GENISYS: A Component Inversion Engine

by

Kalpesh Zinjuwadia

B.E., Electronics Engineering. Faculty of Technology and Engineering

Maharaja Sayajirao University of Baroda. Baroda, India.

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

_____.

Dr.Perry Alexander (Chair Person)

_____.

Dr.David Andrews (Member)

_____.

Dr.Arvin Agah (Member)

_____.

Date Accepted

Acknowledgments

I would like to express my gratitude to my advisor and committee chairman, Dr. Perry Alexander, for his continuous guidance and motivation throughout this project. I am thankful for his immense support, and useful suggestions whenever I needed them. I also thank him for making my thesis work full of fun. It has been an unforgettable experience working for him. He is more than a mentor to me. I would like to thank Dr. David Andrews and Dr. Arvin Agah for consenting to be on my committee.

I wish to thank Krishna Ranganathan, EDAptive Computing Inc., for his insightful suggestions during the course of my research work. Discussions with him really helped in making my thesis a success. I would also like to thank all my colleagues and members of the SLDG group: Garrin Kimmell, Ed Komp, Cindy Kong, Brandon Morel, Murthy Kakarlamudi, Jesse Stanley, Justin Ward, and Jennifer Streb for making my research experience enjoyable and less stressful. I am thankful to EDAptive Computing Inc. for sponsoring this research project.

Thanks to all my friends who have made my stay in Lawrence a memorable one. Also, thanks to the professors, students, and staff members at the Information and Telecommunication Technology Center and at the University of Kansas for making my two year stay a wonderful experience.

Finally, this work wouldn't be complete without the continuous support and inspiration I got from my family. They have given me so much love and encouragement to which I am overwhelmingly thankful. I owe them a lot.

Abstract

A structural component can be represented as a black-box system having inner-components dependent on each other. This inter-dependence results into dependence hierarchy in the system. If a Component Under Test (CUT) is part of such a system, the relation between CUT and the system input has to be determined to test it. One of the techniques is to invert all the components between CUT and the system input and derive the system input that will generate the desired input for CUT. Component inversion allows the tester to retrieve the initial state the component was in before the forward execution. A component may perform various operations that manipulate the input parameters and generate the output parameters. Component inversion inverts all the operations in order from last to first to determine the input parameters. This thesis presents a component inversion engine - GENISYS: Automated Test GENeration in Intelligent SYStem - that inverts a component to determine input parameters for known output values. GENISYS inverts all the inner-components along the data-paths and retrieves the system input of the structural component. There are two phases involved in this process. First phase determines the *component dependence hierarchy*. During the second phase, GENISYS inverts the components along the hierarchy and generates the *system vectors*. zChaff SAT solver is used to invert expressions in a given component. Various XML files comprise the output of the GENISYS tool. These files contain information about the component dependence hierarchy and the interface parameters of the structural component.

to my family

Contents

List of Figures

Chapter 1

Introduction

Testing is among the most important phases in the software product development cycle. The importance of software testing and its implications with respect to software quality is boundless [29] and is a critical factor in software quality assurance. It represents the final review of a system specification, design and code generation [26]. Software is employed in most of the contemporary applications such as maintaining accounts in a local branch of some firm; electronic banking; and controlling valuable machinery. Testing accounts for as much as two-thirds of the total cost of software product development [25]. Well planned and thorough testing is critical given the costs associated with software failures and the importance of software as a system element. At times, software failures can be disastrous and may lead to unexpected situations. Cosidering these facts, it is critical that software being deployed is thoroughly tested.

The major goal of a tester is to find errors in the software product. It is impor-

tant that the tester finds as many errors in the software as possible by executing various test cases over the software. A good test case has a high probability of finding an error. The more likely a test case is to find errors, the more it reduces testing efforts. Hence, efficient testing reduces the testing effort and cost.

There are various forms of testing [16]. *Structural testing* is carried out using implementation-based testing techniques. Since this type of testing technique deals with the interals of the Implementation Under Test (IUT), it is also known as *white-box testing*. *Functional testing* is based upon systems level requirements. It is used to find whether valid inputs are accepted and the obtained outputs conform to the requirements. This testing technique is also called *black-box testing* as it is performed at a higher level of abstraction without worrying about inner details of the system.

The major drawback of implementation-based testing technique is that while it may help verify the implementation is correct, it does not ensure that the correct system has been implemented. This can be overcome by using specification-based testing techniques [26, 30, 16]. Such techniques use conventional testing methods where the IUT is repeatedly stimulated and outputs generated are compared against expected values derived from specifications. In specification-based testing, the test cases are derived from the specifications. Various advantages of specification-based testing include:

- It not only tests the functionality of the IUT, but also the intended behavior of IUT. It makes sure that correct system has been implemented.

- It exposes any inconsistencies or ambiguities in the specifications. Specification-based testing reveals any glitches in the specification before the implementation begins and hence saves the time and effort required in revisiting the specifications after the implementation is complete.

- Test cases can be designed as soon as the specifications are complete. This speeds up the development process of the software product.

- Modifications in the specifications can be done with respect to user interface.

1.1 Motivation

Various technologies for testing include, conformance testing, functional testing, load testing, performance testing, regression testing, unit testing, and stress testing. Various tools have been developed to perform each specific type of testing. Tools like, test case generators, are available to aid the testing process in general. However, the execution cannot be reversed or undone. Inverting, or undoing, a program or a statement in the program allows the user to execute a particular statement multiple times. This feature is helpful while debugging a software program and speeds up the testing process.

A debugger helps in detecting errors in the product through controlled execution of the program. Many contemporary programming languages have their debuggers. Conventional debuggers include: *GNU Project Debugger*, or *GDB* [10], for C, C++, Pascal, Objective-C, and many other languages, *Java Debugger*, or

JDB [8], for Java programming language, and *Insight* [7], a GUI-based GDB, written in Tcl/Tk.

Conventional debuggers offer various features to help the testing process. These features include [19]: allowing the user to set breakpoints; setting watchpoints; setting control parameters; and examining variable values. At times, the debugger over-steps a statement in the program where the error occurs. It would be useful if it were possible to go back and examine the program states at previous statements that have already been executed. Unfortunately, this is not possible in conventional debuggers [19]. They cannot restore execution state to a previous point. The only option the user has is to restart the program execution. For small-sized program this is not a major drawback. However, for large and computation-intensive programs, it is costly to restart the execution after a long debugging session. To be able to invert program statements and retrieve the initial state is very useful while debugging a program. Program inversion would allows the user to analyze the initial state of the program and makes the debugging process more efficient and faster.

These examples suggest interesting yet challenging notion of *statement inversion*. Statement inversion is the process of determining the possible state of the system that will produce the given state after executing the statement. Semantically, the inversion of a statement is equivalent to the construction of the weakest precondition of a statement from the strongest postcondition [20]. Should statement inversion be possible, the debugger only needs to invert the previous state-

ments to reach the faulty statement rather than re-executing the whole process. *Program inversion* is the process of inverting all the statements in a program in order from the last statement to the first statement resulting into the state before the program execution. This ability to restore the initial state of a system makes program inversion important and an integral part of testing. Program inversion makes the testing process complete and more robust.

1.2 Problem Statement

Program inversion is an important aspect of debugging that not only speeds up the testing process, but also makes it more effective. A software product can be represented as a structural model that takes some input and generates corresponding output. This component may contain a group of inter-dependent inner components, where each inner component drives some other inner component. If the tester wishes to test one of the inner components of the system, they need information regarding the input values to the inner Test Component (TC). The structural component is a black-box system for the tester, that hides the inner details of the system. In such a case, the tester needs to figure out the inner components driving and driven by the TC. Given the input and output parameter values of the TC, the tester needs to traverse through the list of driving and driven components and invert them to determine the system input and output that generates these values. When fed to the structural component, these system inputs should produce the same system output as obtained initially. This

thesis is an effort to solve the problem of component inversion in component inter-dependence scenarios.

1.3 Proposed Solution

GENISYS, a component inversion engine, is a tool developed to solve the problem of component inversion. We assume that a system is represented as an assembly of components. Given this, the component inversion process is divided into two phases by GENISYS: the *data path determination phase*, and the *component inversion phase*. In the first phase, paths to the components driving and driven by the test component are determined. This results in an hierarchy of components called the *component dependence hierarchy*. Each path in this hierarchy forms a *component dependence link*. A link connects driving and driven components in the hierarchy. The component dependence hierarchy is an input to the component inversion phase along with the Rosetta specifications of the structural and the test component. The component inversion engine performs the component inversion process over the components in the hierarchy and generates the *system vectors*. Figure 1.1 shows a functional block diagram of GENISYS.

As shown in figure 1.1, there are two inputs to GENISYS: Rosetta specification of the structural component and that of the test component. The two engines used in GENISYS are, data-path generator and component inversion engine. The former generates the component dependence hierarchy. Two component dependence hierarchies are formed for a given component inter-dependence scenario. This

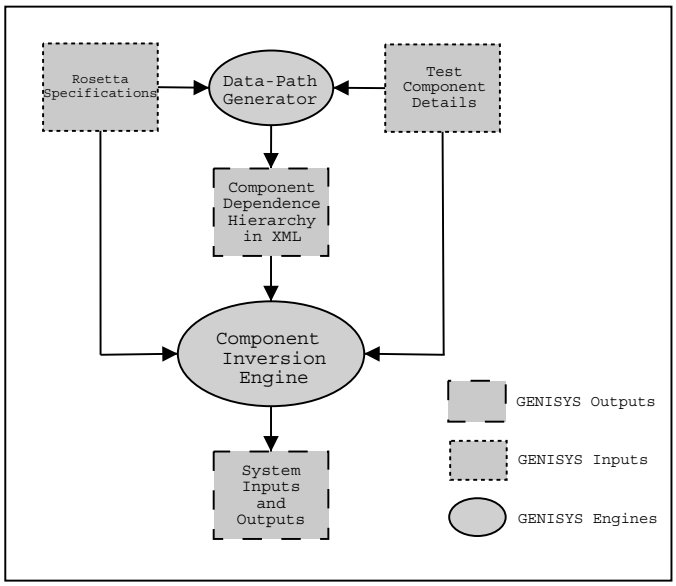

Figure 1.1: Functional Block Diagram of GENISYS

hierarchy gives information about the driving and the driven inner-components in the structural component. Data-path generator also resolves the issues involving scenarios with infinite feedback loop and non-determinacy in the component dependence hierarchy. The component inversion engine is provided with the hierarchies generated in the first phase along with the Rosetta specifications. This phase first identifies the invertible components from the component dependence hierarchies. This process is performed to ensure that only invertible components are processed. Inverting components along both the hierarchies will produce system input and output parameters. The final output of GENISYS includes these parameters in form of system vectors. Each vector contains system output of the structural component and the corresponding system input which generates this output. The GENISYS output is stored in XML format because of its platform-neutrality and inherent flexibility.

1.4 Organization of Thesis

Chapter 2, *Background and Related Work*, gives an introduction to Rosetta [18], a System Level Design Language [17]. It highlights various features of Rosetta including, facets, packages, and domains. It also gives an introduction to XML. Additionally, different features of XML used in this thesis are discussed later in this chapter. DOM [3] is an XML parser used in this thesis. XML Schema [13], a feature of XML, is the template that defines the markup for XML. An introduction to program inversion can be found at the end of this chapter.

Chapter 3, *Data-Path Generator*, gives an overview of the first half of the technical solution. It outlines the data path determination process to form a component dependence hierarchy. An overview of various algorithms used to generate this hierarchy is presented. Vvarious issues including, infinite feedback loops and non-determinacy are tackled by the data path generator towards the end of this chapter.

Chapter 4, *Component Inversion Engine*, describes the second half of the technical solution. The functional block diagram of the component inversion engine is discussed in this chapter. It is followed by an overview of various algorithms used by this engine. Discussion of the conjunctive normal form, boolean SAT solvers, and the zChaff SAT solver [15] employed in this thesis are found here.

In Chapter 5, *Internals of GENISYS*, internal details of GENISYS are explained. Two major databases, *CompMap* and *ParaMap*, used to store information during the data path determination and component inversion process, are

explained in this chapter. It is followed by discussion of the XML and Rosetta Object Model (ROM) parser, and the GENISYS tool usage. The two component dependence hierarchies, *top* and *bottom dependence hierarchy*, and the GENISYS system vectors are discussed here.

Chapter 6, *Testing and Examples*, presents various test scenarios with valid and invalid Rosetta specifications used to test the GENISYS tool. For each valid specification, the GENISYS output XML files are described and for invalid specifications it is shown how GENISYS handles such scenarios.

Chapter 7, *Related Work*, discusses work done by other researchers in the fields of component-based testing and program inversion. Chapter 9, *Conclusions and Future Work*, summarizes this thesis work and proposes future work possible in this field.

