simulated also changes. The modeling should be less resistive to such changes, so the maintenance of the evolving system will be much easier [6]. In this section, we outline our simulation engine that is developed using an object-oriented and layered approach. Our proposed object modeling technique has the following features.

- It allows users to model complex objects which consist of different type of objects with various level of complexity.
- It is simple so that the modeler should be able to model the objects in stepwise refinement. The object modeling technique must provide a way of hiding the information so that modelers and users are not overwhelmed by the details of the objects.
- It separates models and simulation scheme so that modelers should be able to develop models without knowing the simulation scheme employed.
- It provides a mechanism for parallel programmers to develop library modules and data structures independent from simulation models.

3.1 OBJECT MODEL OF THE SIMULATION ENGINE

Our models are cleanly separated from the execution environment, and parallel platform. There are three types of objects in our proposed modeling: *Application Objects*, *Simulation Objects*, and *Simulation Scheme Objects*. Application Objects model the behavior of objects in the application domain to be simulated. They can be developed by modelers of a specific application domain, and are independent from the simulation schemes and hardware platforms. Simulation Scheme Objects depend on simulation scheme used, and include all library modules necessary to run the simulator on a parallel platform. They form the core of the simulation engine, and are developed by parallel programmers with expertise in Time Warp and Chandy-Misra schemes [19]. The system to be simulated consists of many Application Objects. An end user selects the simulation scheme to be used, and instantiates Simulation Application Objects (Simulation Objects in short). As shown in Figure 3.1, a simulation object inherits behaviors and properties from Application Objects, and parallel run time methods from the Simulation Scheme Objects. For example, 166MHz CPU and 200MHz CPU inherit their instruction set from the Pentium class object. To simulate a computer system using Time-Warp, an end-user instantiates an object that inherits its behavior from the 200MHz Pentium object, and its run-time environment and data structures from TW-OBJ, a Time-Warp object.

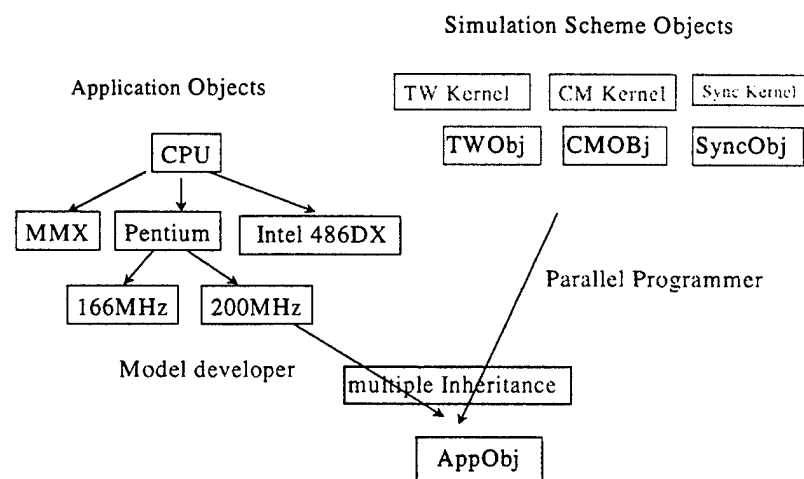


Figure 3.1 Object Models

The modelers populate the libraries of object models independent from the simulation scheme. The rationale of our approach is to allow the front end user to use the application objects developed by the modeler freely. Our object modeling technique provides freedom to three parties of the simulation arena: modelers, parallel programmers, and front end users. Parallel programmers (simulation scheme experts) may concentrate on providing various parallel simulation methods. In other words, once the modeler develops Application Objects, the same objects can be simulated with other simulation schemes, without remodeling them for the new simulation scheme.

Application Objects are organized using the inheritance mechanism among them. Therefore the modeler, with minimum knowledge of modeling language and object oriented modeling, is able to create a library of objects. The front-end user, who simulates the system, mixes and matches the models developed by modelers and the simulation scheme developed by parallel programmers. The simulation scheme class defines the data and methods that each simulation object needs to operate within the system. This class can be viewed as the kernel of the simulation and any type of the simulation domain can use that scheme via its interfaces. Depending on a particular simulation scheme, a simulation scheme object is created. The simulation scheme objects include all the necessary methods for that specific scheme.

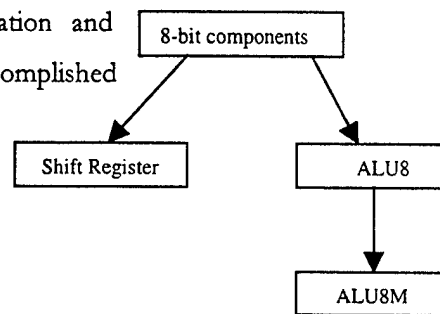
A Simulation Application Object (Simulation Object) is an instance of a class that is obtained from integration of the Application Object class and Simulation Scheme class. It contains the implementation of the methods of the simulation scheme inherited from Simulation Scheme Objects. It inherits its behavior and attributes from Application Objects. For the simplicity of implementation and portability, we used the template class approach rather than the multiple inheritance mechanism. A Simulation Object is responsible for simulating the single object and generating events.

The following class definition shows the skeleton of an Application Object. Other instance variables, member functions, and class relationships can be added into the

definition. If the parent class of the object has member functions of instance variables, the

object does not need to include these fields. Consider the following example. We want to model an 8-bit ALU in the design of 8-bit microprocessor. All components of an 8-bit microprocessor share common characteristics, such as design rules, bus width, minimum gate delays etc. Therefore it is a subclass of 8-bit components. Suppose that the behavior of the 8-bit ALU has the following specifications: 8-bit Input port and Output port, 6-bit control line, and 8-bit addition and subtraction. Later ALU8M can be modeled which is a

special case of ALU8 by providing multiplication and division operations. This specialization can be accomplished by making ALU8M a subclass of ALU8, and inheriting operations of ALU8M from ALU8:



Here is the sample code for the ALU object:

```

class ALU8 {
    public: // Methods that simulates the behavior of the object
        void add();
        void subtract();
        void exception();
        void executeProcess(); };

class UserState : public BasicState { // input and output port of the object
    InSignal<int> x[8], y[8], opcode[2]; // opcode denotes the operation
    OutSignal<int> z[8]; };
  
```

The method `executeProcess()` simulates the behavior of the object. This is the main part of the modeling and the modeler just needs to plug the behavior of the object into the class definition, by conforming to the names that are declared within the class. In this sample code, we declare two other functions, `add()` and `subtract()` that are used within `executeProcess()` method.

To create a model of an ALU with more functions, such as multiplication and division, we can simply model by inheriting from ALU8:

```

class ALU8M : public ALU8 {
    public:
        void multiply();
        void divide();
        void executeProcess(); };
  
```

To simulate the ALU8M, the behaviors of the 8-bit ALU such as add, subtract, and exceptions are inherited from the class of ALU8. ALU8M has its own functions, multiply and divide, which are not defined in ALU8. Moreover, ALU8M can implement its own methods for ADD and SUB.

The sample C++ code is provided as follows:

```
void ALU8M::executeProcess() {
    switch (opcode) {
        case 0: add(); break;
        case 1: subtract(); break;
        case 2: multiply(); break;
        case 3: divide(); break;
        default: exception(); break;
    }
}
```

3.2 SIMULATION ENGINE: LOGICAL PROCESS AND SIMULATION OBJECT

Each processor is an instance of a logical process(LP), which is the simulation engine of the processor. Logical Process is responsible for the global flow-control of the simulation. It instantiates the simulation objects assigned to that particular processor. During the

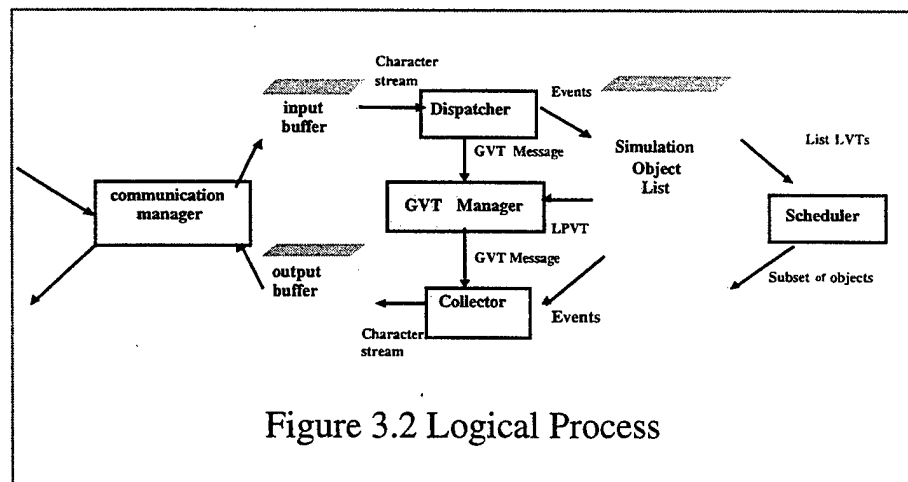


Figure 3.2 Logical Process

execution of the simulator, LP handles the communication of simulation objects via messages, schedules events by selecting simulation objects to be executed in next

simulation cycle, and computes the Global Virtual Time. The LP object depends on the simulation scheme. It consists of Simulation Scheme Objects. Figure 3.2 shows the LP object organization. The Scheduler schedules events based on scheduling policy. Dispatcher reads input messages from the buffer and places events into the event queue of the corresponding simulation object. Messages to be sent are handled by the collector, which aggregate messages and send them according to the communication policy.

3.3 PARTITIONING

We have developed three different partitioning schemes: random, level, and duplication. In random partitioning, objects are assigned to processors randomly so that each processor has an equal number of objects. In level partitioning, objects are partitioned according to depth such that objects in the same depth (from the primary input) are assigned to the same partition. It is well known that random partitioning in general has good performance. We achieved similar performance except a few cases, in which case we used level partitioning. Another scheme we have used is duplication of objects. If a certain object, particularly an input object, has a high output degree then it is much better to duplicate the object. Indeed duplicating objects increases the activity rate and significantly improves the speed of parallel simulation.

4. FRONT END USER INTERFACE

The Front End Interface analyzes VHDL source and generates C++ according to the Simulation Object Model, particularly application objects and simulation objects. 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 IIRPvhdll 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 IIRPvhdll 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 their semantic correctness. Up to now, 47 IIRPvhdll layer classes have been added to the SAVANT class hierarchy and about 6000 lines of C++ code has 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 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.

4.1 THE 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 simulatable objects. When translating a target system into simulatable classes, the BasicObject class should be used as the parent class of all resulting simulatable classes. The BasicObject class defines input signals, output signals, states, and a method called executeProcess(). Through the input and out signals, BasicObject classes can interact with each other. States are used to keep private information for the BasicObject itself. 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 they can be used. Please refer to section 3 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

simulatable class. The ``in" signals in the VHDL process are translated into the input signals in the simulatable 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

The basic format of a class is like this: its declaration shows BasicObject is its parent class; it has 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, all signals and states are registered and initialized.

The following code shows an example C++ class generated using the object model. This example only shows a general structure of the object model C++ classes, it is not the exact code. Appendix B shows the VHDL source code of an AND gate. The exact C++ code for this AND gate is shown in Appendix C.

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

```

VHDL variables are declared within processes (global variables have 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 4.1 shows these functions.

Input Signal	RegisterInSignal(&signal, sizeof(signal))
Output signal	RegisterOutSignal(&signal, sizeof(signal))
State	registerState(&state, sizeof(state))

Table 4.1 Functions For Signal/Variable Registration

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, **a** is

initialized by $a = 0$. If the signal or state doesn't have an initial value, system defined initial values are used. Table 4.2 shows the predefined initial values.

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

Signal/State	Initial Value	Definition
Input signal	X	#define X-1
Output Signal	Y	#define Y-1
State	X	As Above

Table 4.2 System Defined Initial Values

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 .

A 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 [33]. 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 there is no documentation at all which describes 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 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.

4.2 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.

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.

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.

The included file is compiled together with simulation objects and utilities we have developed.

How to Insert an IIRPvhdI Class

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

As an example, let's look at how to insert the IIRPvhdI_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 IIRPvhdI_ProcessStatement class, the IIR_ProcessStatement needs to be derived from the IIRPvhdI_ProcessStatement class. The

IIR_ProcessStatement.hh is then modified like this:

```
#include "IIRPvhd1_ProcessStatement.hh"

class IIR_ProcessStatement : public
IIRPvhd1_ProcessStatement {
    ...
}
```

On the other hand, the IIRPvhd1_ProcessStatement has to be derived directly from the IIRScram_ProcessStatement class. The IIRPvhd1_ProcessStatement.hh looks like this:

```
#include "IIRScram_ProcessStatement.hh"

class IIRPvhd1_ProcessStatement : public
IIRScram_ProcessStatement {
    ...
}
```

Other IIRPvhd1 layer classes should be added to the class hierarchy in a similar manner.

The IIRPvhd1_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 IIRPvhd1_DesignFile class has basically 3 functions:

- Use preprocessor **#ifndef ... #define** to protect the "Classes.h" file.
- Include some ".h" files and use "typedef" to define some data types, like

SavantbitType, SavantintegerType, and SavantrealType.

- 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 [34, 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 details of the `IIRScram_LibraryUnitList` class, please refer to [34, `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 define its corresponding `IIRPvhdl` layer class to shadow it. As a result, not every `IIRScram` layer class has a corresponding `IIRPvhdl` layer class.

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 4.3.

DATA MEMBER TYPE	DATA MEMBER NAME
<code>IIR_GenericList</code>	<code>Generic_clause</code>
<code>IIR_PortList</code>	<code>Port_clause</code>
<code>IIR_DeclarationList</code>	<code>Entity_declarative_part</code>
<code>IIR_ConcurrentStatementList</code>	<code>Entity_statement_part</code>

IIR_DesignUnitList	Architectures
--------------------	---------------

Table 4.3 Predefined Public Data Elements of IIR_Entity Declaration

These data elements are all lists which keep entity related information, such as generic declarations, ports, etc. The IIRScram_EntityDeclaration class uses all of 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 IIRPvhdI_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 parts of the front end interface. This is left to the decision of future participants of this project.

The IIRPvhdI_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 IIRPvhdI_ArchitectureDeclaration 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 [34, `IIRPvhdl_ArchitectureDeclaration.cc`].

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_ProcessStatement` 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 an 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 appearing 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 appearing on the left 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 the entity declaration statement and architecture declaration statement declare signals. Within 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 [34, 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 lists.

In file `IIRPvhd1.hh`, two global variables are declared to keep the two lists:

```

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 `IIRPvhd1` layer classes, such as `IIRPvhd1_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, there is only one IIR node for “c” added to the set. But there are two “arr(1)” IIR nodes added. This may seem strange. The reason Savant does this is that TyVIS requires different object for different array element. Since the SAVANT publisher is TyVIS oriented, 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

A VHDL process usually contains a 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 a child class of the `IIRScram_SequentialStatement` class. They may all contain a 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 the 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. Only when the publishing file name is changed is 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 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 the 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 a declaration item is a variable declaration, we can use the predefined “get_kind()” function to get its type. If the return value is “IIR_Variable_Declaration”, this item is a variable declaration.

We can then publish this declaration item as a variable.

Publishing variable declarations is similar to publishing signal declarations. The only difference is that no prefix is added 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 signals, output signals, and states 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 are used 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 [24].

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 signal or variable 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 needs 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 `IIRPvhd` class is the root class of the `IIRPvhd` layer. The reason to add this class is to declare some global data elements and public functions for the `IIRPvhd` layer classes. The following data and functions are declared in the `IIRPvhd` 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 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 array 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 elements 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 [34, **IIRPvhdl.cc**].

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 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 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 4.1 is an example of the publishing format.

VHDL Source Code

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

PUBLISHED C++ CODE

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

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

FIGURE 4.1 DEFAULT PUBLISHING FORMAT OF SIGNAL ASSIGNMENT

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 be 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 field 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.

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 publishes the system-publishing prefix.

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_EquityOperator` 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 `IR_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_EQUALITY_OPERATOR"`, it means the current operator is an "equality" operator and we can publish the corresponding `"=="` operator in C++. Figure 4.2 is a code segment of the `_publish_cc()` method of the `IIRPvhdI_DyadicOperator` class.

```
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 4.2 Example on How to Publish Expressions

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 4.3 shows this function of the IIRPvhdL_AndOperator class.

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

FIGURE 4.3 THE _PUBLISH_CC_OPERATOR_NAME() METHOD OF IIRPVHDL_ANDOPERATOR

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 4.2. Please refer to [34, IIRPvhdL_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 [34, IIRPvhdl_MonadicOperator.cc].

The If Statement

Like C++, VHDL has a “**if..then..else**” statement which evaluates a condition 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 “elseif” 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 [34, 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 [34,

IIRPvhdl_Elsif.cc].

The Case Statement

VHDL has a “Case” statement in which the behavior depends on the value of a single expression. Different evaluations 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:

- 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.

- Publish “`if(false){}`” as the “then” branch of the C++ If statement, so that all “Case” alternatives can be published as “elseif” branches.
- Call the `_publish_cc()` method of the `case_statement_alternatives` to publish the “Case” alternatives.
- Restore the old `_current_publish_node` value.

For programming details please refer to [34, `IIRPvhdI_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 `IIRPvhdI_CaseStatementAlternativeByExpression` publishes the “Case” alternative as a “elseif” branch of the If statement. It does the following things:

- Publish the “else if” string.
- 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`.
- Publish the “==” string.
- Call `get_choice()->_publish_cc()` to publish the choice expression.
- Publish the “)”.
- Call `sequence_of_statements._publish_cc()` to publish the sequential statement of this “Case” alternative.

Please refer to [34, `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 [34, `IIRPvhdl_CaseStatementAlternativeByOthers.cc`].

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 after 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 a 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 referenced 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 [34, IIRPvhdL_ForLoopStatement.cc].

The While Loop Statement

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 statements 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 is returned by the method. Coding is straightforward. For coding details please refer to [34, IIRPvhdl_WhileLoopStatement.cc].

The Wait Statement

VHDL has a Wait statement, which does not have a 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 are multiple Wait statements in the process, the process executes only until the next Wait statement is encountered [1, page 168-169].

Syntax

Wait statement is a sequential statement with the syntax rule shown in Figure 4.4.

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

FIGURE 4.4 SYNTAX OF WAIT STATEMENT

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 specifies 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 [1, page 114-116].

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 (a 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 a Wait statement requires that a 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 published at the beginning of each **executeProcess()** function using the value of "P" as the 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 a 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 a wait_id. For details please refer to [34, 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()** functions return 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 registered 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 4.5.

```
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 4.5 EXAMPLE OF WAIT FOR TIMEOUT CLAUSE

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 [1, page 116]. This means the sensitivity clause has higher priority than the condition clause. The If condition 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 [34, IIRPvhdl_WaitStatement.cc].

Process with Sensitivity List

SAVANT transforms a process with a sensitivity list into a 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 a process with sensitivity list statement).

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

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 4.6 CONSTANT DECLARATION SYNTAX

The syntax of VHDL constant declaration and the C++ constant declaration syntax is shown in Figure 4.6.