Chapter 2

Background

This chapter provides an introduction to Rosetta, a System Level Design Language. Various constructs in Rosetta including, facet, domain, and package are described and an example illustrating Rosetta specification is used to explain the language. Later in the chapter, an introduction to XML [14] is provided. A brief overview of XML is illustrated with a simple example and various advantages and real world applications of XML are provided. It is followed by discussions of various features of XML including, DOM, and XML Schema. An introduction to program inversion ends this chapter.

2.1 Rosetta Specification Language

Rosetta [18] is a System Level Design Language [17] used to design a system at higher levels of abstraction. With increase in the complexity of a system an abstract language such as Rosetta is very helpful to specify its behavior. Charac-

teristics of Rosetta that make it suitable to use even with incomplete information about a system include [16]:

- the ability to integrate information from multiple heterogeneous sources
- declarative modeling
- support for model composition.

The basic unit of specification in Rosetta is called a *facet*. Each facet provides information about a particular aspect of a component or a system. Each facet in Rosetta may use a different model to provide domain-specific vocabulary and semantics in order to support the notion of heterogeneity in designs. A facet is parameterized over the interface of a component and it encapsulates all the Rosetta definitions from basic unit specifications through components and systems. The *facet* keyword marks the beginning of a Rosetta specification. It is followed by a list of interface parameters identifying the component input and output parameters.

A *package* provides a convenient way of aggregating similar Rosetta structures like facet. Each such structure in a package should have a unique label identifier. A structure is visible only with the package that contains it. Structures in different packages may have same label identifier since their scopes don't overlap.

Figure 2.1 illustrates a simple example of Rosetta specification for a schmidt trigger circuit [27]. This specification defines a package, `schidt_trigger_pkg`, that contains a facet, `schmidt_trigger_fct`. The facet has interface parameters,

```
package schmidt_trigger_pkg :: static is
    /* package body begins here */
    facet schmidt_trigger_fct
         (
           /* interface parameters declared */
           input_voltage :: input real;
           output_value  :: output bit
         ) :: state_based is
        /* locally declared variable*/
        b :: bit;
        /* facet body begins here */
        begin
        pre1: (input_voltage > 0.0) and (input_voltage < 5.0);
        post1: if(input_voltage < 1.0)
                   then (b' = 0)
                   else if(input_voltage > 4.0)
                            then (b' = 1)
                            else (b' = b)
                            end if
                   end if;
        post2: (output_value' = b');
    end facet schmidt_trigger_fct;
end package schmidt_trigger_pkg;
```

Figure 2.1: Rosetta Specification for Schmidt Trigger Circuit

`input_voltage` of mode `input` and type `real` and `output_value` of mode `output` and type `bit`. All Rosetta parameters are declared using the notation $x::T$, where x is the parameter name and T is the type of the parameter. The scope of the parameters extends throughout the facet. Domain used in the specification is declared after these parameters. A domain in Rosetta extends the base definition semantics by adding new definitions specific to a design domain. Existing domains include, static, state_based, discrete time, continuous time, frequency, and finite state. Figure 2.2 shows hierarchical representation of various domains existing in Rosetta.

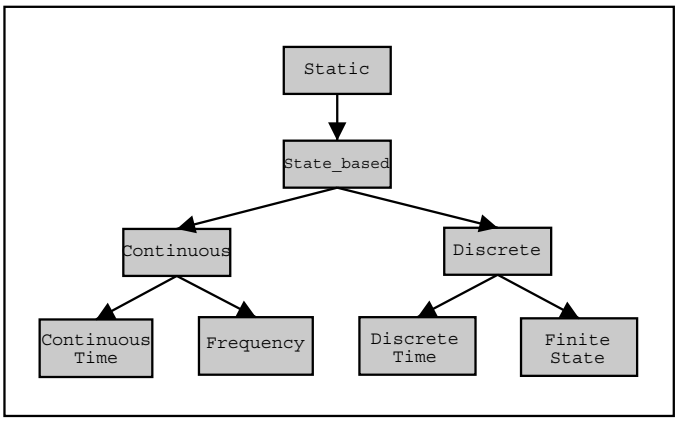

Figure 2.2: Domain Hierarchy in Rosetta

In the schmidt trigger example, the package uses the `static` domain and the facet uses the `state_based` domain. Declaration of local variables is optional and if stated follows the keyword `is`. In the schmidt trigger specification, `b` is the locally declared variable of type `bit` and is visible over the entire facet.

The body of the facet starts with the keyword `begin`. It contains a set of expressions called *terms*. All terms in the specification are boolean expressions. They define the behavior modeled by the facet. The general format of representing a term is *L:Expression*, where *L* is the label associated with the *Expression*. Labels are used to represent the expression specified in a given term. A semicolon ";" marks the end of a term. The body ends with two `end` clauses, which mark the termination of the facet and the enclosing package.

2.2 Extensible Markup Language

XML, or Extensible Markup Language [14], is a W3C-endorsed standard for document markup. This section provides a brief introduction to XML and its features

used in this thesis. Component dependence hierarchies and system vectors, discussed later in this thesis, are represented in XML format. Also the abstract test vectors, used as one of the inputs to the component inversion engine, are generated in XML format.

2.2.1 An Introduction to XML

Because of its inherent flexibility and data-neutrality, XML is widely used as a standard means for data storage and representation. Data is stored in XML documents as strings of text, surrounded by user-defined tags. A particular unit of data and markup is called an *element*. The XML specification defines various rules for representing data in the tags, placing the tags, defining attributes for a particular tag and so on. As mentioned earlier, the tags in XML are user-defined. This helps the users to define their own custom tags based on their application. For instance, a botanist can use tags to describe stem, root, leaf, and other parts of a plant, while a pharmacist can use tags to describe various drugs. XML is similar to HTML, or Hyper Text Markup Language [6]. XML is more flexible than HTML in terms of the tag names, however it is strict in the sense that each and every opening tag has to have a closing tag, which is not the case in HTML.

Figure 2.3 shows a simple example of XML document. `<contact_info>` is the root element of the XML tree. `<address>`, and `<phone>` are its two children elements, that themselves have children. So `<block>`, `<street>`, `<city>` are children of the `<address>` element and grandchildren of root element. Sim-

```
<contact_info>
    <address>
        <block>M</block>
        <street>main</street>
        <city>lawrence</city>
    </address>
    <phone>
        <home>7858888888</home>
        <office>7858648888</office>
    </phone>
</contact_info>
```

Figure 2.3: XML Representation

ilarly, `<home>` and `<office>` are children of `<phone>` element and grandchildren of `<contact_info>` element. In this example, `<contact_info>`, `<address>`, and `<phone>` are complex elements, while the grandchildren of root element are simple elements. An element is complex if it has one or more child element. If an element has only data stored between the opening and closing tags, it is simple.

2.2.2 DOM

DOM, or Document Object Model [3], is a parser for XML documents. It provides a standard programming interface to the applications and is used to access all the building blocks of an XML document. DOM provides Application Programming Interfaces (APIs) for parsing XML documents in many languages and is designed to work with any operating system. The XML document should be loaded in the memory before it can be parsed. It is stored in form of a tree in the memory. The top level element in the XML document becomes the root of the tree and its child elements become children of the root in the tree. Depending on the

element hierarchy in the XML document, the child nodes in the tree may or may not have siblings. The tree terminates with leaf nodes that do not have any children. After loading the XML document in the memory the information is accessed and modified using APIs provided by DOM. Using them a developer can achieve various tasks like, creating new elements, nodes, and attributes, deleting the elements, and modifying their values.

2.2.3 XML Schema

XML schema [13] are used to define the legal building blocks of an XML document. XML schema itself is represented in XML format. It defines all the elements that appear in the XML document. Schema includes all the child elements of a given node and also the order that they may appear. It also provides information regarding the attributes, if any, a particular node has. This information includes, name, type, and use of the attribute. With schema tools can determine whether a given element is empty or not. Since XML schema supports data types, it is easier to describe permissible document content. A schema itself is an XML document and can be parsed using any XML parser. Being extensible in nature, a schema can be reused in other schemas and multiple schemas can be referenced from the same document.

2.3 Introduction to Program Inversion

Program inversion, as a programming technique, has been used in several applications [31, 32, 19]. Inversion is a fundamental concept in mathematics and theoretical computer science [31]. Program inversion refers to the computational process, which determines the input values a program needs in order to generate a given set of output values [31]. A program inverse itself is a program that computes the inverse computation of another program. *Function inversion* is the process of deriving for some one-to-one function $f{:}X \rightarrow Y$ the inverse function $f^{-1}{:}\ Y \rightarrow X$ such that $f^{-1}(f(x)) = x$. For example, let $f(x) = x^2$ and $f^{-1} = \sqrt{x}$. So we get,

$$f^{-1}(f(x)) = \sqrt{x^2} \ = \pm x$$

Program inversion is similar to function inversion. A conventional view of computation is for a program to first read some initial input values, perform some computation, and produce output values as a solution for the given input. Given a final set of output values of a forward-directed program, an inverted computation derives the input values that the program would use to compute the output. In the example given above, given an input value, say m, to f, the inverse function, f^{-1}, is one that takes the squared value of m and returns its square root, say n. Thus,

$$n = \sqrt{m^2} \ = \pm m.$$

This example shows an important fact about program inversion - the output of a program may be produced by many possible inputs. Thus, if the forward-

directed computation is *many-to-one*, the computational inverse is modelled by a relation over multiple sets of input and output values. Figure 2.4 shows the *AND/OR tree* model [24] for a many-to-one forward computation [31].

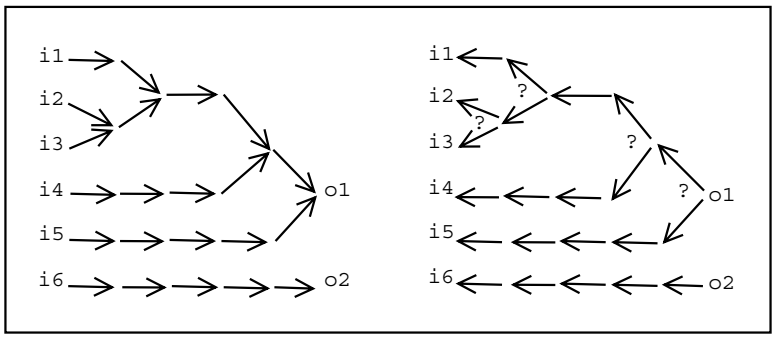

Figure 2.4: AND/OR Tree Model for Forward and Inverse Computation

The forward-directed computation starts at one of a number of initial input states (i_1, ..., i_6) on the left tree. The state changes deterministically until a final state of o_1 or o_2 is reached. Since all the state transition paths are deterministic, the AND/OR tree derivation for each computation takes the form of a single branching tree with only AND nodes. The right tree shows the inverse of the forward computation on the left tree. In this tree, there are nondeterministic choice of paths to take when inverting many computation steps. The AND/OR tree model ascribes OR nodes to such nondeterministic choice of paths. In the figure 2.4, these nodes are labelled with "?".

The computability of inverse of a forward-directed computation is affected by the following [31]:

- *Expression Invertability:* All mathematical primitives must have supplied for them effective means of computing their inverses. To invert an expression, an

adequate definition of its component primitive operators and the availability of data arguments is needed.

- *Control:* Inversion of a computation greatly depends on the type of control structure it has. Inverting the iterative structure of iterative loops requires that the *least fixpoints* of the loops are computed. Non-terminating loops cannot be inverted.

- *Algorithm Properties:* The properties of an algorithm drive the inversion process. These properties decide whether a given algorithm can be inverted or not. They may even allow an algorithm to have a tractable inversion. Since these properties may not be common among all the algorithms, a decision procedure for general inversions does not exist.

Chapter 3

Data-Path Generator

This chapter outlines the data-path determination process in a component dependence hierarchy. An overview of algorithms to tackle various issues like, infinite feedback loop and non-determinacy is presented here. In this thesis, the terms *component* and *facet* are used interchangeably, as each and every component in the component dependence hierarchy has a corresponding facet declared in the Rosetta specification. The only difference between them is that a given component in the hierarchy can instantiate only one facet in the specification, while a given facet can be instantiated by many components. In other words, given the component we can always find the corresponding facet but not the other way round.

3.1 Concept of Level in GENISYS

In GENISYS, there is a concept of level that is used during component inversion phase. All components at the same depth in the component dependence hierarchy are grouped in the same level. This concept of level is used to distinguish the extend of inter-dependence between a given component and the test component (TC). There might also be inter-dependence links among components in the same level.