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 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 4.7.

```
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 4.7 CODE OF CONSTANT DECLARATION

Coding is quite straightforward. For details, please refer to [34, IIRPvhdl_DeclarationList.cc].

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 a 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 (for coding details please refer to [34, IIRPvhdl_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 be considered for enumeration types. For programming details, please refer to [34, `IIRPvhdl_TypeDeclaration.cc`, `IIRPvhdl_EnumerationTypeDefinition.cc`].

Array Types

VHDL supports array types. Unlike C++, array types 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 arrays 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 [34, 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 templates 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 4.8 shows how the assignment operator is overloaded by the one dimensional template array class. For now, the following operators have been overloaded: +, -, *, /, =, !=, \&, |, ^, =. The “=(int)” operation 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.

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

FIGURE 4.8 OVERLOADING ASSIGNMENT OPERATOR FOR ONE DIMENSIONAL ARRAY CLASS

Right now, SAVANT only supports the 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 to 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 [34, IIRPvhdl_ArraySubtypeDefinition.cc].

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 elements is published by its type and name. Also, some operators are overloaded for the record, such as addition, etc. The reason is to support arrays of record. Since the template array classes have overloaded some operators, each defined record type has to overload the same operators. For coding details, please refer to [34, IIRPvhdl_RecordTypeDefinition.cc].

4.3 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.

5. ELABORATION AND INTERCONNECTION OBJECTS

5.1 INTRODUCTION

In this chapter we will present the representation scheme that we've used to encode the interconnection of the networked objects of any VHDL design, or any system in general. In the first section, we spend some time over this subject and the need for extracting this encoded form of interconnection, and also some conventions that we use. In section two, we present the algorithm that we developed for this purpose along with a simple example to clarify the algorithm.

Any system subject to simulation, consists of objects that are interconnected. If MPI (Message Passing Interface) is used to model the system, then objects communicate with each other by sending and receiving messages. Communication, or message passing, is done through the network in which objects are interconnected. In other words, each object could send/receive messages to/from objects that it is directly connected to in that network configuration. Hence, each object should have information about the source of the incoming messages, and also the destination of the out going messages. This implies that we need a representation scheme to present objects in some encoded fashion, so that each object could be uniquely identified in the network. One way would be to assign unique **object IDs** to each object, then each object will be uniquely identified in that network. Doing this, we have solved just part of the problem. Usually objects are connected to several other objects via separate links/channels/ports and behavior of the object may depend on the events it receives from certain channels. For example in logic gate level simulation, an AND gate may be modeled as an object with two input ports

which receive events) and one output port (which sends events). Similarly, any object may be modeled as a multiple port object, some of them to receive messages and some of them to send messages. We call the former **input port** and the latter **output port**. This shows that object IDs on their own are not complete for a representation scheme, since they do not give any information about the ports of the object, where other objects might be associated with. One solution would be, again to assign **port IDs** for each input and output ports and also tag them with input label or output label (to differentiate whether it is an input or output port). To examine this representation scheme, let's apply it to a very simple network as follows. In the Figure 5.1, the interconnection of objects is presented using a *directed graph* and in the Figure 5.2, we have used the proposed representation scheme to encode the network.

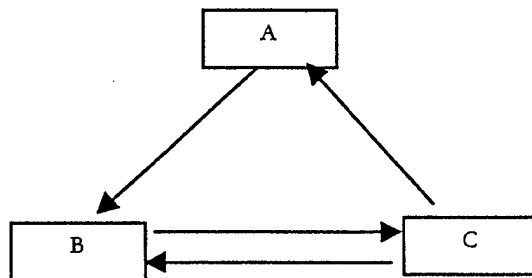


Figure 5.1 Interconnection Objects

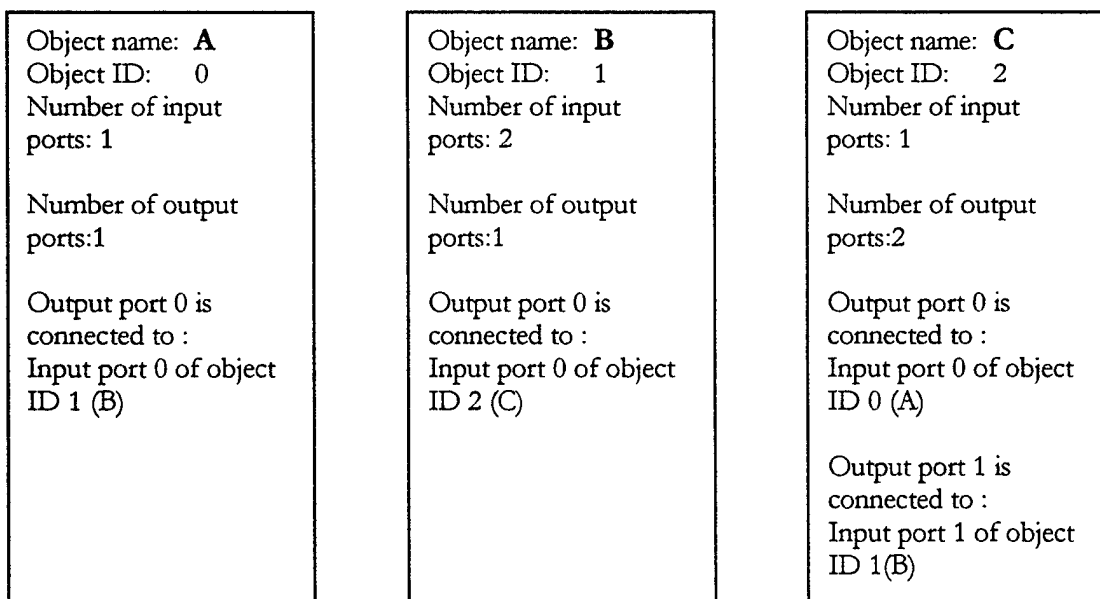


Figure 5.2 The Encoding of the Network

For this network, the proposed representation scheme covers the whole interconnection information that might be necessary in order to handle communication. Note that we do not need any convention about how we assign object IDs to the objects of the network. As long as we assign unique IDs in any arbitrary order to the objects, that works fine and the same statement is true about the port IDs. This is the basic representation that we use as part of our representation scheme for the **VHDL** systems.

5.2 OBJECTS IN VHDL

In this section we will focus on the systems that are modeled by VHDL. We develop our basic representation scheme that we started in section 5.1. However, before getting into all the details, we briefly talk about the tools that we used to extract interconnection information from a VHDL design. In order to visualize and facilitate understanding of our representation scheme and also our algorithm, we will set some conventions on showing objects in the system.

VHDL and SAVANT

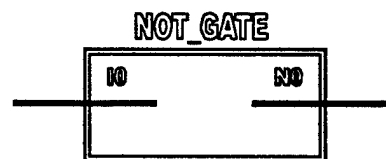
VHDL is a language to model hardware, or in general any concurrent networked system and is used widely to model and simulate different hardware for design and production purposes. Our main goal for this project as stated earlier, is to establish a front-end interface between VHDL designs and our simulation kernel. Front-end interface consists of two main parts:

Functional modeling of the VHDL objects into C++ classes. In this part of the front-end interface, VHDL objects are translated into C++, so that later we can plug them into our simulation kernel. Extracting the interconnection information to handle the communication of those objects is also necessary.

The focus of this section is on the second part. However for both cases, we need a tool to facilitate this process to apply on VHDL source code. We have chosen SAVANT as a tool for this purpose.

The main section of any VHDL design is defined in its architecture statement part, where all the signal assignments and component instantiation are specified. SAVANT automatically converts each signal assignment into a process statement containing that signal assignment and a wait statement on that signal. Any other VHDL architecture statement construction could be either a signal assignment, process statement, component instantiation or generate for statement, which in turn is used either as component instantiation or signal assignment generator. Based on the statement we made a few lines before, each signal assignment is converted to a process statement, and each component instantiation recursively has an architecture statement part, which again may include another layer of component instantiation, signal assignment, etc. Therefore we will see that the bottom line objects of any VHDL design are just processes. In other words processes are the basic objects which can not be further split up into more basic objects. We use this concept of object during the whole section. Now, let's spend some time on our convention of displaying objects. Let's start with a very simple VHDL code.

```
ENTITY NOT_GATE is  
  PORT(I0 : IN BIT; N0 : OUT BIT);  
END NOT;
```



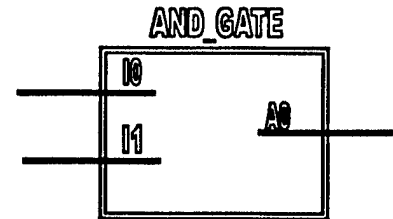
```
ARCHITECTURE behavioral of NOT_GATE is  
BEGIN  
  p1:PROCESS(I0)  
  BEGIN  
    N0 <= NOT I0;  
  END PROCESS;  
END behavioral;
```

This VHDL code models a not gate. If you notice the input port and output port characteristics are specified in the PORT(.) section of the design, which means this module communicates with other modules through these ports. On the right, we used a figure to

show the data functional model of this design. Each VHDL design is displayed by a box labeled with the entity name. We also display the input and output ports. If it contains multiple basic objects (several concurrent signal assignment or component instantiation), we also show them. The simple blank box is just used for the basic objects (processes). In the similar way, we can display an and gate, which only contains one process statement:

```
ENTITY AND_GATE is
  PORT(I0, I1 : IN BIT; A0 : OUT BIT);
END AND_GATE;

ARCHITECTURE behavioral of AND_GATE is
BEGIN
  p1:PROCESS(I0, I1)
  BEGIN
    A0 <= I0 AND I1;
  END PROCESS;
END behavioral;
```



Now assume the following code where this not gate is instantiated in another VHDL design (buffer):

```
ENTITY BUFFER_GATE is
  PORT(I0 : IN BIT; B0 : OUT BIT);
END BUFFER_GATE;

ARCHITECTURE structural of BUFFER_GATE is
  COMPONENT NOT_GATE
    PORT(I0 : IN BIT; N0 : OUT BIT);
  END COMPONENT;

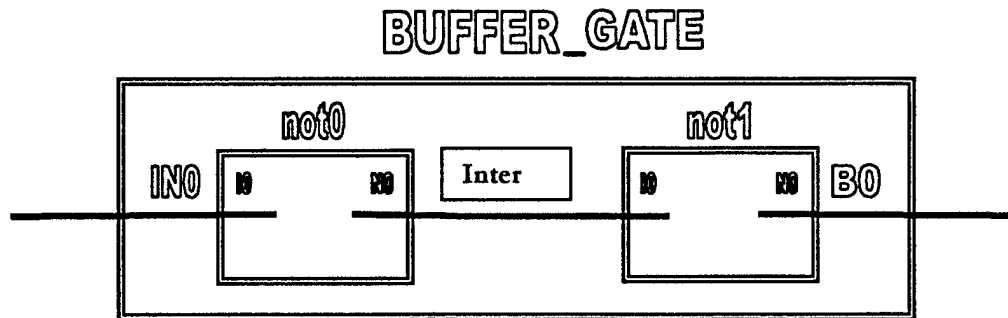
  FOR not0, not1: NOT_GATE USE ENTITY WORK.NOT_GATE(behavioral)
    PORT MAP (I0, N0);

  SIGNAL inter : BIT='0';

BEGIN

  not0: NOT_GATE PORT MAP(I0=>IN0, N0=>inter);
  not1: NOT_GATE PORT MAP(I0=>inter, N0=>B0);

END structural;
```



Since this buffer uses two instances of the not gate, it is not a basic object and therefore we also display the components that it contains. If you notice we have also included the intermediate signal to show how these two components are connected. The buffer might be instantiated in other VHDL designs, as in the Delayed_AND:

```
ENTITY Delayed_AND is
  PORT(X0, X1 : IN BIT; Y0 : OUT BIT);
END Delayed_AND;
```

ARCHITECTURE structural of Delayed_AND is

```
COMPONENT AND_GATE
  PORT(I0, I1 : IN BIT; A0 : OUT BIT);
END COMPONENT;
```

```
COMPONENT BUFFER_GATE
  PORT(IN0 : IN BIT; B0 : OUT BIT);
END COMPONENT;
```

```
FOR and0: AND_GATE USE ENTITY WORK.AND_GATE(behavioral)
  PORT MAP (I0, I1,A0);
FOR buf0, buf1: BUF_GATE USE ENTITY WORK.BUFFER_GATE(behavioral)
  PORT MAP (IN0, B0);
```

```
SIGNAL inter0, inter1 : BIT:= '0';
```

```
BEGIN
```

```
  buf0: BUFFER_GATE PORT MAP(IN0=>X0, B0=>inter0);
  buf1: BUFFER_GATE PORT MAP(IN0=>X1, B0=>inter1);
  and0: AND_GATE PORT MAP(I0=>inter0, I1=>inter1, A0=>Y0);
END structural;
```

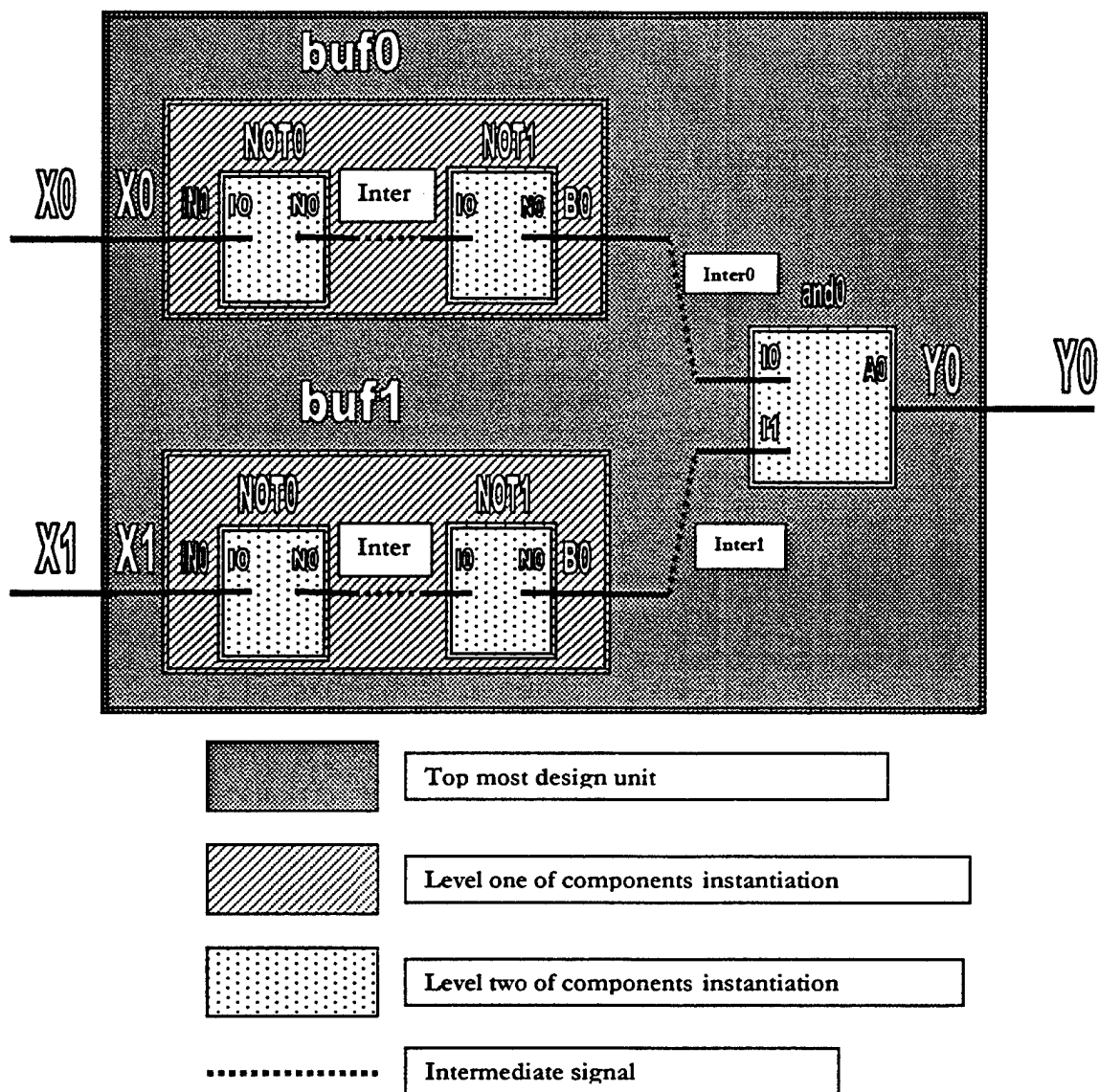
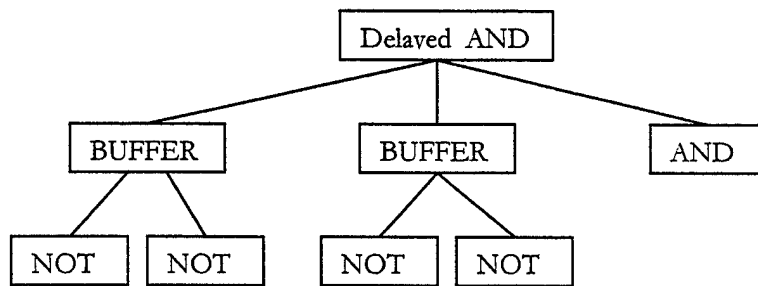



Figure 5.3. Delayed And Architecture

If you notice, this notation captures the way all the basic objects are connected, without concerning their functional behavior, and this is what we are exactly looking for, to extract the interconnection information. The reason we introduce this notation is to give a better understanding of the algorithm we have used. In addition to this notation, we also use a tree graph to show the general interconnection of the objects as follows:



This figure simply shows that the top most design is Delayed_AND which includes three components: two instances of buffer and one instance of and. Each buffer component includes two instances of not. We can go further down the tree, since all of the leaves of the tree are basic objects. As you notice, it is good to think of any VHDL design as a tree, where the leaves of the tree are the basic objects. This figure also tells us that in order to find the interconnection information among the basic objects, we need to traverse the tree down to the leaves to obtain such information.

5.3 ALGORITHM OVERVIEW

The algorithm can be stated in three steps:

1. Identify the basic objects (processes) and assign globally unique Ids to each.
2. For each basic object, assign locally unique port Ids to each input and output port and also intermediate signals.
3. For each basic object, find the destinations of its output ports in terms of couples (global ID, port ID).

This is a very general form of the algorithm and we at this point keep it in this way to help understanding. Later we will explain the detailed algorithm in more depth. Now we show how the algorithm works and we explain it based on the VHDL design Delayed_AND. SAVANT keeps the information about any VHDL design, similar to the tree graph that we pointed out earlier. In SAVANT we can start with the top most design as the root of that design, and by traversing the tree, we can reach to any node of the tree. If you look at the Delayed_AND design, signal X0 is mapped to IN0 of buf0, and inside the buffer VHDL code, IN0 is mapped to the I0 of not0. We can simply write down this mapping information for all of the signals manually:

```

X0 ⇔ IN0 (of buf0) ⇔ I0 (of not0 of buf0)
X1 ⇔ IN0 (of buf1) ⇔ I0 (of not0 of buf1)

inter (of buf0) ⇔ N0 (of not0 of buf0) ⇔ I0 (of not1
of buf0)
inter (of buf1) ⇔ N0 (of not0 of buf1) ⇔ I0 (of not1
of buf1)

inter0 ⇔ N0 (of not1 of buf0) ⇔ I0 (of and0)
inter1 ⇔ N0 (of not1 of buf1) ⇔ I1 (of and0)

A0 (of and0) ⇔ Y0

```

Unfortunately, SAVANT does not support a data structure that gives you the list of the signals that are linked to any particular signal. In other words, there is no data structure in SAVANT, such that you submit a query for a signal, for example X0 and it returns a list of signals (like IN0, I0) that are connected to this signal. This implies that we have to traverse the tree of the design rooted at the top most design and extract such information. We need to encode this information, in terms of numbers, so that the simulator can efficiently use them. To see how this works, let's apply the first two steps of the proposed algorithm to the Delayed_AND design:

Object name	Object ID	Number of output ports	Port ID and port type of this Output	Number of input ports	Port ID of the first input	Port ID of the second input
Not0 (of buf0)	0	1	(0,output) (N0)	1	(0,input) (I0)	-
Not1 (of buf0)	1	1	(0,output) (N0)	1	(0,input) (I0)	-
Not0 (of buf1)	2	1	(0,output) (N0)	1	(0,input) (I0)	-
Not1 (of buf1)	3	1	(0,output) (N0)	1	(0,input) (I0)	-
And0	4	1	(0,output) (A0)	2	(0,input) (I0)	(1,input) (I1)

So far we have assigned unique object IDs to each object and also we have assigned locally unique port IDs to each port of the object. Lets now summarize our ID assignment ruling system:

1. Each basic object is assigned to a globally unique ID and the order of assigning IDs is arbitrary. For example if there are N basic objects in the system, we assign Ids 0 to N-1 to these N objects. We can assign them in any arbitrary order. In our example, there are five basic objects and we have assigned them IDs 0 to 4.
2. For each object, we assign locally unique port IDs to each of input and output ports. For example if a basic object has m inputs and n outputs, we assign IDs 0 to m-1 to the inputs and IDs 0 to n-1 to the outputs. In our example, if we pick and0, it has 2 inputs and one output. Therefore we assign 0 and 1 to the inputs and 0 to the output.
3. In addition to port IDs, we also need labels to show whether this port is input or output, otherwise we will not be able to distinguish input or output ports, based on the port IDs (since port IDs for both input and output start from 0).
4. We claim that with this information, each object can be uniquely identified and moreover, each port of each object is also uniquely identified. For example, object not0 of buf0, can be represented as (objectID:00) or output port of the not1 of buf1 can be represented as (objectID:3, portID:0, port_type:output). Or for example the encoded form (2,0,input), tells us about input port with port ID 0 of basic object 2,

which is I0 of not0 of buf0.

So far, we have applied the first and second steps of the algorithm. The most difficult part of the algorithm is the third part which is to obtain a destination list for each output port of all basic objects. We are interested in extracting information as specified in Figure 5.3 to find out how inputs and outputs of the basic objects are connected to each other.

Connectivity, is symmetric and transitive, which means “if a is connected to b” then:

- “b is connected to a”
- and if “b is connected to c”, then “a is also connected to c”

The transitivity rule, plays a very important role in extracting interconnection information. For example, consider not0 and not1 of buf0. There is no explicit information in SAVANT that tells us buf0:not0:N0 is connected to buf0:not1:I0, but if we extract some information as specified in Figure 5.3, then we can use transitivity property and conclude this:

N0 (of not0 of buf0) $\Leftrightarrow$ inter (of buf0) $\Leftrightarrow$ I0 (of not1 of buf0)
 Implies that
 N0 (of not0 of buf0) $\Leftrightarrow$ I0 (of not1 of buf0)

This procedure is called **Elaboration**, in which by applying the transitivity rule in several steps we obtain new connectivity information. Step 3 of the algorithm is the application of

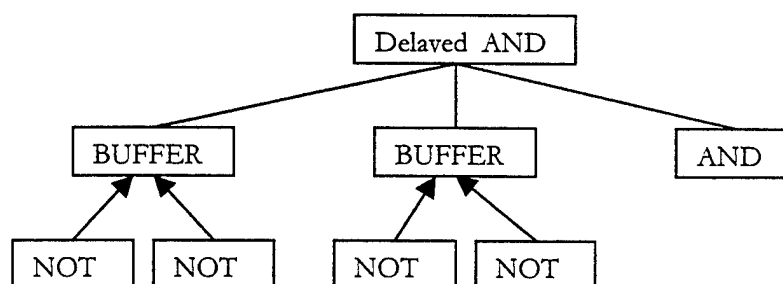


Figure 5.4. Decomposition Hierarchy

elaboration on the design tree to extract all the interconnection information. Before we

get into the details of our elaboration procedure, let's overview our approach. In Figure 5.4, we displayed the tree representation of our example. As we explained earlier, each leaf of this tree represents one basic object. As shown in Figure 5.4, each leaf or basic object, using the transitivity rule, reports to the parent node the connectivity information (which means that each of the basic object input/output ports are connected to which signal in the parent node). After the parent node receives all reports, it does the same procedure and reports the newest elaborated information to its parent node. This process continues till all the information is collected at the root, which is the top most design.

Let's see how this works for our example. For example, consider not0 of buf0. First not0 reports to buf0 that $I0 \Leftrightarrow IN0$ and $N0 \Leftrightarrow inter$. Then after buf0 collects the same information from not1 (which is $I0 \Leftrightarrow inter$ and $N0 \Leftrightarrow B0$), it reports to Delayed_AND that $I0 \Leftrightarrow X0$ and $B0 \Leftrightarrow inter0$. Delayed_AND, after receiving this information and the same kind of information from buf1 and and0, can then simply combine and elaborate this information and form the final interconnection information.

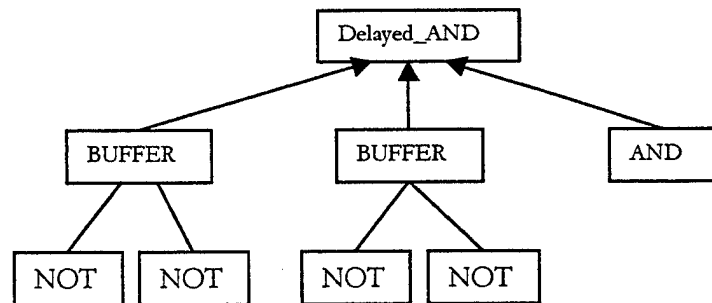
• Phase 1

not0 reports to buf0 :
 buf0: not0: I0 $\Leftrightarrow$ buf0: IN0
 buf0: not0: N0 $\Leftrightarrow$ buf0: inter

not1 reports to buf0 :
 buf0: not1: I0 $\Leftrightarrow$ buf0: inter
 buf0: not1: N0 $\Leftrightarrow$ buf0: B0

not0 reports to buf1 :
 buf1: not0: I0 $\Leftrightarrow$ buf1: IN0
 buf1: not0: N0 $\Leftrightarrow$ buf1: inter

not1 reports to buf1 :
 buf1: not1: I0 $\Leftrightarrow$ buf1: inter
 buf1: not1: N0 $\Leftrightarrow$ buf1: B0



• Phase 2

buf0 reports to Delayed_AND :
 buf0: not0: I0 $\Leftrightarrow$ Delayed_AND: X0

```

buf0:not0:N0 ⇔ buf0:inter
buf0:not1:I0 ⇔ buf0:inter
buf0:not1:N0 ⇔ Delayed_AND:inter0

```

```

buf1 reports to Delayed_AND :
buf1:not0:I0 ⇔ Delayed_AND:X1
buf1:not0:N0 ⇔ buf1:inter
buf1:not1:I0 ⇔ buf1:inter
buf1:not1:N0 ⇔ Delayed_AND:inter1

```

```

and0 reports to Delayed_AND :
and0:I0 ⇔ Delayed_AND:inter0
and0:I1 ⇔ Delayed_AND:inter1
and0:A0 ⇔ Delayed_AND:Y0

```

Note that we have used for example “buf0:inter”, instead of “inter” itself, in order to avoid confusion between inter of buf0 and inter of buf1. This example shows why we need to assign unique object IDs to the basic objects, input/output ports and intermediate signals, so that we can uniquely identify them. The main reason for doing this is because we use “names” for the elaboration process and multiple naming, different entities may occur. For example in Delayed_AND, two signals following elaboration will have same name as “inter”, however these two signals are absolutely different from each other and it is our responsibility to somehow differentiate between them. In our approach by assigning IDs to objects and their input/output ports, we have overcome this problem. To solve the same problem for the intermediate signals, we simply rename them based on a tag ID. At each architecture statement part, we use a unique tag to rename the intermediate signal and when we move to the other architecture statement part of another design unit in the design unit tree, we can simply increment the tag to obtain a unique tag for that design. For example, if we initially set tag variable to 0, in buf0, we use this tag to rename “inter” to “_inter_0” and when we move to buf1, we increment the tag variable to 1, and therefore we rename “inter” located in buf1 to “_inter_1” and we can see how this approach enables us to differentiate between buf0:inter and buf1:inter.

In the following section we will explain the implementation of our algorithm in more detail, based on our example.

5.4 Implementation Overview

We introduce a simple record as a unit of interconnection information and we call this unit a *NetSegment*.