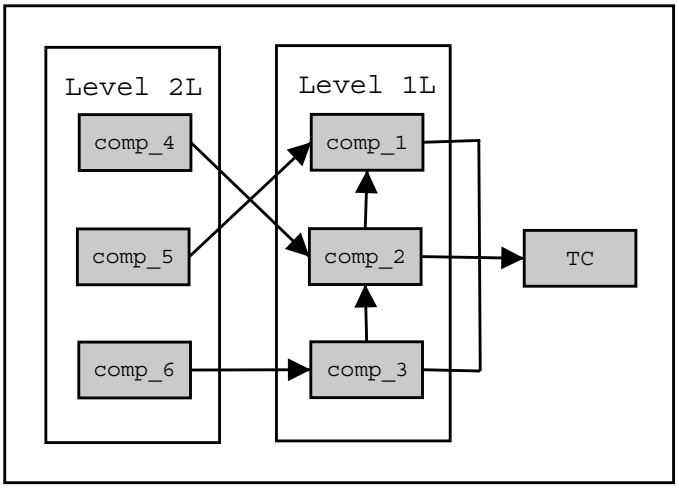

Figure 3.1: Level in GENISYS

In figure 3.1, components $comp_1$, $comp_2$, and $comp_3$ belong to level 1L. Components, $comp_4$, $comp_5$, and $comp_6$ belong to level 2L. There are also *inter-dependence links* among components in the same level. Components $comp_2$ and $comp_3$ also belong to level 2L due to following inter-dependence links.

$$comp_2 \rightarrow comp_1 \rightarrow \texttt{TC}$$

$$comp_3 \rightarrow comp_2 \rightarrow \texttt{TC}$$

Components $comp_3$, $comp_4$, and $comp_6$ belong to level 3L as a result of the following links.

$$comp_3 \rightarrow comp_2 \rightarrow comp_1 \rightarrow \texttt{TC}$$

$$comp_4 \rightarrow comp_2 \rightarrow comp_1 \rightarrow \texttt{TC}$$

$$comp_6 \rightarrow comp_3 \rightarrow comp_2 \rightarrow \texttt{TC}$$

Component $comp_6$ belongs to level 4L due to the link:

$$comp_6 \rightarrow comp_3 \rightarrow comp_2 \rightarrow comp_1 \rightarrow \texttt{TC}$$

In specifications, a component may belong to several levels simultaneously. During component inversion such a component is grouped in the highest numbered level it belongs to. This is due to the fact that during component inversion, we traverse through the components hierarchy in a fashion that ensures that the output parameters are known before inverting a given component. Due to the inter-dependence links among various component in the same level, it is inevitable to separate the level they belong to. Otherwise, trying to invert a component with some of the output parameters unknown will make the component non-invertible. Applying this concept to the example given in figure 3.1 gives result shown in figure 3.2.

As shown in the figure 3.2, the components are grouped in level as far away from TC as possible. This ensures that before inverting a given component all of its output parameters are known.

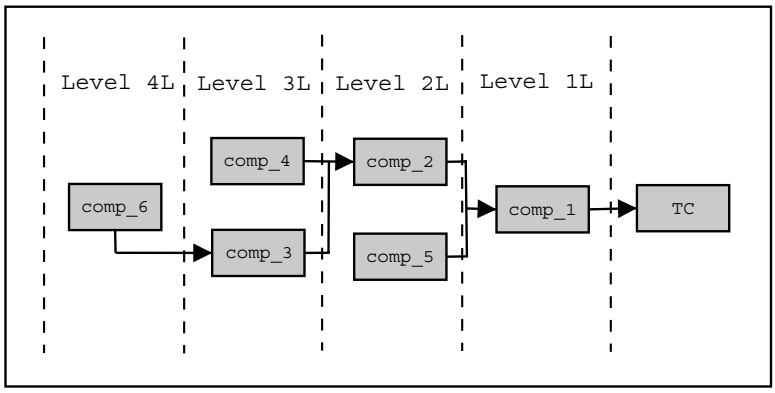

Figure 3.2: Expanded Level in GENISYS

3.2 Data path Determination

Figure 3.3 shows a block diagram representation of the GENISYS data-path determination phase. The data-path generator takes in two inputs: Rosetta specifications of various components in the component dependence hierarchy; and the test component. The output of this phase comprises of two component dependence hierarchies traversing up and down from the test component. The data-path generator tackles various issues such as, infinite feedback loop and non-determinacy in a given component dependence hierarchy.

3.3 Component Dependence Hierarchy

The component dependence hierarchies are produced as part of the output of the data-path determination phase of GENISYS. Figure 3.4 illustrates the component dependence hierarchy. As explained earlier, each component belongs to a unique level. The level a component belongs to is an indication of the extend of inter-dependence between the component and the test component. The closer the level

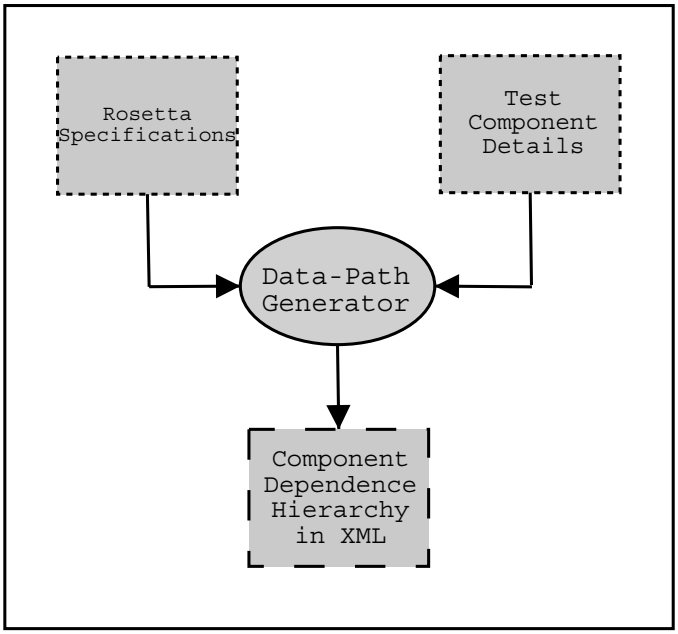

Figure 3.3: Data-Path Generator

is to the test component, the more inter-dependence is present. In a given scenario, a component may belong to several levels simultaneously. This issue is resolved using the technique described earlier.

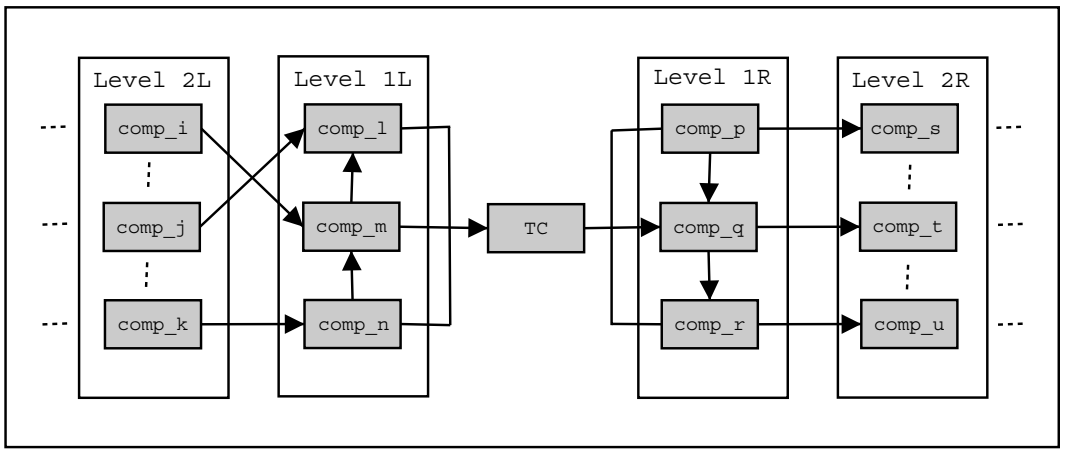

Figure 3.4: Component Dependence Hierarchy

The two component dependence hierarhies are generated in XML format. One traverses from the test component up to the top of the hierarchy as depicted in

figure 3.4. This hierarchy is called *driving component dependence hierarchy*. The general format of this hierarchy is shown in figure 3.5.

```
<COMPONENT_HIERARCHY file = "STRUCTURAL_COMPONENT_SPECS_FILE">
 <TEST_COMPONENT component_name = "TEST_COMPONENT_NAME">
  <DRIVING_COMPONENT component_name = "DRIVING_COMPONENT_1"
                     driving_variable = "DRIVING_PARA_1">
   <DRIVING_COMPONENT component_name = "DRIVING_COMPONENT_2"
                      driving_variable = "DRIVING_PARA_2">
   </DRIVING_COMPONENT>
  </DRIVING_COMPONENT>
  .      .      .
 </TEST_COMPONENT>
</COMPONENT_HIERARCHY>
```

Figure 3.5: Driving Component Dependence Hierarchy

The second hierarchy traverses from the test component down to the bottom of the component dependence hierarchy. This hierarchy is called *driven component dependence hierarchy*. Figure 3.6 shows the general format of driven component dependence hierarchy. Examples for both the hierarchies are given in chapter 6.

```
<COMPONENT_HIERARCHY file = "STRUCTURAL_COMPONENT_SPECS_FILE">
 <TEST_COMPONENT component_name = "TEST_COMPONENT_NAME">
  <DRIVEN_COMPONENT component_name = "DRIVEN_COMPONENT_1"
                    driving_variable = "DRIVEN_PARA_1">
   <DRIVEN_COMPONENT component_name = "DRIVEN_COMPONENT_2"
                     driving_variable = "DRIVEN_PARA_2">
   </DRIVEN_COMPONENT>
  </DRIVEN_COMPONENT>
  .      .      .
 </TEST_COMPONENT>
</COMPONENT_HIERARCHY>
```

Figure 3.6: Driven Component Dependence Hierarchy

3.4 Feedback Loops

A facet *fct_driven* depends on another facet *fct_driving* if one or more input parameters to fct_driven are directly or indirectly driven by fct_driving. A parameter is said to be driven by a facet if it is an output parameter of that facet. Consider the facet signatures given in figure 3.7.

```
facet fct_driving(A :: input bit; B :: output real) :: logic is
...
end facet fct_driving;

facet fct_driven(B :: input real; C :: output real) :: logic is
...
end facet fct_driven;
```

Figure 3.7: Facet Signatures illustrating Facet Dependence Link

In figure 3.7, the facet `fct_driven` depends on the facet `fct_driving` through the parameter `B`. In other words, facet fct_driving provides input to facet fct_driven through B. Facet fct_driving might be driven by a facet, that depends on some other facet. This results in a *facet dependence hierarchy*. At times, this hierarchy contains loops, wherein the driven facet directly or indirectly drives the driving facet. This loop is termed as a *feedback loop*. In such cases, tracing the facet dependence hierarchy results in looping infinitely along the hierarchy. Hence it is also called an *infinite feedback loop*. Figure 3.8 illustrates a simple example of infinite feedback loop.

Facet `fct_fdbk_1` drives facet `fct_fdbk_2` through the parameter, `B`, while the facet `fct_fdbk_2` drives facet `fct_fdbk_2` through the parameter, `A`. Infinite

```
facet fct_fdbk_1(A :: input bit; B :: output real) :: logic is
...
end facet fct_fdbk_1;

facet fct_fdbk_2(B :: input real; A :: output bit) :: logic is
...
end facet fct_fdbk_2;
```

Figure 3.8: Facet Signatures illustrating Infinite Feedback Loop

feedback loop needs to be broken. In this section, first different types of feedback loops are described with examples and later on various issues to be kept in mind while breaking a feedback loop are discussed.

3.4.1 Types of Feedback Loops

Feedback loops are formed due to inter-dependence between one or more facets in the hierarchy. Feedback loops are grouped into the following categories depending on the number of facets involved in the loop.

Self Feedback Loop

Self feedback loops are direct feedback loops involving a single facet. They are formed when the facet *drives* itself. Specifically, when a facet has one parameter common in its input and output parameter lists, a self feedback loop is formed. Figure 3.9 gives an example of this scenario illustrated in figure 3.10 as a Rosetta specification.

As shown above, facet `selfFB` drives back itself through the parameter `A` resulting into a self feedback.

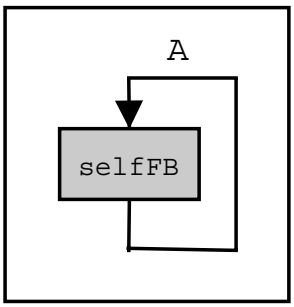

Figure 3.9: Self Feedback Loop

```
facet selfFB(A :: input real; B :: output real;
             A :: output real) :: logic is
...
end facet selfFB;
```

Figure 3.10: Facet Signature illustrating Self Feedback Loop

Primary Feedback Loop

Feedback loops formed in scenarios with exactly two facets involved in the loop are called *primary feedback loops*. Figure 3.11 illustrates this scenario while a Rosetta specification illustrating primary feedback loop is given in figure 3.12.