```
Record NetSegment
{
    signal_name;
    objectID;
    portID;
    port_type;
}
```

Here, the `signal_name` is the name of the input/output port ; `objectID` is the object ID of the basic object that has this signal as its input/output port; `portID` is the port ID of this signal within that basic object; and `port_type` is the type of the port, input or output. These are the minimal number of fields we need for elaboration and our real implementation is a more complicated class than a record, with more data members and function members.

For each input/output part of each basic object, we create such a *NetSegment* and fill in the fields. Then elaboration is done based on these *NetSegments*. Each node of the design tree receives a set of *NetSegments* from its children, elaborates them, and then reports the elaborated set to its parent, in the same way we explained in section 5.3. Lets do this in our example and see how it works:

Phase 1

buf0:not0 reports to buf0

Signal_name	I0 $\Leftrightarrow$ IN0
ObjectID	0
PortID	0
Port_type	Input

buf0:not0 reports to buf0

Signal_name	N0 $\Leftrightarrow$ _inter_0
ObjectID	0

PortID	0
Port_type	Output

buf0:not1 reports to buf0

Signal_name	I0⇔_inter_0
ObjectID	1
PortID	0
Port_type	Input

buf0:not1 reports to buf0

Signal_name	N0⇔B0
ObjectID	1
PortID	0
Port_type	Output

buf1:not0 reports to buf1

Signal_name	I0⇔IN0
ObjectID	2
PortID	0
Port_type	Input

buf1:not0 reports to buf1

Signal_name	N0⇔_inter_1
ObjectID	2
PortID	0
Port_type	output

buf1:not0 reports to buf1

Signal_name	I0⇔_inter_1
ObjectID	3
PortID	0
Port_type	input

buf1:not1 reports to buf1

Signal_name	N0⇔B0
ObjectID	3
PortID	0
Port_type	output

Phase 2

buf0:reports to Delayed_AND

Signal_name	I0⇔X0
ObjectID	0

PortID	0
Port_type	input

buf0:reports to Delayed_AND

Signal_name	N0↔_inter_0
ObjectID	0
PortID	0
Port_type	Output

buf0:reports to Delayed_AND

Signal_name	I0↔_inter
ObjectID	1
PortID	0
Port_type	Input

buf0:reports to Delayed_AND

Signal_name	N0↔_inter0_2
ObjectID	1
PortID	0
Port_type	Output

buf1:reports to Delayed_AND

Signal_name	I0↔X1
ObjectID	2
PortID	0
Port_type	Input

buf1:reports to Delayed_AND

Signal_name	N0↔_inter_1
ObjectID	2
PortID	0
Port_type	Output

buf1:reports to Delayed_AND

Signal_name	I0↔_inter_1
ObjectID	3
PortID	0
Port_type	Input

buf1:reports to Delayed_AND

Signal_name	N0↔_inter1
ObjectID	3
PortID	0
Port_type	Output

and0:reports to Delayed_AND

Signal_name	I0⇔_inter0
ObjectID	4
PortID	0
Port_type	Input

and0:reports to Delayed_AND

Signal_name	I1⇔_inter1
ObjectID	4
PortID	1
Port_type	Input

and0:reports to Delayed_AND

Signal_name	A0⇔Y0
ObjectID	4
PortID	0
Port_type	Output

Note that for the signal_name field, instead of an elaborated name, we have used both signal name and elaborated name to make it easier to understand. In the real implementation, we simply substitute the current signal name with the elaborated signal name. In the previous example, the left hand side is the current signal name and the right hand side is the elaborated signal name (for example I0⇔IN0 in the first phase, not0 of buf0 simply substitutes I0 with IN0 in the corresponding NetSegment).

After all the NetSegments are collected at the root, we can simply extract the interconnection information. For example to find the destination list of any output signal with the name xxx, we simply search through all NetSegments, and find those NetSegments labeled with the xxx as the signal name with port_type of input. The search can be done very efficiently, since we can make a sorted list of the NetSegments based on the signal names and for example, apply binary search to locate any specific NetSegment.

6. BENCHMARKS

We have benchmarked the performance of Time Warp and Synchronous simulators on the SP2 and Origin 2000. A maximum speed-up of 31 has been achieved using 64 processors. MPI was used on the SP2 and Origin2000 supercomputers. Benchmark circuits include ISCAS circuits with up to 18,000 objects. We have simulated these circuits with $n=1000$ vectors for 2,000,000 units of time. To measure the performance of each simulator, we have inserted a computational granularity as a parameter. For the s35932x1 and oddeven benchmarks, we have simulated with up to 64 processors, and the remaining circuits, we have tested with up to 16 processors.

Benchmark Circuit Name	Number of Objects	P=1 Execution time	The best execution time with up to P=64 or P=16	Maximum Speed-up	Optimal P
s35932x2	17829	594	36	16	32
oddeven	16650	787	25	31	64
c7552*	3620	315	49	6.3	16
multi32*	6950	4373	326	13.4	16
s15850*	5192	54	8.4	6.3	16

The following figures show the speed-up with s35932X2 with up to 64 processors using Time Warp protocol. As shown in the Figure 6.1, the Time-Warp simulation speed-

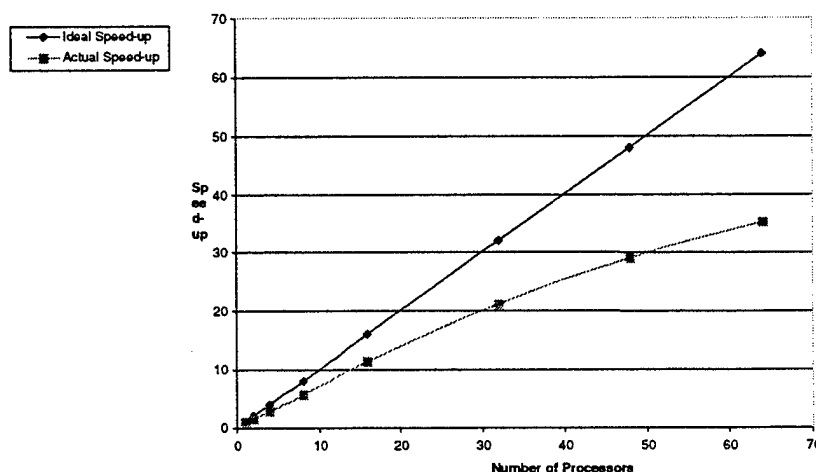


Figure 6.1. Speed-up of s35932x2

up increases as the number of processors increases, but the rate of speed-up slows down as the number of processors increases.