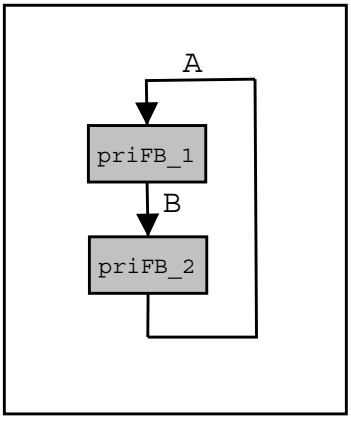

Figure 3.11: Primary Feedback Loop

As shown in these figures, facet `priFB_1` drives facet `priFB_2` through the

```
facet priFB_1(A :: input real; B :: output real) :: logic is
...
end facet priFB_1;

facet priFB_2(B :: input real; A :: output real) :: logic is
...
end facet priFB_2;
```

Figure 3.12: Facet Signatures illustrating Primary Feedback Loop

parameter `B`, while the facet priFB_2 drives facet priFB_1 through the parameter `A` forming a primary feedback loop.

Secondary Feedback Loop

Secondary feedback loops, or indirect feedback loops, are formed in scenarios where more than two facets are involved in the loop formation. Figure 3.13 illustrates this scenario and figure 3.14 gives Rosetta specification illustrating a secondary feedback loop.

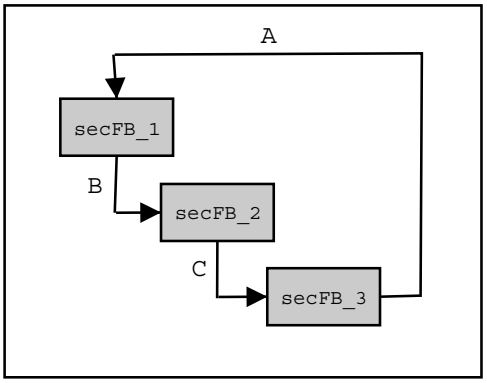

Figure 3.13: Secondary Feedback Loop

In these figures, facet `secFB_1` drives facet `secFB_2` through the parameter `B`, facet `secFB_2` drives facet `secFB_3` through the parameter `C`, and facet `secFB_3`

drives facet `secFB_1` through the parameter `A` completing the secondary feedback loop.

```
facet secFB_1(A :: input real; B :: output real) :: logic is
...
end facet secFB_1;

facet secFB_2(B :: input real; C :: output real) :: logic is
...
end facet secFB_2;

facet secFB_3(C :: input real; A :: output real) :: logic is
...
end facet secFB_3;
```

Figure 3.14: Facet Signatures illustrating Secondary Feedback Loop

3.4.2 Loop Considerations

When a feedback loop is detected, it has to be *broken* to avoid infinite looping while traversing the facet dependence hierarchy. Deciding which link involved in the feedback loop is to be broken involves various considerations. The simple and efficient solution to this problem is to break the link that results into the feedback loop. In other words, break the last link found after which the feedback loop is detected. The feedback loop is not detected until all the links forming it are found. So breaking the last link, which completes the feedback loop, gives a simple yet fast remedy to cure the infinite feedback loop problem. The decision depends on the following criteria.

- Number of input and output parameters to the facets involved in the feed-

back loop.

- Position in the facet dependence hierarchy. If the facet is generating system outputs, its link is less likely to be broken.

- Role played by the facet in the facet dependence hierarchy. The more important a facet is in the hierarchy, the less probable is its link to be broken. A facet is important in the hierarchy if breaking its link causes the hierarchy to be non-contiguous because component inversion is not possible if the hierarchy is non-contiguous.

3.5 Non-Determinacy in Hierarchy

Non-determinacy represents a situation where a parameter is driven by more than one facet. Such a situation occurs when output parameter lists of two or more facets have atleast one parameter in common. When two or more facets have same output parameter, the value of that parameter cannot be determined deterministically. This non-determinism is caused due to the fact that the parameter is driven by more than one facet and since there is no sequential ordering between these facets its value is ambiguous.

Figure 3.15 illustrates a scenario of non-determinacy found in the component dependence hierarchy and the corresponding Rosetta specification is given in figure 3.16. As shown in these figures, facets `nonDeter_1` and `nonDeter_2` both drive facet `nonDeter_3` through the parameter `B`, which results into non-determinacy in

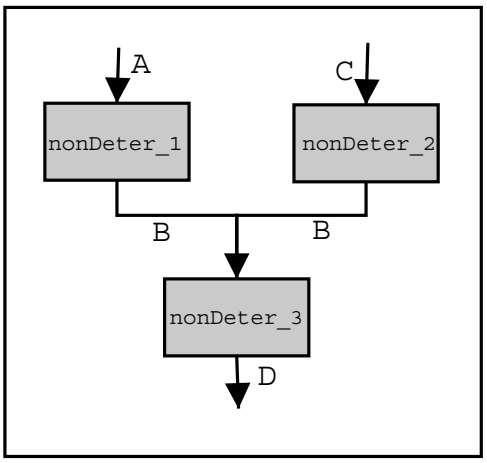

Figure 3.15: Non-Determinacy in Hierarchy

the hierarchy.

```
facet nonDeter_1(A :: input real; B :: output real) :: logic is
...
end facet nonDeter_1;

facet nonDeter_2(C :: input real; B :: output real) :: logic is
...
end facet nonDeter_2;

facet nonDeter_3(B :: input real; D :: output real) :: logic is
...
end facet nonDeter_3;
```

Figure 3.16: Facet Signatures illustrating Non-Determinacy in Hierarchy

Chapter 4

Component Inversion Engine

This chapter describes the inversion phase of GENISYS and forms a major portion of this thesis work. An overview of the component inversion phase is given in the first section. Various algorithms used during this phase are explained here. Two major algorithms described are, algorithm to invert boolean and if-then-else expressions. Brief introductions to conjunctive normal form (CNF), Boolean Satisfiability (SAT), and SAT solvers are also presented in this chapter.

4.1 Component Inversion Phase of GENISYS

Figure 4.1 represents a functional block diagram of the component inversion engine. As shown in the figure, the engine takes a Rosetta specifications of various components, test component details, and the component dependence hierarchy as inputs. The hierarchy is the output of the data-path generator described earlier.

The component inversion engine produces the system vectors as the output.

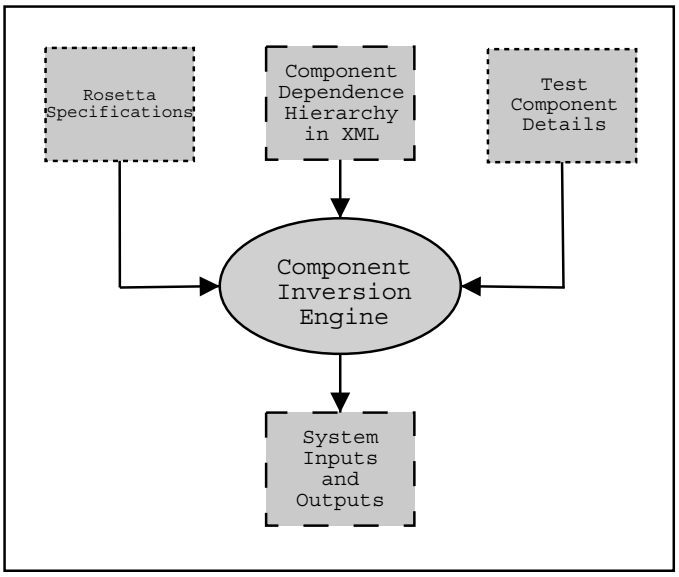

Figure 4.1: Component Inversion Engine

These system vectors comprise system inputs and outputs of the structural component representing test vectors. The format of these vectors is shown below.

```
<TestData>
   <Config>
      <DataConfig>
         <Name>ipt_para_name</Name>
         <Index>ipt_para_index</Index>
         <InputType/>
      </DataConfig>
      .     .     .
  </Config>
   <TestSet>
      <TestVector>
         <Input>
            <Name>ipt_para_name</Name>
            <Value>ipt_para_value</Value>
         </Input>
         .     .     .
      </TestVector>
      .     .     .
   <TestSet>
</TestData>
```

4.2 Algorithm for Component Inversion Engine

The component inversion engine follows the following algorithm.

- Identify invertible components from the component dependence hierarchy. If all the output parameters of a component are known, it is invertible. Components with unknown output parameters and components not in the component dependence hierarchy are non-invertible.
- Assign each component to a given level.
- Parse the abstract test vectors for the test component and perform the component inversion for each test vectors found in the abstract test vectors. Component inversion will generate values for input parameters for each test vector.
- Invert components starting from *Level 1L* up to last level on left-side of the test component. This will ensure that all the output parameters of a component are known before inverting it.
- Invert all invertible components in the given level using the algorithm described later in this chapter.

4.3 Algorithm to Identify Invertible Components

An algorithm to determine invertible components is required to ensure that only invertible components are processed. A component is said to be invertible if it

belongs to the component dependence hierarchies generated during the data-path determination phase of GENISYS. If a component belongs to any of these hierarchies, it is directly or indirectly related to the test component. If a component does not belong to any of these hierarchies, it is independent of the test component. The component inversion engine inverts the components directly or indirectly driving the test component to generate the system inputs. This engine can never reach components independent of the test component. Hence these components can not be inverted. Such independent components are marked as *non-invertible components.*

4.4 Algorithm to Invert a Component

An algorithm is used to invert a component. This algorithm is invoked while inverting all the component in a given level for all the level in a given component dependence hierarchy. This algorithm follows the following steps.

- Populate the local information storage specific to the given component from the global information storage.
- Make sure that all the output parameters of the given component are known. If any output parameter is unknown, the component cannot be inverted.
- Invert all the expressions of the component in order from last to first. Expressions currently supported are boolean/bit and if-then-else expressions. Apply corresponding algorithm to invert these expressions. It is assumed

that a given expression contains all the parameters of the same type. In other words, a boolean expression has all its parameters of type boolean.

- Populate the local information storage with the values computed after an expression is inverted.
- After all the expressions are inverted, populate the global information storage from the local information storage with the input parameter values computed for the given component.
- Print the system vector corresponding to the given test vector in the XML format.

4.5 Algorithm to Invert Boolean Expressions

A boolean expression is an expression that involves computation over boolean or bit parameters. In this section, the algorithm used to invert a boolean expression is described. A brief overview of conjunctive normal form, Boolean Satisfiability (SAT), and various SAT solvers is given towards the starting of this section.

4.5.1 Conjunctive Normal Form

CNF, or Conjunctive Normal Form, is a common form of representing boolean expressions. For our purposes, Boolean satisfiability problems are represented in CNF format and fed as input to the SAT solver. This form consists of the logical *AND* of one or more *clauses*, that consist of the logical *OR* of one or more *literals*.

In other words, CNF comprises of conjunction of disjunctions of literals. The literal comprises the fundamental logical unit in the expression, being merely an instance of a variable or its complement. Complement is represented by $\neg$. The general format of CNF is given below, where, $clause_1$, $clause_2$, . . . , $clause_n$ are clauses of the CNF format.

$clause_1 \wedge clause_2 \ \ldots \ \wedge clause_n$

Each clause is disjunction of literals. The general format of a clause is given below.

$var_1 \vee var_2 \ \ldots \ \vee var_m$

The CNF expression shown above is the same as the *product of sum* form in boolean algebra. The advantage of CNF is that in this form, for an entire expression to be satisfied (*sat*) all of its clauses must also be *sat*, since they are logically *AND-ed*. An example of CNF expression is given below.

`f = (A` $\vee$ `B` $\vee \neg$ `C)` $\wedge$ `(B` $\vee$ `D)` $\wedge$ `(`$\neg$ `A)`

`A, B,` and `C` are literals, each of which is a variable or the negation of a variable. `(A` $\vee$ `B` $\vee \neg$ `C)` is a clause, which is disjunction of literals. Each clause is a requirement which must be satisfied for `f` to be *sat*. Any boolean expression can be converted into CNF expression. An algorithm to convert a given boolean expression into CNF is discussed later in this chapter.

4.5.2 Boolean Satisfiability (SAT)

Boolean satisfiability is one of the most studied of the combinational optimization problems [28]. Significant effort has been spent trying to provide practical solutions to the satisfiability problem. This problem has been found in a vast variety of applications ranging from Electronic Design Automation to Artificial Intelligence. The SAT problem consists of determining a satisfying variable assignment, V, for a Boolean function, f, or determining that no such V exists. SAT is one of the central NP-complete problems. Because SAT lies at the core of many practical application domains, the subject of practical SAT solvers has received considerable research attention, and numerous solver algorithms have been proposed and implemented.

4.5.3 Introduction to SAT Solvers

Various SAT solvers have been developed by researchers working in this field. Some of the SAT solvers include, Chaff [28], GRASP [5], SATO [9], and Walk-SAT [12]. Chaff SAT solver, employed in this thesis, has an efficient implementation of the Boolean Constraint Propagation (BCP). GRASP, or Generic seaRch Algorithm for the Satisfiability Problem, is a propositional satisfiability (SAT) solver. Most of these SAT solvers employ combination of two main strategies: Davis-Putnam backtrack search [22] and heuristic local search. The former technique is more complete than the latter in the sense that the latter technique does not guarantee to find a satisfying assignment if one exists or prove unsatisfiability.