We also ran the parallel simulator on the SP2. Figure 6.2 shows the comparison of execution time on SP2 and Origin 2000 for Time Warp and Synchronous protocols with ISCAS S953. The Origin is approximately twice as fast as the SP2. As shown in the figure, the Time Warp is much slower than the synchronous one with 2 processors, but the Time Warp outperforms the synchronous one as the number of processors increase. As the number of the processor increases, both simulation schemes suffer because of the communication overhead.

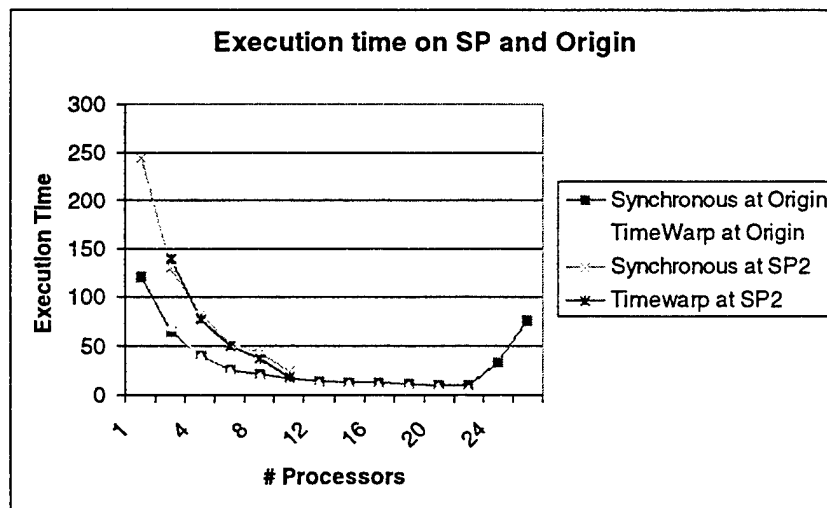


Figure 6.2. Comparison of Time Warp & Synchronous Protocols on SP2 and Origin 2000

Figures 6.3 and 6.4 compare the performance of simulation protocols as the amount of computation increases per event (called computation granularity). The synchronous one is better than Time Warp when the number of processors are small and the amount of computation per event is small. However, as the amount of computation increases, the performance of Time Warp outperforms the synchronous one. We also found that rollback rate 10~20% is acceptable for Time Warp.

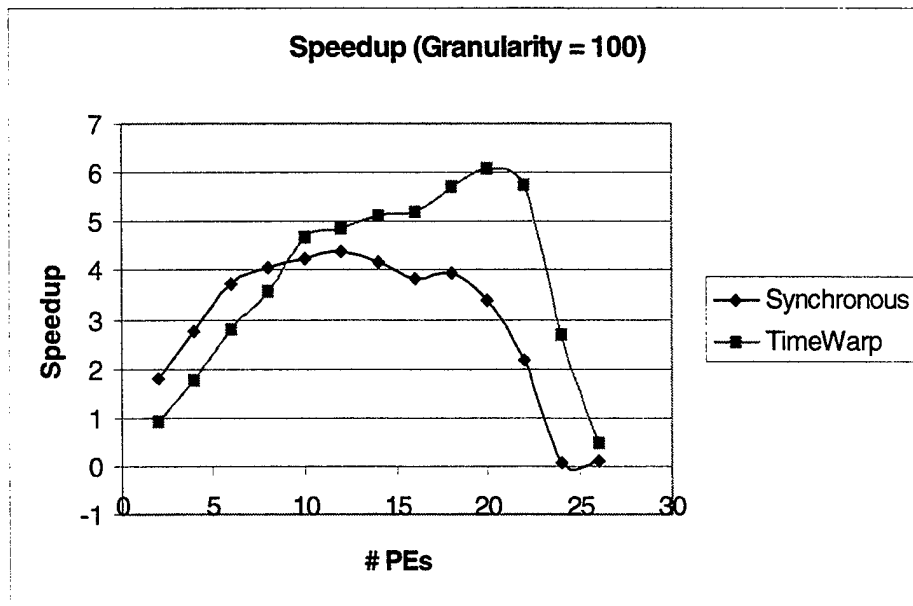


Figure 6.3. Speed-up with ISCAS S953 for Grain size 100.

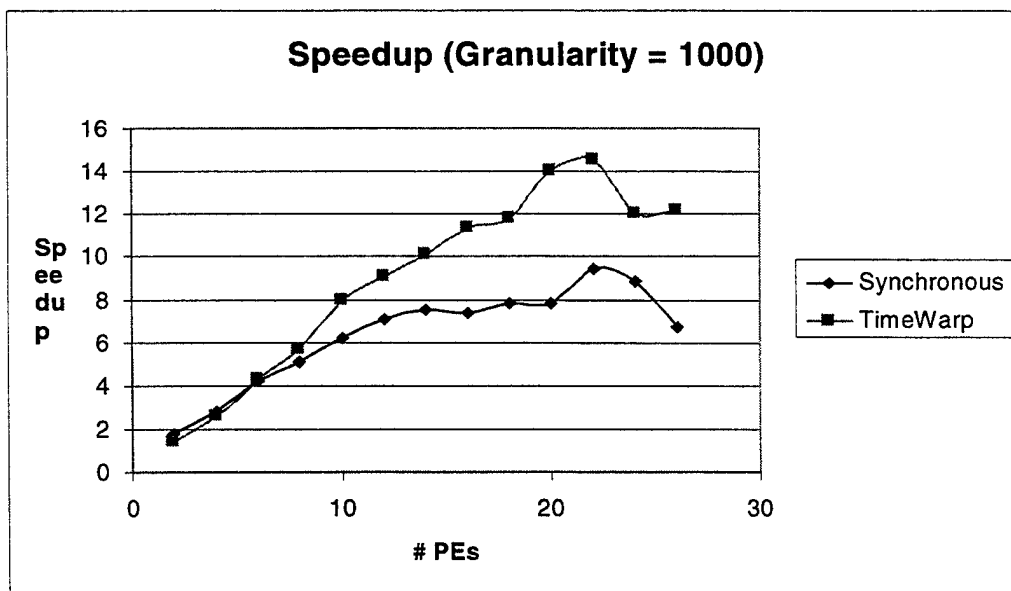


Figure 6.4. Speed-up with ISCAS S953 for Grain size 1000.

The performance of Time Warp is closely related to the amount of computation per event. The following three figures show the performance of Time Warp parallel simulation for various computation grain sizes (per event). Note that for a small grain size, the performance gain is minimal, but as the computation grain size increases, we can get a good speed-up for up to 32 processors. However, the communication cost eventually dominates as the number of processor increases and the speed-up gain drops as the number of processors becomes too large.

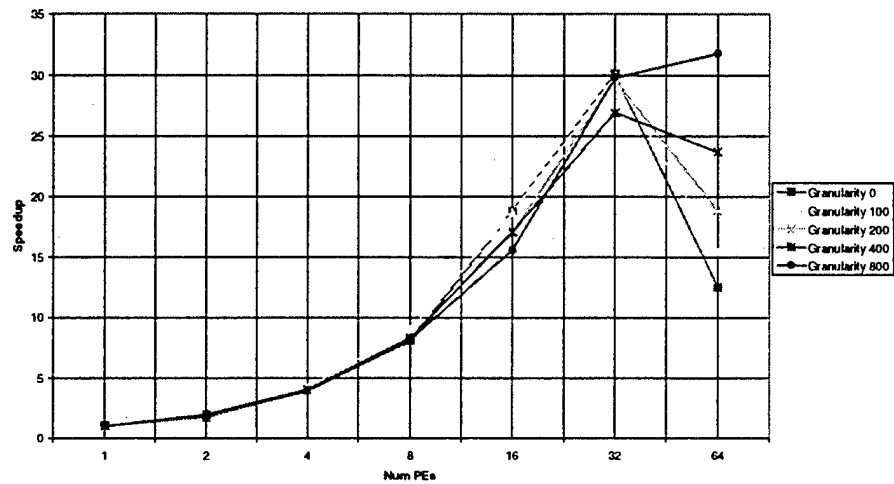


Figure 6.5. S35932x1 Speed-up

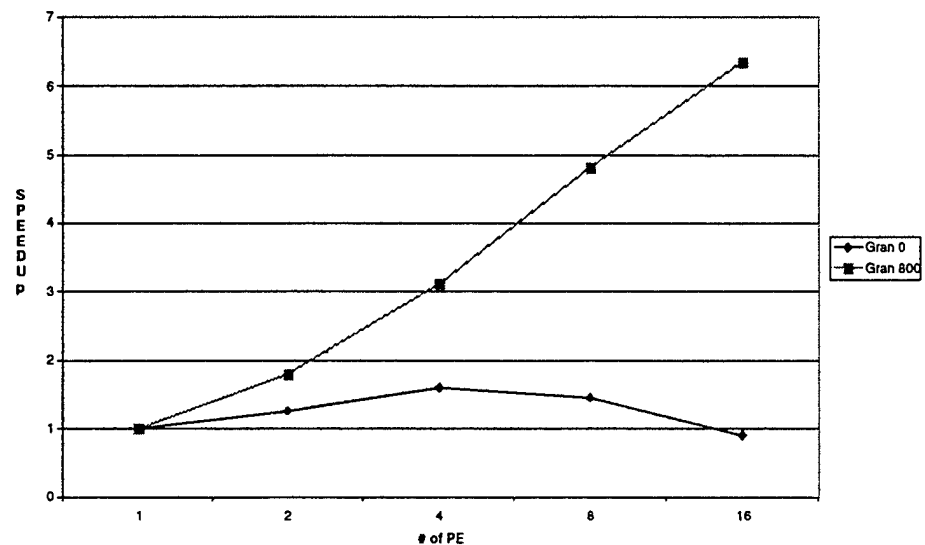


Fig. 6.6 Speed-up of S15850

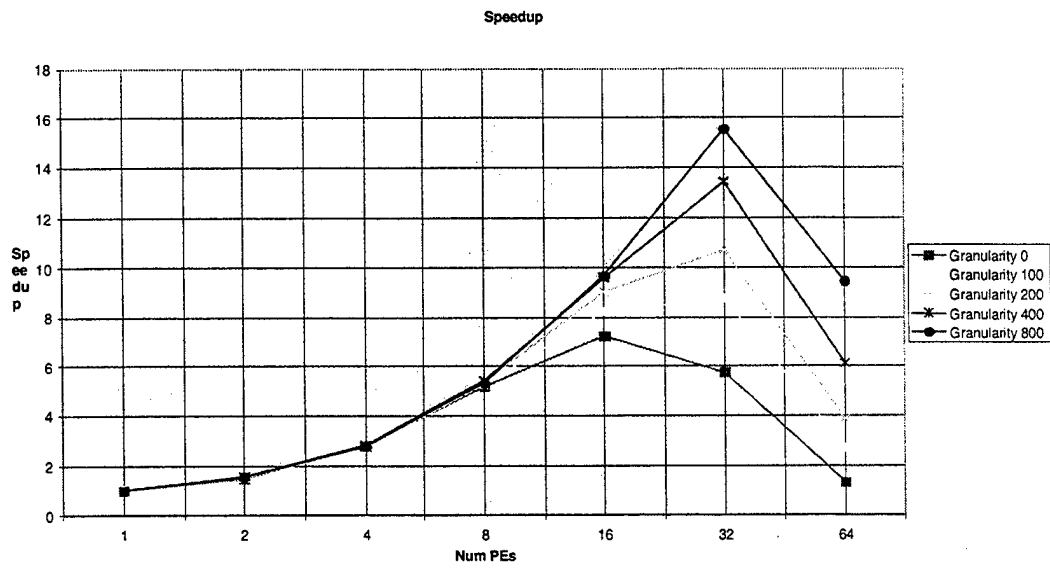


Figure 6.7 Odd-even sorter

7. CONCLUSIONS

We have implemented a parallel VHDL simulation, especially targeted for behavioral level simulation. The developed simulator could improve the speed-up up to 31 times using 64 processors. It has achieved our original goal of improving the speed of VHDL simulation, a bottleneck of microelectronic design.

The speed of a parallel program depends on various factors such as the efficiency of algorithms (or schemes), data structures, communication mechanisms, load balancing, and programming styles. Among those, the communication overhead is one of the most important factors. Thus, the effect of communication latency hiding and overlapping computation and communication was not significant. Also the experimental results show that the computation grain size of each event affects the performance greatly. Particularly, we found out that many VHDL models we have simulated have very small grain size, limiting the speed-up gain.

We have also developed an object modeling technique and a front-end interface for parallel simulation. The front-end interface translates VHDL models into C++ Object models. Our approach is extensible so that the user can mix and match the models developed by domain experts and the simulation scheme developed by parallel programmers. Therefore, our simulation engine can be applied to other areas of discrete event simulation such as the force-simulation and network simulation. The benefit of the object oriented nature of our approach is that by its very design, it is simple to “plug in” a different simulation kernel to get an efficient simulation.

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_Oa;
    SavantbitType in_work_Dand2_Ob;

    // output signals
    SavantbitType out_work_Dand2_Oy;

    // 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_On;

public:
    work_Dand2_Dbehav_of_work_Dand2_class0() :
BasicObject()
    {
```

```

        // register input signals

registerInSignal(&in_work_Dand2_Oa,sizeof(in_work_Dand2_Oa
));

registerInSignal(&in_work_Dand2_Ob,sizeof(in_work_Dand2_Ob
));

        // register output signals

registerOutSignal(&out_work_Dand2_Oy,sizeof(out_work_Dand2
_Oy));

        // register the wait for signals

registerInSignal(&in_wait_for_signal,sizeof(in_wait_for_si
gnal));

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_Oa = X;
in_work_Dand2_Ob = X;

        // output signal initial values
out_work_Dand2_Oy = 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_Oy = gatework_Dand2_Dbehav_On;
    assignDelay(&out_work_Dand2_Oy, DELTA, TRANSPORT);

    // line 12 "and.tex"
    P++;
BLOCK1:
    if( !hasEvent(&in_work_Dand2_Oa) &&
        !hasEvent(&in_work_Dand2_Ob) ) return;

    // line 13 "and.tex"
    gatework_Dand2_Dbehav_On = (in_work_Dand2_Oa &
in_work_Dand2_Ob);

    // 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(Pvhd11DArray<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 == (Pvhd11DArray<Dtype> &obj) {
        for(int i=0;i<size;i++)
            if(array[i] !=          obj.array[i]) return 0;
    }