Hence DP search algorithm is mostly used in complete SAT solvers like Chaff.

4.5.4 zChaff SAT Solver

Chaff [28], is a complete SAT solver employing the DP search algorithm. It achieves significant gain in performance by carefully engineering all aspects of the search algorithm. The developers of Chaff have especially optimized the implementation of *Boolean Constraint Propagation* (BCP). Chaff has been successful in achieving one to two orders of magnitude performance improvement on even difficult SAT benchmarks in comparison with other SAT solvers like GRASP and SATO.

zChaff [15], an implementation of the Chaff SAT Solver, is maintained by Zhaohui Fu [4]. The latest version of zChaff has been used in this thesis for inverting boolean and bit-value expressions. zChaff is designed with performance and capacity in mind. zChaff can be compiled into a linkable library for integration purpose so that the users do not need to export instances into intermediate files to use zChaff.

4.5.5 zChaff CNF File Format

zChaff accepts a specific format of the CNF expression. A file containing the expression in this format is passed to the SAT solver. The format of the CNF file has the following rules [33]:

- CNF file name should end with *.cnf* extension.

- A line in the CNF file starting with the character 'c' contains comments and hence is ignored by the SAT solver. This feature can be used to provide additional information like, the boolean expression whose CNF form this file contains.

- The prelude of the CNF file contains information regarding number of variables and clauses in the CNF expression. The format of a typical prelude is given below. N_{vars} is the number of variables used in the expression and $N_{clauses}$ is the number of clauses used in the expression.

  `p cnf` N_{vars} $N_{clauses}$

- *Variables* in the CNF file are expressed as numbers from 1 to N_{vars}.

- A *literal* can either be a variable or its complement.

- The complement of a variable is expressed as the *negation* of the number representing the variable. For instance, if *6* corresponds to variable *x6*, *-6* will represent $\neg x6$.

- A *clause* in the CNF file is a line of literals separated by spaces. It terminates with a *0*. For example, consider the following conversion.

  `((x1)` $\vee$ `(¬x3)` $\vee$ `(¬x1)` $\vee$ `(x10)` $\vee$ `(¬x2))` $\rightarrow$ `(1 -3 -1 10 -2 0)`

- A line with a single *0* denote end of the CNF file.

Below is an example of a CNF file:

```
c CNF form of the bitvalue expression:

c ((x1 and not(x2))  or (x3 and x4) or (not(x5) and x6))

p cnf 6 3

1  -2  0

3   4  0

-5  6  0

0
```

4.5.6 Algorithm to Convert Boolean Expressions to CNF

An algorithm to convert any given boolean expression into an equivalent *Conjunctive Normal Form*, or CNF, is used to prepare inputs for SAT solver. This algorithm is quite common [1, 2]. The steps involved are:

- Eliminate the arrows using definitions.

$$(\mathrm{A} \rightarrow \mathrm{B}) \equiv \neg\mathrm{A} \vee \mathrm{B}$$

- Drive the negations in using De Morgan's Laws.

$$\neg(\mathrm{A} \vee \mathrm{B}) \equiv \neg\mathrm{A} \wedge \neg\mathrm{B}$$

$$\neg(\mathrm{A} \wedge \mathrm{B}) \equiv \neg\mathrm{A} \vee \neg\mathrm{B}$$

- Distribute *OR* over *AND*.

$$(\mathrm{A} \vee (\mathrm{B} \wedge \mathrm{C})) \equiv (\mathrm{A} \vee \mathrm{B}) \wedge (\mathrm{A} \vee \mathrm{C})$$

Additional rules used to simplify a given expression in CNF are:

- An empty clause is false, since no options are there to be satisfied.
- A sentence with no clauses is true, as there are no requirements.
- A sentence containing an empty clause is false because it is impossible to satisfy this clause.

Following are the steps involved in coverting the expression $(A \vee B) \rightarrow (C \vee D)$ to conjunctive normal form.

- Eliminate arrows: $\neg(A \vee B) \vee (C \vee D)$
- Drive in negations: $(\neg A \wedge \neg B) \vee (C \vee D)$
- Distribute: $(\neg A \vee C \vee D) \wedge (\neg B \vee C \vee D)$

After a given boolean expression is converted to its equivalent conjunctive normal form, the CNF file is created following the rules stated earlier. This file is passed to the zChaff SAT solver to solve the given boolean expression. The output of the SAT solver will either be, valid assignments to the parameters involved in the boolean expression, or an error in case the expression can not be satisfied. Constraints, if any, can also be passed to the SAT solver along with the CNF equivalent of the boolean expression. For example, if in the above expression value of A is restricted to 0, a clause: `-x 0` can be added to the CNF file, where `x` is the number associated with the parameter A. This clause will force the SAT solver to generate valid assignments, if possible, for all the parameters such that value of A is 0.

4.6 Algorithm to Invert If-then-else Expression

In Rosetta, every term is a boolean expression. Thus, an *if-then-else* expression is also boolean rather than a control statement. Any if-then-else expression can be *flattened* into an equivalent boolean expression. Consider the if-then-else expression given below.

```
if(x)
   then (y) else (z)
end if;
```

This expression can be flattened into an boolean expression: `(x` $\wedge$ `y)` $\vee$ `(`$\neg$`x` $\wedge$ `z)`

Rosetta supports both *simple* and *nested* if-then-else expressions. Since any if-then-else expression can be expressed in an boolean form, conceivably the SAT solver can be used to solve the boolean equivalent of the if-then-else expression. The following are the steps involved in inverting a given if-then-else expression.

- Ensure that the if-then-else expression is in a valid format. In other words, check the validity of all the sub-expressions. Make sure that the if condition returns a boolean.
- Traverse the expression until the inner most if-then-else expression is reached.
- Transform the if-then-else expression into a boolean form as explained earlier.
- Recursively convert the if-then-else expressions into their boolean equivalent and pass them to their parent expression, if any.

- After generating the equivalent boolean expression of a given if-then-else expression, invert it using the SAT solver. Pass constraints over the boolean expressions to the SAT solver.

- The SAT solver will produce valid assignments for all the sub-expressions, if possible. Otherwise, it will generate an error stating that the expression can not be satisfied.

- In case of valid output generated by the SAT solver, utilize the output to assign values to all the parameters involved in the if-then-else expression.

An example of a complex if-then-else expression in Rosetta is given below.

```
if(%signal)
      then if(%sigNum')
                  then (opt' = sigVal') else (opt' = not(sigVal'))
                  end if
      else if(not(%sigStr'))
                  then (opt' = sigStr') else (opt' = not(sigStr'))
                  end if
      end if;
```

A simplified version of the if-then-else expression given above is:

```
if(A)
   then if(B)
           then (C)  else (D)
           end if
   else if(E)
           then (F)  else (G)
           end if
   end if;
```

In the above expression `A, B,...,G` are boolean. Applying the algorithm to the above mentioned if-then-else expression will produce the following results.

- `if(B) then (C) else (D)` transforms into `(B` $\wedge$ `C)` $\vee$ `(`$\neg$`B` $\wedge$ `D)`.
- `if(E) then (F) else (G)` transforms into `(E` $\wedge$ `F)` $\vee$ `(`$\neg$`E` $\wedge$ `G)`.
- The entire if-then-else expression transforms into:

  `(A` $\wedge$ `((B` $\wedge$ `C)` $\vee$ `(`$\neg$`B` $\wedge$ `D)))` $\vee$ `(`$\neg$`A` $\wedge$ `((E` $\wedge$ `F)` $\vee$ `(`$\neg$`E` $\wedge$ `G)))`

A simplified boolean form of this expression is given below. It is the *disjunctive normal form*, DNF, of the boolean expression generated above.

`(A` $\wedge$ `B` $\wedge$ `C)` $\vee$ `(A` $\wedge$ $\neg$`B` $\wedge$ `D)` $\vee$ `(`$\neg$`A` $\wedge$ `E` $\wedge$ `F)` $\vee$ `(`$\neg$`A` $\wedge$ $\neg$`E` $\wedge$ `G)`

Performing boolean expression inversion over the above mentioned boolean expression will generate valid assignments for all the sub-expressions (`A, B,..., G`). Value of 1 implies that the corresponding sub-expression is *true* and should be inverted to compute the unknown parameters. Value of 0 implies that the corresponding sub-expression is *false* and should only be inverted if it does not clash with existing known parameters. The parameters in a true sub-expression play an important role in deciding the flow of the expression, while those in a false sub-expression have little or no influence over the flow. The approach employed first inverts all the true sub-expressions. Later, the false sub-expressions are inverted to compute the values of parameters that are still unknown.

4.7 Component Inversion and SAT Solver

As described earlier, a SAT solver can be used to derive values of unknown parameters in an expression. This expression can be a function F over some parameter

X where the value of the output parameter Y is known to be V. Thus, the SAT solver will try to find the value of X that satisfies the following equation. If SAT solver is able to derive some value W for X that satisfies this equation, the expression is *sat*. Otherwise, it is not *sat*.

$$F(X) = V$$

In Rosetta every term is boolean and a boolean expression can be converted into its equivalent conjunctive normal form (CNF). This concept is used in the component inversion phase of GENISYS. A component is inverted by inverting all its expressions in order from last to first. While inverting an expression, GENISYS utilizes the fact that an expression can be converted into its equivalent CNF format and the SAT solver can be used to process the expression. As shown in the example above, the SAT solver tries to derive values for the unknown variables in the expressions if the expression is satisfiable. These unknown variables are the input parameters and locally declared variables in the component. Inverting all the satisfiable expressions in a component using the SAT solver generates the input parameters of the component. These input, or driving, parameters of a component are output, or driven, parameters of another component. Inverting all the inner-component will generate the system inputs of the structural component.

Chapter 5

GENISYS Internals

This chapter outlines the internals of the GENISYS tool. Various databases used to store information related to the component dependence hierarchy are described in this chapter. It is followed by discussion of the two parsers used in this thesis: *XML* and *ROM parser*. The GENISYS tool usage is also described in this chapter. Later, GENISYS output XML files containing, the two component dependence hierarchies, *top* and *bottom dependence hierarchy* and the *system vectors*, are explained.

5.1 CompMap - Database

In GENISYS, a database is used to store various information regarding a component. A data structure called *GenisysCompMap* is used for this purpose. This database stores the following information regarding a given component.

- **Facet Object:** A mapping from the component to the corresponding facet object is required to perform component inversion.

- **Inversion Type:** Information regarding whether the component is invertible or not is required to make sure that only invertible components are processed.

- **Level:** Information regarding the level to which a component belongs is required while performing component inversion.

- **Interface Parameters:** Information regarding the interface parameters of the component is required to form component dependence hierarchy.

This database is populated by the data-path generator while determining the component dependence hierarchy and is used by the component inversion engine during second phase of GENISYS.

5.2 ParaMap - Database

A parameter in the Rosetta specification may be a facet interface parameter or a locally declared variable. *GenisysParaMap* is a data structure that stores following information related to a given parameter.

- **Parameter Name:** Name assigned to the parameter to uniquely identify it in the component it belongs to.

- **Parameter Type:** Type of the parameter. It can be any of the possible valid types defined in Rosetta.
- **Parameter Value:** Value assigned to the parameter depending on the type.
- **Driven Component List:** A list of components driven by the parameter. These components take this parameter as one of the input.
- **Driving Component:** Component driving the parameter. There can be only one driving component for a given parameter. If more than one component tries to drive a given parameter, the parameter value is non-deterministic.

5.3 XML Parser

The abstract test vectors of the test component generated by the Design Verification Test Generation tool (DVTG) [26, 16, 30, 34] are in XML format. Each vector contains a set of input values and a corresponding set of expected output values. These vectors are parsed using an XML parser in order to retrieve the input parameter values. These vectors are fed to the GENISYS engine for component inversion. They provide information about the input and desired output parameters of the test component. Shown below is the format of these test vectors generated by the VectorGen [11].

```
<TestData>
   <Config>
      <DataConfig>
```

```
         <Name>ipt_para_name</Name>
         <Index>ipt_para_index</Index>
         <InputType/>
      </DataConfig>
   </Config>
   <TestSet>
      <TestVector>
         <Input>
            <Name>ipt_para_name</Name>
            <Value>ipt_para_value</Value>
         </Input>
      </TestVector>
   </TestSet>
</TestData>
```

Document Object Model (DOM) [3] is the XML parser used in this thesis. It forms an hierarchical tree structure with the top level XML element as the root of the tree. The entire tree is parsed in hierarchical order retrieving desired information. Leaf nodes mark end of the tree.

5.4 ROM Parser

A *Rosetta Object Model (ROM)* is built to store the information regarding the Rosetta specification. ROM is populated while parsing the specifications. GENISYS needs information stored in ROM during the data-path determination and the component inversion phase. To retrieve this information a ROM parser is used. It parses the ROM and stores the required data. This data is utilized while performing various other operations. Various issues including, infinite feedback loop, and non-determinacy are resolved by the parser while traversing the ROM. If any

invalid specification is found, the parsing process terminates with an appropriate error message.

5.5 GENISYS Tool Usage

The GENISYS tool performs two operations: *data-path determination* for component dependence hierarchy; and *component inversion* using the information from the hierarchy with a single command. This implies that the command-line argument should have enough information to perform both the operations without any further user intervention. The command-line argument must have:

- the information about the Rosetta specification file containing the structural facet.
- the information regarding the *test component*.
- the information about the file containing the abstract test vectors for the test component.

Figure 5.1 shows the command-line argument of the GENISYS tool. The path to the file containing abstract test vectors in XML format is optional. If the path is not specified, the default location of this file is in the directory containing the Rosetta specification file.

```
genisys [OPTION] [VECTORS.xml_filepath] <Rosetta_filepath>
                                          <Test_Component_Name>

OPTION:
 -d : debug On (debug is Off by default)

Example : genisys -d ~/test_VECTORS.xml ../test.sld ADDER

        where, genisys          - executable script
               test_VECTORS.xml - VECTORS.xml File
               test.sld         - Rosetta Specification  File
               -d               - Enables the Debug Option
               ADDER            - Test Component
```

Figure 5.1: Command-line argument of the GENISYS tool

5.6 GENISYS Output XML files

There are various files generated as part of the output of the GENISYS tool. Two files are generated during the data path determination phase. These files contain information about the component dependence hierarchy. In an hierarchical fashion, they show which components drive the test component and which components are driven by the test component. Each layer in the hierarchy contains information about which parameter caused the component dependence link in that layer. One file contains the hierarchy of all the components driving the test component, while other contains the hierarchy of all the components driven by the test component. The root of the hierarchy in both the XML files is the test component. GENISYS tool also generates a file that contains information about system vectors containing the input and output parameters of the structural component. This file stores the values of system output and input parameters for all

the test vectors in the abstract test vectors file.

5.6.1 Driving Component Hierarchy

The *driving component hierarchy* contains information about all the components directly or indirectly driving the test component and the corresponding driving parameters in an XML hierarchical fashion. This hierarchy is also referred to as *top dependence hierarchy* as it contains hierarchy from the test component to the top-most components in the component dependence hierarchy. Figure 5.2 shows a sample XML file containing the driving component hierarchy.

```
<COMPONENT_HIERARCHY file="test/testgenisys">
   <TEST_COMPONENT component_name="COMP1">
      <DRIVING_COMPONENT component_name="COMP2"
                         driving_variable = "E">
         <DRIVING_COMPONENT component_name="COMP3"
                    driving_variable = "S">
         </DRIVING_COMPONENT>
         <DRIVING_COMPONENT component_name="COMP4"
                           driving_variable = "D">
            <DRIVING_COMPONENT component_name="COMP5"
                               driving_variable = "H">
            </DRIVING_COMPONENT>
         </DRIVING_COMPONENT>
      </DRIVING_COMPONENT>
   </TEST_COMPONENT>
</COMPONENT_HIERARCHY>
```

Figure 5.2: Top Dependence Hierarchy in XML

The root of the top dependence hierarchy in XML is `<COMPONENT_HIERARCHY>` element. It marks the beginning of the hierarchy. It has an attribute, `file`, whose value is the name of the Rosetta specification file. The root has one child element,

<TEST_COMPONENT>. This tag contains information about the test component. The attribute of this element, component_name, contains the name of the test component, $COMP_1$ in this example. The <TEST_COMPONENT> element can have zero or more children. If it has no children, no other component drives the test component.

<DRIVING_COMPONENT> element of the XML hierarchy specifies the components directly or indirectly driving the test component. This element has two attributes: component_name contains the name of the driving component and driving_variable indicates the name of the parameter forming this component dependence link. A component driving the test component may itself be driven by one or more components. This scenario is depicted in Figure 5.2 where component $COMP_2$ and $COMP_4$ drive the test component and are driven by components $COMP_3$ and $COMP_5$ respectively. Block diagram representation of the scenario shown in figure 5.2 is given in figure 5.3.

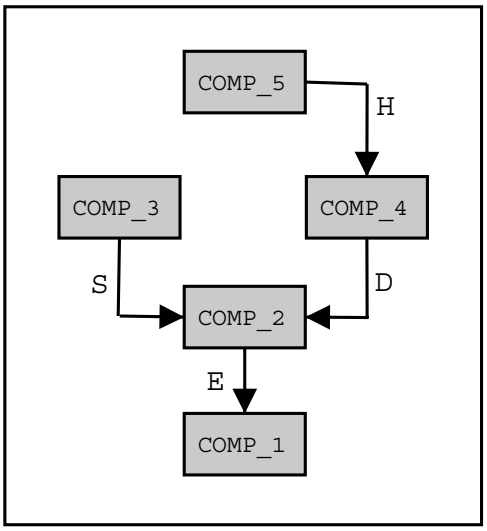

Figure 5.3: Top Dependence Hierarchy

5.6.2 Driven Component Hierarchy

Driven component hierarchy contains the components directly or indirectly driven by the test component along with the parameters driven by the test component. It is also referred to as *bottom dependence hierarchy* as it contains hierarchy from the test component to the bottom of the component dependence hierarchy. Figure 5.4 shows a sample XML file containing the bottom dependence hierarchy.

```
<COMPONENT_HIERARCHY file="test/testgenisys">
   <TEST_COMPONENT component_name="COMP1">
      <DRIVEN_COMPONENT component_name="COMP6" driven_variable="K">
         <DRIVEN_COMPONENT component_name="COMP7" driven_variable="J">
         </DRIVEN_COMPONENT>
         <DRIVEN_COMPONENT component_name="COMP8" driven_variable="L">
            <DRIVEN_COMPONENT component_name="COMP9"
                              driven_variable="M">
            </DRIVEN_COMPONENT>
         </DRIVEN_COMPONENT>
      </DRIVEN_COMPONENT>
   </TEST_COMPONENT>
</COMPONENT_HIERARCHY>
```

Figure 5.4: Bottom Dependence Hierarchy in XML

The bottom dependence hierarchy starts with `<COMPONENT_HIERARCHY>` element as the root. Like the top dependence hierarchy, it has an attribute, `file`, whose value is the name of the Rosetta specification file. The root has one child element, `<TEST_COMPONENT>`, which contains information about the test component. The attribute of this element, `component_name`, contains the name of the test component, $COMP_1$ in this example. As in the top dependence hierarchy, the `<TEST_COMPONENT>` element can have zero or more children. If it

has no children, no other component depends on the test component. Components directly or indirectly driving the test component are specified in the `<DRIVEN_COMPONENT>` element of the XML hierarchy. This element has two attributes: `component_name` contains the name of the component driven by the test component; and `driven_variable` contains the name of the parameter driven by the test component.

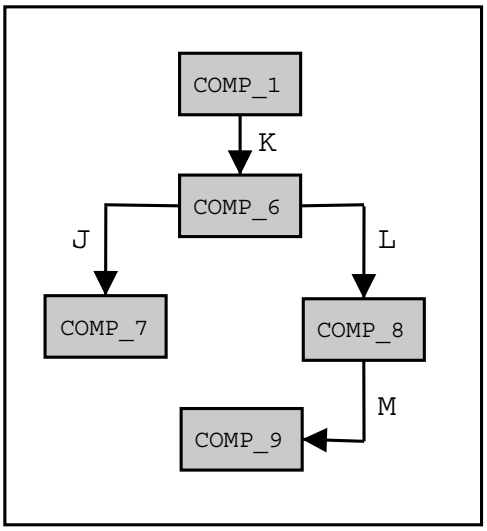

Figure 5.5: Bottom Dependence Hierarchy

A component driven by the test component may itself drive one or more components. Figures 5.4 and 5.5 show one such scenario both using the XML hierarchy and block diagram representation. As shown in the figures, component $COMP_6$, driven by the test component, drives components $COMP_7$ and $COMP_8$. Component $COMP_8$ further drives component $COMP_9$.

5.6.3 GENISYS System Vectors

The GENISYS tool generates system vectors that contain the input and output values of the structural component in XML format. The system vectors are generated for each test vector in the abstract test vectors for the test component. For each vector, the input values are the values obtained by inverting all the inner-components along the component dependence hierarchy in the structural component. Shown below is an example of system vectors produced by the component inversion engine.

```
<TestData>
   <Config>
      <DataConfig>
         <Name>input_state</Name>
         <Index>1</Index>
         <InputType/>
      </DataConfig>
      .     .     .
   </Config>
   <TestSet>
      <TestVector>
         <Input>
            <Name>input_state</Name>
            <Value>0</Value>
         </Input>
      </TestVector>
      .     .     .
   <TestSet>
</TestData>
```

The root element of the system vectors is `<TestData>`. It has two children: `<Config>` and `<TestSet>`. The former child contains all the interface parameters of the structural component. It has one or more instances of element `<DataConfig>`. This element contains name, index, and interface type of a given interface parameter. `<Name>` and `<Index>` respectively contain the name and

the index of the parameter; whereas the interface type is represented by either, <InputType>, <LocalVar>, or <OutputType> element. The <TestSet> element contains <TestVector>. Each <TestVector> contains three elements: <Input>, <LocalVar>, and <Output>. Each element contains <Name> and <Value>. They respectively contain the name and value of all the interface parameters. If the value of a given interface parameter is unknown, it is represented using "-".

Chapter 6

Testing and Examples

Thus far, the design and implementation of the GENISYS tool has been discussed exclusively. In this chapter, examples are used to illustrate the operation of the tool. Various components used in testing are explained with Rosetta specification in this chapter. Several scenarios used to test GENISYS are described. For each scenario, the outputs of the GENISYS tool - component dependence hierarchies and the system vectors - are discussed.

6.1 Rosetta Specification for various Components

This section will describe the inner-components that form the structural component. Each inner-component is a circuit performing a particular operations. These circuits include, a trigger-based circuit, an inverter, a positive trigger circuit, a negative trigger circuit, an OR gate, and a multiplexer. The Rosetta specification for these circuits is given in the following subsections.

6.1.1 Trigger-based Circuit

The Rosetta specification below describes a *trigger-based circuit*. The interface parameter list includes two input trigger bits, eight input parameters and eight output parameters. The two trigger bits drive the eight output parameters.

```
package TRIGGER_CIRCUIT :: static is
    /* package body begins here */
    facet TRIGGER_CIRCUIT
        (  /* interface parameters declared */
           cntl_bit1  :: input  bit; cntl_bit2  :: input  bit;
           ipt_sig1   :: input  bit; ipt_sig2   :: input  bit;
           ipt_sig3   :: input  bit; ipt_sig4   :: input  bit;
           ipt_state1 :: input  bit; ipt_state2 :: input  bit;
           ipt_state3 :: input  bit; ipt_state4 :: input  bit;
           opt_sig4   :: output bit; opt_sig3   :: output bit;
           opt_sig2   :: output bit; opt_state4 :: output bit;
           opt_state3 :: output bit; opt_state2 :: output bit;
           opt_state1 :: output bit;  opt_sig1  :: output bit
        ) :: state_based is
        /* facet body begins here */
        begin
        req1: if (%cntl_bit1)
                  then (opt_state4 = ipt_state4) and
                       (opt_state3 = ipt_state3) and
                       (opt_state2 = ipt_state2) and
       (opt_state1 = ipt_state1)
                  else (opt_state4 = 1) and
       (opt_state3 = 1) and
                       (opt_state2 = 1) and
       (opt_state1 = 1)
                  end if;
        req2: if (not(%cntl_bit2))
                  then (opt_sig4 = ipt_sig4) and
                       (opt_sig3 = ipt_sig3) and
                       (opt_sig2 = ipt_sig2) and
                       (opt_sig1 = ipt_sig1)
                  else (opt_sig4 = 1) and (opt_sig3 = 1) and
```

```
                    (opt_sig2 = 1) and (opt_sig1 = 1)
                end if;
    end facet TRIGGER_CIRCUIT;
end package TRIGGER_CIRCUIT;
```

6.1.2 Negative Trigger Circuit

The Rosetta specification given below describes a *negative trigger circuit*. There are two input parameters: a trigger bit and a input parameter value. Depending on the value of the trigger bit the output parameter is either assigned the input parameter value or it is set to 1.

```
package NEGATIVE_TRIGGER :: static is
    /* package body begins here */
    facet NEGATIVE_TRIGGER
        (  /* interface parameters declared */
           ipt_val :: input  bit;  trigger :: input bit;
           opt_val :: output bit
        ) :: state_based is
        /* facet body begins here */
        begin
        req1: if (not(%trigger))
                   then (opt_val = ipt_val)
                   else (opt_val = 1)
                   end if;
    end facet NEGATIVE_TRIGGER;
end package NEGATIVE_TRIGGER;
```