```

```

        return 1;
    }

    // overload !=
    int operator != (Pvhd11DArray<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 =
    Pvhd11DArray<Dtype>& operator=(Pvhd11DArray<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
    Pvhd11DArray<Dtype>& operator=(int value) {
        for( int i=0; i<size; i++)
            array[i] = value;
        return *this;
    }

    // overload +
    Pvhd11DArray<Dtype>& operator+(Pvhd11DArray<Dtype> &obj)
    {
        Pvhd11DArray<Dtype> *temp = new Pvhd11DArray<Dtype>;
        temp->allocateArray(obj.size);

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

```

```

    return *temp;
}

// overload -
Pvhd11DArray<Dtype>& operator-(Pvhd11DArray<Dtype> &obj)
{
    Pvhd11DArray<Dtype> *temp = new Pvhd11DArray<Dtype>;
    temp->allocateArray(obj.size);

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

// overload *
Pvhd11DArray<Dtype>& operator*(Pvhd11DArray<Dtype> &obj)
{
    Pvhd11DArray<Dtype> *temp = new Pvhd11DArray<Dtype>;
    temp->allocateArray(obj.size);

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

// overload /
Pvhd11DArray<Dtype>& operator/(Pvhd11DArray<Dtype> &obj)
{
    Pvhd11DArray<Dtype> *temp = new Pvhd11DArray<Dtype>;
    temp->allocateArray(obj.size);

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

// overload &
Pvhd11DArray<Dtype>& operator&(Pvhd11DArray<Dtype> &obj)
{
    Pvhd11DArray<Dtype> *temp = new Pvhd11DArray<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

APPENDIX H

INSTRUCTIONS AND MANUALS

FRONT END INTERFACE

To run the system, follow these steps :

0) **cd ~pvhdl/savant/src/aire/iir/TEST**. This is the main directory for generating files.

1) Type and enter :

interface <Design File> <Entity Name> <Architecture Name> <Number of events> <Clock Width> <Number of processors>

This will create a directory with the name <Design File>.FILES, which includes all the generated files.

2) **cd <Design File>.SIMULATION_FILES**

There are three subdirectories: simulation_files, DEBUG and WORK. **We just need the files in simulation_files** and the other two directories are for debugging purposes.

simulation_files directory, includes all the files that are used by the simulator. These files are :

- <Entity Name>_<Architecture_Name>.net
- <Entity Name>_<Architecture_Name>.primary_inputs
- <Entity Name>_<Architecture_Name>.primary_outputs
- <Entity Name>_<Architecture_Name>.partition. <Number of processors>
- <Entity Name>_<Architecture_Name>.vec. <Number of events>
- Classes.h
- InstantiateObject.cpp

3) **cd simulation_files**

4) copy files to the simulator's directory.

Since <Entity Name>_<Architecture_Name>.net is independent of partition

information or test vector size, the following utilities can be used to generate a new partition
(new <Entity Name>_<Architecture_Name>.partition. > <Number of processors>
file)
or a new test vector (in <Entity Name>_<Architecture_Name>.vec. <Number of
events>).

To generate a new partition, follow these steps :

1) **cd ~pvhdl/savant/src/aire/iir/TEST**

2) Type and enter :

**random_partition <Design File> <Entity Name> <Architecture Name>
<Number of processors>**

2) **cd <Design File>.FILES**

3) **cd simulation_files**

4) copy <Entity Name>_<Architecture_Name>.partition. <Number of
processors>
to the simulator's directory.

To generate a new test vector, follow these steps :

1) **cd ~pvhdl/savant/src/aire/iir/TEST**

2) Type and enter :

**generate_vector <Design File> <Entity Name> <Architecture Name>
<Number of events> <Clock Width>**

2) **cd <Design File>.FILES**

3) **cd simulation_files**

4) copy <Entity Name>_<Architecture_Name>.vec. <Number of events>
to the simulator's directory.

Example :

Suppose that the design file is **shifter.vhd** , with the top most design unit **entity shifter**

and architecture structural :

Design file : shifter.vhd
Top most entity name : shifter
Top most architecture name : structural
User specified number of events per primary input : 400
User specified clock width : 200
Number of processors for this simulation : 4

Then we follow the direction :

```
$ cd ~pvhdl/savant/src/aire/iir/TEST
$ interface shifter.vhd shifter structural 400 200 4
```

This will create a directory
~pvhdl/savant/src/aire/iir/TEST/shifter.vhd.SIMULATION_FILES

```
$ cd shifter.vhd.SIMULATION_FILES
$ cd simulation_files
$ls
```

will show :

```
shifter_structural.net
shifter_structural.primary_inputs
shifter_structural.primary_outputs
shifter_structural.partition.4
shifter_structural.vec.400
Classes.h
InstantiateObject.cpp
$ cp * <SIMULATION_DIRECTORY>.
```

Now, suppose that you want to generate a new partition with 8 processors :

```
$ cd ~pvhdl/savant/src/aire/iir/TEST
$ random_partition shifter.vhd shifter structural 4
$ cd shifter.vhd.SIMULATION_FILES
$ cd simulation_files
$ cp shifter_structural.partition.8 <SIMULATION_DIRECTORY>.
```

Or you may want to generate a test vector with larger size :

```
$ cd ~pvhdl/savant/src/aire/iir/TEST
$ generate_vector shifter.vhd shifter structural 600 100
$ cd shifter.vhd.SIMULATION_FILES
$ cd simulation_files
$ cp shifter_structural.vec.600 <SIMULATION_DIRECTORY>.
```

Description :

interface <Design File> <Entity Name> <Architecture Name> <Number of events> <Clock Width> <Number of processors>

It is a C shell script, which glues different files that are created by scram and other utilities. It creates a subdirectory with the name <Design File>.SIMULATION_FILES, which includes all the generated files. It also creates three subdirectories under the <Design File>.SIMULATION_FILES. These directories are

1) **simulation_files**, which includes all the files which are plugged into simulator. The following files are created in this directory :

- **<Entity Name>_<Architecture Name>.net**

Includes the interconnection information of the design (generated by scram).

- **<Entity Name>_<Architecture Name>.primary_inputs**

Includes information about the primary inputs of the design (generated by scram).

- **<Entity Name>_<Architecture Name>.primary_outputs**

Includes information about the primary outputs of the design (generated by scram).

- **<Entity Name>_<Architecture Name>.partition. <Number of processors>**

Includes the partition information of the design (not generated by scram).

- **<Entity Name>_<Architecture Name>.vec. <Number of events>**

Includes test bench for this design (not generated by scram).

- **Classes.h**

Includes C++ classes generated by scram.

- **InstantiateObject.cpp**

Includes initialization code for the simulator.

2) **DEBUG**, which includes additional files useful for debugging. . The following files are created in this directory :

- **<Entity Name>_<Architecture Name>.net.debug**

Includes the debugging interconnection information of the design (generated by scram).

- **<Entity Name>_<Architecture_Name>.net.seg**

Includes the interconnection information of the design (generated by scram).

- **<Entity Name>_<Architecture_Name>.primary_inputs.debug**

Includes debugging information about the primary inputs of the design (generated by scram).

- **<Entity Name>_<Architecture_Name>.primary_outputs.debug**

Includes debugging information about the primary outputs of the design (generated by scram).

3) **WORK**, which is a working directory for the scripts. Basically it contains the union of the previous two directories.

This script invokes scram and three other perl scripts :

(1) **partition.prl**, which generates <Entity Name>_<Architecture_Name>.partition.<Number of processors> file.

(2) **test_vector.prl**, which generates <Entity Name>_<Architecture_Name>.vec.<Number of events> file.

(3) **instantiation.prl**, which generates InstantiationObject.cpp

PARALLEL SIMULATION

Parallel Simulation

After generating C++ models, they can be simulated by the simulation kernel. The steps are:

- Test Vector Generation
- Partitioning
- Compiling

There are three different simulation kernels developed, sequential, synchronous, and Time Warp. The following section describes the details how C++ models can be simulated. To simulate the model in parallel, the user must specify the simulation kernel, the test vector size, the duration of simulation cycle, partitioning scheme, among others. To simplify these steps, we have developed scripts, and the following shows step by step how users can run a parallel simulation.

Directory structure

To simplify the running on HPC platforms, we have developed a *script* that works under two parallel computers, Origin 2000 and IBM SP2. They share the source code in the system but their binary code is different. There are two subdirectories. Subdirectory "o2k" contains the files for the Origin 2000 and subdirectory "sp2" contains the files for the IBM SP2.

"src" directory

The "src" directory contains all the source code for the simulation kernels and partitioning program. There are three simulation kernels. They are "sequential", "synchronous" and "timewarp" separately. "partition" subdirectory contains the source code for three different ways of partitioning. They include random partition, random partition with duplication and level partition.

"bin" directory

The "bin" directory contains all the executables. "sequential" subdirectory includes the make file and simulation file. Because there is a difference in the PBS format for Origin 2000 and IBM SP2, there were two subdirectories for "synchronous" and "timewarp" directory. Each of them contains a make script and a simulation script both in batch and interactive mode. For detailed information, please see the README file in the directory or check the complete documentation.

The "partition" directory contains executables for random partition, random partition with duplication and level partition. "util" directory contains utilities like checking the correctness of data file and checking the output result of simulation. Please see the README file of that directory of the complete documentation.

"data" directory

The "data" directory contains all the data generated by the interface program. The interface program generates (from the VHDL file) the C++ model and elaboration information. Each directory contains the files for one VHDL source file. The name of the directory must conform to the format of "entity_architecture", otherwise it can not be correctly simulated.

"result" directory

The "result" directory contains the output results of a simulation run by batch mode. The simulation kernel can be either synchronous or timewarp. Depending on the simulation kernel, parallel computer (o2k, sp2) and the data, a corresponding directory will be created if it does not exist. For example, if "oddeven_structure" is run with the timewarp kernel on the sp2, directory "result/timewarp/sp2/oddeven_structure" will be created. Inside the directory are all the results that ran with these parameters. If different numbers of processors are used in simulation, the file name will be different in that directory. The number of processors can identify them.

The following figure shows the directory structure.

```

PVHDL ---- src ---- sequential
          |
          |---- synchronous
          |
          |---- timewarp
          |
          |---- partition

          ---- bin ---- sequential
                |
                |---- synchronous ---- o2k
                |                      |
                |                      |---- sp2
                |
                |---- timewarp ---- o2k
                |                    |
                |                    |---- sp2
                |
                |---- partition ---- o2k
                |                    |
                |                    |---- sp2
                |
                |---- util

          ---- data ---- entity_architecture_1
                |
                |---- ...
                |
                |---- entity_architecture_k

          ---- result ---- synchronous ---- o2k ---- entity_architecture_1
                |                      |          |
                |                      |          |---- ...
                |                      |          |---- entity_architecture_k
                |                      |
                |                      |---- sp2 ---- entity_architecture_1
                |                      |          |
                |                      |          |---- ...
                |                      |          |---- entity_architecture_k

                |---- timewarp ---- o2k ---- entity_architecture_1
                |                    |          |
                |                    |          |---- ...
                |                    |          |---- entity_architecture_k
                |                    |
                |                    |---- sp2 ---- entity_architecture_1
                |                    |          |
                |                    |          |---- ...
                |                    |          |---- entity_architecture_k

```

Compiling, Partitioning and Running

To compile and run a program, go to the "bin" directory. For the synchronous simulation kernel, go to "bin/synchronous". You will find two subdirectories under "bin/synchronous". One is "o2k", the other is "sp2". Directory "o2k" contains executables for SGI Origin 2000 and directory "sp2" contains executables for IBM SP2. To get detailed information for compiling and running, check the README file of these two directories.

Similarly, "bin/timewarp" contains the timewarp simulation kernel and "bin/sequential" contains the sequential simulation kernel.

Compiling

Before simulation, the C++ model has to be compiled and linked with different simulation kernels. There are several shell scripts that do this task.

Sequential Simulation

Directory: bin/sequential/o2k or bin/sequential/sp2

Command: seqmake <entity_arch>

Example: seqmake s35932x1_structural

Note: All the files below must be present in data/s35932x1_structural directory.

Classes.h

InstantiateObject.cpp

Synchronous Simulation

Directory: bin/synchronous/o2k or bin/synchronous/sp2

Command: syncmake <entity_arch>

Example: syncmake s35932x1_structural

Timewarp Simulation

Directory: bin/timewarp/o2k or bin/timewarp/sp2

Command: twmake <entity_arch>

Example: twmake s35932x1_structural

Partitioning

To simulate in parallel, it is necessary to partition the data statically for a particular number of processors you want to simulate the C++ model with. The system provides three different partitioning algorithms: random partition, random with duplication and level partitioning.

Random Partition:

Circuit graphs are partitioned in random assignment.

It is well known that random partition works well in most cases.

Random with duplication:

Certain gates that have high out-degree are duplicated to reduce the communication cost.

Level Partitioning:

Gates that are in the same level are grouped together. This partitioning scheme works well in the synchronous protocol.

Good partitioning can improve the performance of simulation. Depending on the C++ model, one partitioning algorithm may perform better than the other ones. Synchronous simulation works only with random partitioning. Timewarp simulation works with all the partitioning algorithms.

Random Partitioning

Directory: bin/partition/o2k or bin/partition/sp2

Command: RP <entity_arch> <Num Partition>

Example: RP s35932x1_structural 8

Note: The command in the example creates partitions from 1 up to 8 processors for the model s35932x1_structural. The command requires that file "s35932x1_structural.net" has to be present in the directory "data/s35932x1_structural". After the successful execution, files "s35932x1_structural.partition.1" to "s35932x1_structural.partition.8" will be generated in the directory "s35932x1_structural".

Random partitioning with duplication

Directory: bin/partition/o2k or bin/partition/sp2

Command: RP_Dup <entity_arch> <Num Partition>

Example: RP_Dup s35932x1_structural 8

Note: The command in the example creates partitions from 1 up to 8 processors for the model s35932x1_structural. The command requires that file "s35932x1_structural.net" has to be present in the directory "data/s35932x1_structural". After the successful execution, files "s35932x1_structural.partition.1.Dup" to "s35932x1_structural.partition.8.Dup" will be generated in the directory "s35932x1_structural".

Level Partition with duplication

Directory: bin/partition/o2k or bin/partition/sp2

Command: LP_Dup <entity_arch> <Num Partition>

Example: LP_Dup s35932x1_structural 8

Note: The command in the example creates partitions from 1 up to 8 processors for the model s35932x1_structural. The command requires that file "s35932x1_structural.net" has to be present in the directory "data/s35932x1_structural". After the successful execution, files "s35932x1_structural.partition.1.Level" to "s35932x1_structural.partition.8.Level" will be generated in the directory "s35932x1_structural".

Running the simulation

After compiling and partitioning, you are ready to run the simulation. We provided the interactive and batch scripts.

Sequential Simulation

Directory: bin/sequential/o2k or bin/sequential/sp2
Command: `run_i <entity_arch> <VecSize> <MAXGVT> [PRINT]`

Parameters:

- <entity_arch>: the model to simulate
- <VecSize>: the size of test vector
- <MAXGVT>: the time limit of simulation
- [PRINT]: 0 - Do not print events (Fastest)
 - 1 - only print events on primary output
 - 2 - print all the events

Example: `run_i s35932x1_structural 1000 10000 0`
The command above will simulate model "s35932x1_structural"
With 1000 test vectors and time limit 10000. This simulation does not print any events.

The following files have to be in directory "data/s35932x1_structural"

s35932x1_structural.net
s35932x1_structural.partition.1
s35932x1_structural.primary_inputs
s35932x1_structural.primary_outputs
s35932x1_structural.vec.1000

Synchronous Simulation

Directory: bin/synchronous/o2k or bin/synchronous/sp2
Command:

Interactive : `run_i <entity_arch> <NumProcessor> <VecSize> <MAXGVT> [PRINT]`

Batch: `run_batch <entity_arch> <NumProcessor> <VecSize> <MAXGVT> [Granularity]`
[PRINT]

Parameters:

- <entity_arch>: the model to simulate
- <NumProcessor>: Number of processors to use during the simulation
- <VecSize>: the size of test vector
- <MAXGVT>: the time limit of simulation
- [PRINT]: 0 - Do not print events (Fastest)
 - 1 - only print events on primary output
 - 2 - print all the events

Example: `run_batch s35932x1_structural 8 1000 10000 0 0`

The command above will simulate model "s35932x1_structural" with 1000 test vectors and time limit 10000. This simulation does not print any event. The number of processors used during the simulation is 8.

The following files have to be in directory "data/s35932x1_structural"

s35932x1_structural.net
s35932x1_structural.partition.8
s35932x1_structural.primary_inputs
s35932x1_structural.primary_outputs
s35932x1_structural.vec.1000

Timewarp Simulation

Directory: bin/timewarp/o2k or bin/timewarp/sp2
Command:

Interactive: run_i <entity_arch> <NumProcessor> <VecSize> <MAXGVT> <Partition>
<Schedule> [PRINT] [GVTWINDOW] [MSGCHECK]

Batch: run_batch <entity_arch> <NumProcessor> <VecSize> <MAXGVT> <Partition>
<Schedule> [Granularity] [PRINT] [GVTWINDOW] [MSGCHECK]

Parameters:

<entity_arch>: the model to simulate
<NumProcessor>: Number of processors to use during the simulation
<VecSize>: the size of test vector
<MAXGVT>: the time limit of simulation
<Partition>: 0 - Random Partition
 1 - Random Partition with Duplication
 2 - Level Partition
<Schedule>: Number of objects active per communication cycle
[Granularity] default=0
[PRINT]: 0 - Do not print events (Fastest)
 1 - only print events on primary output
 2 - print all the events
[GVTWINDOW]: GVT window size. Default is MAXGVT.
[MSGCHK]: Number of objects active before checking message. Default is <Schedule>

Example: run_batch s35932x1_structural 8 1000 100000 1 50 0 0

The command above will simulate model "s35932x1_structural" With 1000 test vectors and time limit 100000. This simulation does not print any events. The number of processors used during the simulation is 8 and in each communication cycle 50 objects are active.

The following files have to be in directory "data/s35932x1_structural"

s35932x1_structural.net
s35932x1_structural.partition.8.Dup
s35932x1_structural.primary_inputs
s35932x1_structural.primary_outputs
s35932x1_structural.vec.1000

REFERENCES

1. Peter J. Ashenden, "The Designer's Guide to VHDL", Morgan Kaufmann, San Francisco, 1996.
2. R. Baldwin, M.J. Chung and Y. Chung, "Overlapping Window Algorithm for Computing GVT in Time Warp," Proc. 1991 International Conference on Distributed Computing Systems, pp. 534-541. Also to appear Parallel Programming and Applications.
3. Rajive L. Bagrodia, Designing Efficient Simulations Using Maisie, UCLA Technical Report, 1997.
4. A. Cabrera, M.J. Chung and Yunmo Chung, "A parallel VHDL Simulator on the Connection Machine," Proc. of VHDL spring 92 Conf. pp.83-94.
5. 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
6. Moon Jung Chung and Jiashen Zhou, "Version Control and Configuration Management in Simulation," Technical Report, Michigan State University, 1998.
7. M.J. Chung and Y. Chung, "Performance Prediction to Gate to Processor Ratio," Proc. 1992 International Conference on Parallel Processing, pp. 246-253.
8. M.J. Chung and Y.M. Chung, "An Experimental Analysis of Simulation Clock Advancement in Parallel Logic Simulation on a SIMD Machine," Advances in Parallel and Distributed Simulation, Vol. 23, No. pp. 125-133.
9. M.J. Chung and Y. Chung, "Efficient Parallel Logic Simulation Techniques for

- the Connection Machine," Proc. of Supercomputing'90, pp. 606-614.
10. M.J. Chung and Y. Chung, "Data Parallel Logic Simulation using Time Warp on the Connection Machine," 1989 Design Automation Conference, pp. 98-103.
 11. James P. Cohoon and Jack W. Davidson, "C++ Program Design - An Introduction to Programming and Object-Oriented Design", Times Mirror Higher Education Group, 1997.
 12. R. M. Cubert, and P. A. Fishwick. MOOSE: An Object-Oriented Multimodeling and Simulation Application Framework Submitted to Simulation, June 1997.
 13. J. Engelsma, Y. Chung and M.J. Chung, "Distributed Token-Driven Logic Simulation on a shared-memory multiprocessor," Proc. 6th Workshop in Parallel and Distributed Simulation, 1992, pages 197-198.
 14. Fishwick, P. A. A Visual Object-Oriented Multimodeling Design Approach for Physical Modeling Revision of tr96-026 send to ACM Transactions on Modeling and Computer Simulation, April 1997.
 15. D.R. Jefferson, "Virtual Time", ACM Trans. Programming Languages and Systems, 1985, July, pp. 404-425.
 16. Venkatram Krishnaswamy and Prithviraj Banerjee, "Actor Based Parallel VHDL Simulation Using Time Warp, to appear in Proceedings of the 1996 Workshop on Parallel and Distributed Simulation, Philadelphia, PA, May 1996.
 17. K. Lee and P.A. Fishwick. 1996. Dynamic Model Abstraction, In 1996 Winter Simulation Conference, December, San Diego, CA, pp. 764-771.
 18. Y. Lin and E.D. Lazowaska and M.L. Bailey, "Comparing Synchronization Protocols for Parallel Logic-Level Simulation", Proc. of the 1990 International Conference on Parallel Processing, pp. 223-227.

19. J. Misra, 1995, Distributed discrete-event simulation, *Computing Surveys*, 18(1).
20. David Perllerin and Douglas Taylor, "VHDL Made Easy", Printice Hall, Upper Saddle River, 1997.
21. Bruno R. Preiss, The YADDES Distributed Discrete Event Simulation Specification Language and Execution Environments, 1989 SCS Multiconference on Distributed Simulation, pp. 139-144.
22. R. Radhakrishnan, T. J. McBrayer, K. Subramani, M. Chetlur, V. Balakrishnan, and P. A. Wilsey, "A Comparative Analysis of Various Time Warp Algorithms Implemented in the WARPED Simulation Kernel," *Proceedings of the 29th Annual Simulation Symposium*, 107-116, March 1996.
23. Hassen Rajaei, SIMA: An environment for parallel discrete-event simulation, *The 25th Annual Simulation Symposium*, pp. 147-149. April 1992.
24. Khashayar Rohanimanesh, "Generating Map File by SAVANT", Technical Document of Parallel VHDL Simulation Project, Department of Computer Science, Michigan State University, July, 1998.
25. Jeffrey S. Steinman, Scalable Parallel and Distributed Military Simulations using the Speedes Framework, 1996.
26. 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.
27. 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.
28. Mai Yang, Department of Computer Science, Michignan State University. IIRPvhdl Layer Source Code. [Online] Available <telnet://chung-pvhdl.cps.msu.edu>,

- /home/savant/savant/src/aire/iir/IIRPvhdl, July, 1998.
29. Zeigler, Bernard P.: "Object-Oriented Simulation with Hierarchical, Modular Model", Academic Press, 1990.
 30. 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.
 31. Dept. of ECECS, University of Cincinnati. SAVANT: An Extensible Intermediate for VHDL. [Online] Available <http://www.eecs.uc.edu/~paw/savant/index.html>, July 11, 1998.
 32. Department of ECECS, University of Cincinnati. TyVIS: A VHDL Simulation Kernel. [Online] Available <http://www.eecs.uc.edu/~paw/tyvis/index.html>, May 25, 1998.
 33. Department of ECECS, University of Cincinnati. AIRE Home Page - Advanced Intermediate Representation with Extensibility (AIRE). [Online] Available <http://www.eecs.uc.edu/~paw/aire/index.html>.
 34. Dept. of ECECS, University of Cincinnati. SAVANT Programmer's Manual. [Online] Available <http://www.eecs.uc.edu/~paw/savant/doc/programmers/index.html>, Aug. 25, 1997.