6.1.3 OR Gate Circuit

The Rosetta specification given below describes a simple *OR gate*. The two input parameters are logically *OR-ed* and the result is assigned to the output parameter.

```
package OR_GATE :: static is
    /* package body begins here */
    facet OR_GATE
        (  /* interface parameters declared */
           I0  :: input bit;  I1 :: input bit;
           OUT_PORT :: output bit
        ) :: state_based is
        /* facet body begins here */
        begin
        req1: OUT_PORT = ( I0 or I1 );
    end facet OR_GATE;
end package OR_GATE;
```

6.1.4 Inverter Circuit

Given below is the Rosetta specification for an *inverter circuit*. The input parameter value is toggled and assigned to the output parameter.

```
package INVERTER :: static is
    /* package body begins here */
    facet INVERTER
        (  /* interface parameters declared */
           in_port :: input bit; out_port :: output bit
        ) :: state_based is
        /* facet body begins here */
        begin
        req1: out_port = not(in_port);
    end facet INVERTER;
end package INVERTER;
```

6.1.5 Positive Trigger Circuit

The Rosetta specification given below shows a *positive trigger circuit*. It has three interface parameters: two input and one output parameter. The output parameter is either assigned the input parameter value or 1 depending on the value of the input trigger bit.

```
package POSITIVE_TRIGGER :: static is
    /* package body begins here */
    facet POSITIVE_TRIGGER
        (  /* interface parameters declared */
           ipt_val :: input bit; trigger :: input bit;
           opt_val :: output bit
        ) :: state_based is
        /* facet body begins here */
        begin
        req1: if (%trigger)
                  then (opt_val = ipt_val)
                  else (opt_val = 1)
                  end if;
    end facet POSITIVE_TRIGGER;
end package POSITIVE_TRIGGER;
```

6.1.6 Multiplexer Circuit

Shown below is the Rosetta specification of a *quadra 2 X 1 multiplexer circuit*. Depending on the value of the input signal bit, the four output parameters are assigned four of the eight input parameters.

```
package QUAD_MUX2X1 :: static is
    /* package body begins here */
    facet QUAD_MUX2X1
```

```
        (  /* interface parameters declared */
           Signal     :: input  bit; ipt_sig0   :: input  bit;
           ipt_sig1   :: input  bit; ipt_state1 :: input  bit;
           ipt_val0   :: input  bit; ipt_val1   :: input  bit;
           ipt_state0 :: input  bit; ipt_port0  :: input  bit;
           ipt_port1  :: input  bit; opt_sig    :: output bit;
           opt_state  :: output bit; opt_val    :: output bit;
           opt_port   :: output bit
        ) :: state_based is
        /* facet body begins here */
    begin
    req1:  if (%Signal)
               then (opt_sig = ipt_sig0) and
                    (opt_state = ipt_state0) and
                    (opt_val = ipt_val0) and
                    (opt_port = ipt_port0)
               else (opt_sig = ipt_sig1) and
                    (opt_state = ipt_state1) and
                    (opt_val = ipt_val1) and
                    (opt_port = ipt_port1)
               end if;
    end facet QUAD_MUX2X1;
end package QUAD_MUX2X1;
```

6.1.7 Rosetta Specification for the Structural Component

In the specification, all the inner-components are instantiated by supplying appropriate interface parameters to them. This results into one of a number of possible inter-dependence patterns between them. This inter-dependence results into the component dependence hierarchy. This specification illustrates a linear component dependence hierarchy where, `COMPONENT_1` drives `COMPONENT_2`, `COMPONENT_2` drives `COMPONENT_3`, and so on. However, GENISYS tool can resolve any possible

component dependence hierarchy formed in the structural component.

```
package STRUCT_COMPONENT :: logic is
    /* package body begins here */
    /* use the Rosetta Specification of inner components */
    use INVERTER, POSITIVE_TRIGGER, NEGATIVE_TRIGGER, QUAD_MUX2X1,
    OR_GATE, TRIGGER_CIRCUIT ;
    facet STRUCT_COMPONENT
        (  /* interface parameters declared */
           a  :: input  bit; B  :: input  bit; d  :: input  bit;
           h  :: input  bit; G  :: input  bit; x  :: input  bit;
           z  :: input  bit; W  :: input  bit; u  :: input  bit;
           Aa :: output bit; BB :: output bit; eE :: output bit;
           Ff :: output bit; DD :: output bit; Gg :: output bit;
           E  :: input  bit; Y  :: input  bit; V  :: input  bit;
           HH :: output bit; CC :: output bit; zZ:: output bit
        ) :: state_based is
        /* locally declared variables */
        C, F, I, J, K, L, M :: bit;

        /* facet body begins here */
        begin
        COMPONENT_1 : INVERTER(A, C);
        COMPONENT_2 : POSITIVE_TRIGGER(C, B, F);
        COMPONENT_3 : NEGATIVE_TRIGGER(F, D, I);
        COMPONENT_4 : OR_GATE(I, F, ZZ);
        COMPONENT_5 : QUAD_MUX2X1(A, B, C, D, E, F, G,
                                  H, I, J, K, L, M);
        COMPONENT_6 : TRIGGER_CIRCUIT(U,  V,  W,  X,  Y,  Z,
                                      J,  K,  L,  M,  AA, BB,
                                      CC, DD, EE, FF, gg, HH);
    end facet STRUCT_COMPONENT;
end package STRUCT_COMPONENT;
```

6.2 Test Scenarios

Given the Rosetta specification for various components in the previous section, several scenarios were used to test GENISYS. The output of GENISYS hugely depends on the test component and its position in the component dependence hierarchy. This section describes how various scenarios influence the GENISYS tool output and how GENISYS handles various invalid specifications.

6.2.1 Test Scenario-1

If `TRIGGER_CIRCUIT` is chosen as the test component, all inner-components directly or indirectly drive the test component. The two component dependence hierarchies formed as part of data-path determination phase are shown in the figures given below. Figure 6.1 shows the top component dependence hierarchy and figure 6.2 shows the bottom component dependence hierarchy.

As expected, the driving/top component dependence hierarchy contains all the component dependence links formed along with the interface parameters, while the driven/bottom component dependence hierarchy is empty, as the test component drives no other inner-component. Shown below is a snippet of the system vectors generated by GENISYS. In the system vectors given below, the value of the output parameter `ZZ` is unknown due to the fact that it is independent of the test component `TRIGGER_CIRCUIT`. The value of an unknown parameter is represented by a "`-`" in the system vectors.

```
<TestData>
```

```
<COMPONENT_HIERARCHY file="STRUCT_COMPONENT" >
  <TEST_COMPONENT component_name="COMPONENT_6" >
    <DRIVING_COMPONENT component_name="COMPONENT_5"
                       driving_variable="J" >
      <DRIVING_COMPONENT component_name="COMPONENT_1"
                         driving_variable="C" >
      </DRIVING_COMPONENT>
      <DRIVING_COMPONENT component_name="COMPONENT_2"
                         driving_variable="F" >
        <DRIVING_COMPONENT component_name="COMPONENT_1"
                           driving_variable="C" >
        </DRIVING_COMPONENT>
      </DRIVING_COMPONENT>
      .          .          .
   </DRIVING_COMPONENT>
  </DRIVING_COMPONENT>
 </TEST_COMPONENT>
</COMPONENT_HIERARCHY>
```

Figure 6.1: Driving Component Dependence Hierarchy for Test Scenario-1

```
<COMPONENT_HIERARCHY file="STRUCT_COMPONENT" >
  <TEST_COMPONENT component_name="COMPONENT_6">
  </TEST_COMPONENT>
</COMPONENT_HIERARCHY>
```

Figure 6.2: Driven Component Dependence Hierarchy for Test Scenario-1

```
  <Config>
    <DataConfig>
      <Name>A</Name>
      <Index>1</Index>
      <InputType/>
    </DataConfig>
    .    .    .
  </Config>
  <TestSet>
    <TestVector>
      <Input>
        <Name>A</Name>
        <Value>0</Value>
```

```
      </Input>
      .    .    .
      <LocalVar>
        <Name>C</Name>
        <Value>0</Value>
      </LocalVar>
      .    .    .
      <Output>
        <Name>ZZ</Name>
        <Value>-</Value>
      </Output>
    </TestVector>
    .    .    .
  </TestSet>
  .    .    .
</TestData>
```

6.2.2 Test Scenario-2

If `INVERTER` is chosen as the test component, all other inner-components are driven directly or indirectly by it. The component dependence hierarchies are shown in the figures given below. Figure 6.3 shows the driving component dependence hierarchy, which is empty as no other inner-component drives the test component. In figure 6.4, the driven component dependence hierarchy is shown. This hierarchy contains all the component dependence links as all the inner-components are driven by the test component.

A snippet of the system vectors generated by GENISYS for the test scenarios-2 are shown below. Since the test component is the top-most component in the component dependence hierarchy, it depends on no other inner-components. Hence

```
<COMPONENT_HIERARCHY file="testgenisys_all" >
  <TEST_COMPONENT component_name="COMPONENT_1" >
  </TEST_COMPONENT>
</COMPONENT_HIERARCHY>
```

Figure 6.3: Driving Component Dependence Hierarchy for Test Scenario-2

```
<COMPONENT_HIERARCHY file="testgenisys_all" >
  <TEST_COMPONENT component_name="COMPONENT_1" >
    <DRIVEN_COMPONENT component_name="COMPONENT_5"
                     driven_variable="C" >
      <DRIVEN_COMPONENT component_name="COMPONENT_6"
                        driven_variable="J" >
      </DRIVEN_COMPONENT>
      <DRIVEN_COMPONENT component_name="COMPONENT_6"
                        driven_variable="K" >
      </DRIVEN_COMPONENT>
      <DRIVEN_COMPONENT component_name="COMPONENT_6"
                        driven_variable="L" >
      </DRIVEN_COMPONENT>
     <DRIVEN_COMPONENT component_name="COMPONENT_6"
                      driven_variable="M" >
     </DRIVEN_COMPONENT>
   </DRIVEN_COMPONENT>
   .       .       .
  </TEST_COMPONENT>
</COMPONENT_HIERARCHY>
```

Figure 6.4: Driven Component Dependence Hierarchy for Test Scenario-2

the GENISYS component inversion engine does not invert any inner-component. This is apparent from the system vectors shown below, where the only known parameters are the interface parameters of the test component - `A` and `C`. All other parameters are unknown.

```
<TestData>
 <Config>
  <DataConfig>
   <Name>A</Name>
   <Index>1</Index>
```

```
   <InputType/>
  </DataConfig>
    .    .    .
 </Config>
 <TestSet>
  <TestVector>
   <Input>
    <Name>A</Name>
    <Value>0</Value>
   </Input>
   <Input>
    <Name>B</Name>
    <Value>-</Value>
   </Input>
   .    .    .
   <LocalVar>
    <Name>C</Name>
    <Value>1</Value>
   </LocalVar>
   .    .    .
  <Output>
    <Name>ZZ</Name>
    <Value>-</Value>
   </Output>
  </TestVector>
  .    .    .
 </TestSet>
</TestData>
```

6.2.3 Test Scenario-3

In the discussion thus far, scenarios with valid component dependence hierarchy have been considered. In this subsection, scenarios with invalid component dependence hierarchy are analyzed. The following discussion describes various invalid

specifications and how GENISYS handles such situations. Various issues including, non-determinacy, infinite feedback loop, and invalid interface parameters are discussed in this subsection.

Non-Determinacy in Hierarchy

As shown in the Rosetta specification below, the interface parameter F is driven by components, COMPONENT_2 and COMPONENT_4 simultaneously. This should result into termination of the component inversion process with appropriate error message.

```
package INVALID_CDH_1 :: logic is
    /* package body begins here */
    /* use the Rosetta Specification of inner components */
    use INVERTER, POSITIVE_TRIGGER, NEGATIVE_TRIGGER, OR_GATE;
    facet INVALID_CDH_1
        (  /* interface parameters declared */
           A  :: input  bit; B   :: input  bit;
           D  :: input  bit; OPT :: output bit
        ) :: state_based is
        /* locally declared variables */
        C, F, I :: bit;
        /* facet body begins here */
        begin
        COMPONENT_1 : INVERTER(A, C);
        COMPONENT_2 : POSITIVE_TRIGGER(C, B, F);
        COMPONENT_3 : NEGATIVE_TRIGGER(F, D, I);
        /* Non-Determinacy over the parameter F */
        COMPONENT_4 : OR_GATE(I, D, F);
        COMPONENT_6 : INVERTER(F, OPT);
    end facet INVALID_CDH_1;
end package INVALID_CDH_1;
```

Executing the GENISYS tool over the structural component given above, will terminate the process with the following error message as expected.

```
Invalid Specification! Non Determinacy found over:
VARIABLE  :  F
COMPONENT : COMPONENT_2 AND COMPONENT_4
```

Infinite Feedback Loop in Hierarchy

Consider the Rosetta specification for a structural component given below. As shown in the specification, an infinite feedback loop exists between the components, COMPONENT_2, COMPONENT_3, and COMPONENT_4.

```
package INVALID_CDH_2 :: logic is
    /* package body begins here */
    /* use the Rosetta Specification of inner components */
    use INVERTER, POSITIVE_TRIGGER, NEGATIVE_TRIGGER, OR_GATE;
    facet INVALID_CDH_2
        (  /* interface parameters declared */
           A  :: input  bit; B   :: input  bit;
           D  :: input  bit; OPT :: output bit
        ) :: state_based is
        /* locally declared variables */
        C, F, I :: bit;
        /* facet body begins here */
        begin
        COMPONENT_1 : INVERTER(A, C);
        COMPONENT_2 : POSITIVE_TRIGGER(C, B, F);
        COMPONENT_3 : INVERTER(F, I);
        /* Infinite Feedback loop over components
         * COMPONENT_2, COMPONENT_3, & COMPONENT_4
         */
        COMPONENT_4 : NEGATIVE_TRIGGER(I, D, B);
        COMPONENT_5 : OR_GATE(I, D, opt);
    end facet INVALID_CDH_2;
end package INVALID_CDH_2;
```

The GENISYS tool handle the feedback loop by breaking the last link that completes the loop, as shown in the output below.

```
Infinite Feedback Loop Found! Loop exists between following components:
COMPONENT_2
COMPONENT_3
COMPONENT_4
Ignoring the following Link:
COMPONENT_3 drives COMPONENT_4 through parameter "I"
```

Invalid Interface Parameter

In the Rosetta specification given below, the component COMPONENT_2 takes the system output parameter, OPT, as one of its inputs.

```
package INVALID_CDH_3 :: logic is
    /* package body begins here */
    /* use the Rosetta Specification of inner components */
    use INVERTER, POSITIVE_TRIGGER, NEGATIVE_TRIGGER, OR_GATE;
    facet INVALID_CDH_3
        (  /* interface parameters declared */
           A  :: input  bit; B   :: input  bit;
           D  :: input  bit; OPT :: output bit
        ) :: state_based is
        /* locally declared variables */
        C, F, I :: bit;
        /* facet body begins here */
        begin
        COMPONENT_1 : INVERTER(A, C);
        /* System output parameter can't be inner
         * component's input parameter
         */
        COMPONENT_2 : POSITIVE_TRIGGER(opt, B, F);
        COMPONENT_3 : INVERTER(F, I);
        COMPONENT_4 : NEGATIVE_TRIGGER(I, D, OPT);
```

```
    end facet INVALID_CDH_3;
end package INVALID_CDH_3;
```

This results into termination of the component inversion process with an error message given below.

```
Execution of GENISYS tool terminated
Error: System Output: OPT cannot be input to Component: COMPONENT_2.
```

In addition to the above mentioned invalid specifications, GENISYS generates an error message when:

- the command-line argument is invalid or incomplete.
- an output parameter of a component to be inverted is unknown.
- the input files do not exist or are empty.
- an error occurs while parsing the ROM or the XML file.
- a parameter is not found, or has invalid interface mode.

The test scenarios described in this chapter cover various cases and show how GENISYS operates in these situations. Cases ranging from the one where the system has valid configurations and the sytem vectors are generated as expected to the one where system configuration is invalid and causes the GENISYS process to terminate with appropriate error message are described. One of the test scenarios had a system output parameter as an input to one of the inner-component. This resulted into termination of the component inversion process and generated an error message stating that system output cannot be input to an inner-component

as expected. Other scenarios include, infinite feedback loop and non-determinacy in the component dependence hierarchy. While processing the component dependence hierarchy, if an infinite feedback loop is detected, the loop is broken by removing the link among the two inner-components that forms the loop and the component inversion process is resumed. Non-determinacy occurs in the hierarchy if two inner-components drive the same parameter. This scenarios cannot be resolved by GENISYS and hence results into termination of the process with an error message.

Chapter 7

Related Work

Component inversion, or program inversion as it is widely known, has been mentioned at various places as a programming technique. Program inversion first appeared in [23]. An informal argument is presented explaining how the inverted program undoes the effect of the forward program, thus restoring all the old values. In other word, program inversion means for a given forward-directed program S constructing a program S^{-1} that works like the reverse of S. In this chapter, various work done in the field of program inversion are highlighed.

In the paper, *Running Programs Backwards: The Logical Inversion of Imperative Computation* [31], the author explores the feasibility of inverting imperative computations using logic programming technology. The declarative semantics of program relations implicitly denote both forward and inverse computations. This view of computation has practical applications when logic programming technology is considered. An imperative program has a logic program derived for it which

abductively describes its inverted behavior. The paper shows how a number of nontrivial imperative computations can be inverted with minimal logic programming tools. The advantage of this approach is that non-deterministic inversions are possible, which permits sets of inputs to be computed for a particular output.

The paper, *A formal approach to program inversion* [21], introduces a formal approach to inverting programs. The usefulness of this formal approach in programming is demonstrated by applying it to develop an in-place algorithm for the LU-multiplication, which in other case might be hard to find.

Author of the paper, *Program inversion in the refinement calculus* [32], presents a calculational method for inverting programs by inverting the components separately by using assertions as commands and by permitting constructs that exhibit angelic nondeterminism. The author also gives examples to illustrate this method. The paper also contains rules to transform inverted program so that the angelic constructs are removed.

In the paper, *Program Inversion* [20], the author presents a method for producing the inversion of a program. In this paper, it is shown that inversion establishes a natural relationship between various well known programs like, sorting and permutation generation programs. The author introduces basic concept of automata, program, function, and relation. The paper initially shows methods for inversion of statements like, if statements, and while loops. Later on, inversion of entire programs like, sorting, and permutation generation is described. The author has used Pascal language to illustrate examples throughout the paper.

The paper, *Reverse Execution of Programs* [19], describe situations where inverse of a statement is very useful while debugging a program. The authors then introduce a new concept of a debugger called BDb, or Bi-directional Debugger, that supports statement inversion. BDb provides an option to the user to execute the program in either forward or backward direction. This debugger is developed for programs in C language, however similar effect can be obtained in other programming languages too. Inversion of various types of statements including, assignment, selection, iterative and unstructured statements have been described with examples in this paper.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis, we have developed a tool named GENISYS that performs the process of component inversion in a component dependence hierarchy. This process is performed with respect to a given test component (TC) in the hierarchy. The process is divided into two sub-processes: the data-path generation; and the component inversion. The data-path determination phase generates the data-paths along the component dependence hierarchy. This phase produces two component dependence hierarchies traversing up and down from TC. The component inversion phase performs component inversion over the invertible components along the hierarchy and generates system vectors containing the inputs and outputs of the structural component. GENISYS takes in the Rosetta specifications of all the components involved in the hierarchy, abstract test vectors generated for TC,

and the TC as inputs and performs the component inversion. The output of the GENISYS tool is stored in XML format, since XML is getting widely deployed in the industry as standard way for data representation. All these files store data in XML format.

This thesis presents the first version of the GENISYS tool. GENISYS has been successfully tested over various test scenarios described in chapter 7. Scenarios with both, valid and invalid Rosetta specifications were used. GENISYS produced valid system test vectors for system with valid specifications and reported the invalid specifications with appropriate error messages. Each system test vector generated by GENISYS contains a set of system inputs. These inputs will generate the desired system outputs after forward execution of the program. Thus, using the Rosetta specification of the structural and inner-components and the test vectors of TC, GENISYS is able to generate valid system inputs. These inputs describe the initial state the system has to be in to reach the current state. Although being still in its development stage, GENISYS seems to be a promising tool for complex component inversions.

8.2 Future Work

This is the first version of the GENISYS tool. This version does not implement all the features of the Rosetta. There are several enhancements possible to this tool. Here is a listing of some of the features that can be added to GENISYS.

1. Allowing each inner component in the structural component to be a structural component in itself will increase the usability of the tool.

2. Currently all the inner components are not defined in the structural component. Supporting the definition of a inner component in the structural component will make the specifications more readable.

3. Function inversion is not currently supported. Allowing functions in the specification and performing function inversion is a future goal.

4. The algorithm used to invert a component can be made more sophisticated. For example, the SAT solver produces one of a number of valid assignments possible to an unknown parameter. Later on, this assignment may result into an unsatisfiable expression. Such a situation can be avoided by maintaining a pointer to location where an assumption was made regarding the value of a given parameter and whenever an unsatisfiable expression is found, the control should roll back to the location where the assumption was made and a new assumption should be made for the parameter. The down-side of this new approach is that it is computation-intensive as every time the control rolls back the component inversion process will restart from the location where the assumption was made.

Bibliography

[1] Conjunctive Normal Form(CNF). World Wide Web:

http://www.enm.bris.ac.uk/research/aigroup/enjl/logic/sld005.htm.

[2] Convert Sentence to Conjunctive Normal Form. World Wide Web:

http://www.cs.yale.edu/homes/cc392/node5.html.

[3] DOM - XML Parser. World Wide Web: http://www.w3schools.com/dom.

[4] Fu, Zhaohui. World Wide Web: http://ee.princeton.edu/ zfu/.

[5] GRASP SAT Solver. World Wide Web: http://sat.inesc-id.pt/ jpms/grasp.

[6] HTML Tutorial. World Wide Web: http://www.w3schools.com/html/.

[7] Insight Debugger. World Wide Web: http://sources.redhat.com/insight/.

[8] JDB - The Java Debugger. World Wide Web:

http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html.

[9] SATO SAT Solver. World Wide Web:

http://www.cs.uiowa.edu/hzhang/sato.html.

[10] The GNU Project Debugger. World Wide Web:
http://sources.redhat.com/gdb/.

[11] VectorGen - Automated Test Generation, EDAptive Computing, Inc. World Wide Web: http://www.edaptive.com/products/vectorgen/index.htm.

[12] Walksat SAT Solver. World Wide Web:
http://www.cs.washington.edu/homes/kautz/walksat.

[13] XML Schema Tutorial. World Wide Web:
http://www.w3schools.com/schema/default.asp.

[14] XML Specifications. World Wide Web: http://www.w3schools.com/xml.

[15] zChaff SAT Solver. World Wide Web:
http://www.ee.princeton.edu/ chaff/zchaff.php.

[16] Srinivas Akkipeddi. Advanced test vector generation from rosetta. Master's thesis, University of Kansas, 2001.

[17] Perry Alexander. Details of Rosetta. World Wide Web:
http://www.sldl.org.

[18] Perry Alexander, David Barton, Cindy Kong, and Catherine Menon. The rosetta strawman. Technical report, The University of Kansas, 2002.

[19] Bitan Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Not.*, 34(4):61–69, 1999.

[20] Rommert Casimir. Program Inversion. World Wide Web: citeseer.nj.nec.com/casimir80program.html.

[21] Wei Chen. A formal approach to program inversion. In *Proceedings of the 1990 ACM annual conference on Cooperation*, pages 398–403. ACM Press, 1990.

[22] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[23] Edsger W. Dijkstra. Program Inversion. *Selected Writings on Computing: A Personal Perspective*, pages 351–354, 1982.

[24] David Harel. And/Or Programs: A New Approach to Structured Programming. *ACM Trans. Program. Lang. Syst.*, 2(1):1–17, 1980.

[25] M. Harrold, D. Liang, and S. Sinha. An approach to analyzing and testing component-based systems, 1999.

[26] Murthy Kakarlamudi. Automatic Test Vector Generation in XML from Rosetta Specifications. Master's thesis, University of Kansas, 2002.

[27] Trish LeBlanc. Schmidt Trigger Circuit. World Wide Web: http://www.saintjohn.nbcc.nb.ca/ dei/SCHMTTR.HTM.

[28] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[29] R. Pressman. *Software Engineering: A Practicioner's Approach.* McGraw Hill, New York, NY, third edition, 1992.

[30] Krishna Ranganathan. DVTG, Design Verification Test Generation from Rosetta Specifications. Master's thesis, University of Cincinnati, 2001.

[31] Brian J. Ross. Running Programs Backwards: The Logical Inversion of Imperative Computation. *Formal Aspects of Computing*, 9(3):331–348, 1997.

[32] Joakim von Wright. Program Inversion in the Refinement Calculus. *Information Processing Letters*, 37(2):95–100, 1991.

[33] Yinlei Yu. How to Use/Hack zChaff SAT Solver? Lecture Notes.

[34] Kalpesh Zinjuwadia and Perry Alexander. DVTG and Test Harnessing using Rosetta Specifications. In *Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-based Systems ECBS'04*, pages 136–144, Brno, Czech Republic, May 24-27 2004.