Automatic Test Knowledge Extraction From VHDL (ATKET*)

Praveen Vishakantaiah, Jacob Abraham
Computer Engineering Research Center
University of Texas at Austin
Austin, TX 78758

Magdy Abadir
Microelectronics and
Computer Technology Corporation
Austin, TX 78759

Abstract

Behavioral information about modules in a design is exploited by test engineers to handle the complex problem of test generation for large designs. However, automation of the process of exploiting behavioral information to facilitate test generation is in its infancy. This paper introduces ATKET, an Automatic Test Knowledge Extraction Tool which automatically generates pieces of test knowledge by using structural and behavioral information in the VHDL description of a design. Results obtained from ATKET for a circuit which is difficult to test are presented.

1 Introduction

The emergence of Computer Aided Design technology in the past decade has facilitated the design of highly dense and complex integrated circuits. There has been a continued effort to introduce new test methodologies which not only would refrain from increasing product cost or degrading performance, but also keep pace with the design technology and reduce the burden of testing on the design engineer. Present-day test engineers exploit the behavior of modules in the design while deriving tests manually for large circuits using tools capable of generating tests for small circuits.

Sarfert et al. [9], Roy et al. [8] and Kunda et al. [6] propose methodologies to speed up the test generation process by using high level primitives. These techniques lack intelligence to combine module knowledge at a global level to generate a priori knowledge which can guide test generation process.

Work by Bending [3] requires the user to provide knowledge to guide test generation while the work by Crastes de Paulet [5], Shirley [10] and Thearling [11] use some form of knowledge extraction. Work by Crastes de Paulet is restricted to generating accessibility modes for each chip by using structural descriptions. Shirley's work is restricted to data paths. Thearling's work is aimed at designs developed using a silicon compiler. Also, none of the above techniques exploit VHDL behavioral descriptions.

ExperTest [7] uses a subset of VHDL for its work. Details of techniques used by ExperTest to generate test knowledge are not available.

2 Research motivation

Automatic test generation for large sequential circuits targeted at gate or switch level faults is very expensive. This is because of the large search space that needs to be considered during test generation process. Knowledge about the behavior of modules in the design can be exploited to prune the search space [4] so as to make test generation feasible while still targeting faults at any desired level of abstraction.

Consider the hierarchical module example shown in Figure 1. While targeting a fault inside module O, a typical automatic test generator does not take into consideration the constraints due to modules X, Z, P and Q. Instead, justification is done through these modules whenever choices are available on the lines connecting module O with X, Z, P and Q. This is a very expensive procedure and can be...
eliminated by identifying justification and propagation paths through these modules in a preprocessing step and storing the knowledge.

This paper describes ATKET, an Automatic Test Knowledge Extraction Tool which synthesizes test knowledge using structural and behavioral information available in the VHDL description of a design. ATKET can be used in conjunction with tools like TIGER [1] and thereby eliminate the need for the designer to provide transparent modes of operation for modules.

A VHDL analyzer (available from Microelectronics and Computer Technology Corporation [MCC]) produces an intermediate representation of the information contained in a VHDL design. ATKET interfaces to this intermediate representation to access structural and behavioral information in the design and stores it in suitable data structures. A convenient representation called the Module Operation Tree (MOT) is used to capture the behavior of modules in the design. Information stored in the MOT along with structural information describing connections between modules in the design is used to generate test knowledge.

Section 3 describes extraction of information from the intermediate representation and introduces the concept of a MOT. Section 4 deals with the types of test knowledge and techniques used to generate them. Section 5 discusses an example circuit and results obtained using ATKET. The concluding remarks are given in Section 6.

3 Information extraction

Structural and behavioral information of the circuit described in VHDL is obtained by accessing the intermediate representation produced by MCC's VHDL analyzer.

3.1 Module operation tree

A Module Operation Tree (MOT) is used to represent behavioral information of a module internally. The MOT is a convenient representation which facilitates generation of pieces of test knowledge for the modules in the design. It ensures that only the designer specified set of operations are used during high-level justification and propagation. It also captures the sequentiality of statements within the VHDL process statement.

The MOT is a binary tree in which the nodes correspond to VHDL conditional statements. Each node has a left child which corresponds to the conditional statement at the node being false and a right child which corresponds to the conditional statement at the node being true. The nodes in the MOT are categorized into two types depending upon the conditional statement in the conditional constructs. The first type is one in which the conditional statement involves a port/signal/variable of width 1 (number of bits is 1) whose binary value decides module behavior. The \( C_{RST} = 1 \) node in Figure 2 is an example of this type of node. The second type is one in which the conditional statement involves a port/signal/variable of width greater than 1 and validity of the statement decides module behavior. The node containing the statement \( COUT = 500 \) in Figure 2 is an example of this type of node.

The nodes of MOT may contain a list of statements which correspond to statements at the same level as the conditional statement. Storage of such a list at MOT nodes is necessary since these are applicable to each of the subsequent nodes that are generated. For example, in Figure 3, the statement, \( C \leftarrow A + B \) applies to both leaf nodes which means that if the condition at the node is false, \( C \leftarrow A + B \) as well as Ovf is set to '0'. If the condition is true, \( C \leftarrow A + B \) as well as Ovf is set to '1'.

![Figure 2: Description of a Counter and its MOT](image)

Leaf nodes in the MOT contain a list of statements which correspond to the input-output relationships defined for the module by the designer. The list may contain only one statement as is the case in Figures 2 and 3.

Depending on the module input-output relationship, edges in the MOT are categorized so that information about leaf nodes is readily available in the MOT. The basic categories are:
1. Initialization Edge (IE), which will lead to a leaf corresponding to a module output being initialized to a constant value. The right edge of node C.RST = 1 in Figure 2 is an example.

2. Propagation Edge (PE), which will lead to a leaf corresponding to an identity mode \[2\] of the module. The right edge of node LD = 1 in Figure 2 is an example.

3. Hold Edge (HE), which will lead to a leaf corresponding to a module output that holds the previous value. The left edge of node CLK = 1 in Figure 2 is an example.

4. State Edge (SE), which will lead to a leaf corresponding to a module output which changes its value depending on its previous value. The right edge of node COUT <= 3000 in Figure 2 is an example.

<table>
<thead>
<tr>
<th>RHS of Statement</th>
<th>$E_b$</th>
<th>$E_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant value (zero or any specific constant)</td>
<td>X</td>
<td>III</td>
</tr>
<tr>
<td>Single port and not an expression</td>
<td>X</td>
<td>PE</td>
</tr>
<tr>
<td>Port is a module IP port</td>
<td>X</td>
<td>P</td>
</tr>
<tr>
<td>Port is same as LHS</td>
<td>Q</td>
<td>P</td>
</tr>
<tr>
<td>Adjoint one port in same as LHS</td>
<td>X</td>
<td>Q</td>
</tr>
<tr>
<td>All ports are module input ports</td>
<td>Q</td>
<td>Q</td>
</tr>
<tr>
<td>Does not contain any port</td>
<td>Q</td>
<td>Q + III</td>
</tr>
</tbody>
</table>

$E_b$ — Edge type after analyzing statement; $E_a$ — Edge type before analyzing statement; X — Edge with no type; Q — Edge is of some type 'Q'.

Figure 4: Rules to categorize edges pointing to leaf nodes

A combination of these basic categories can also be used to categorize edges in the MOT. Each statement in the leaf node is analyzed using the rules shown in Figure 4 to decide the type of an edge pointing to the leaf node. The type of an edge which does not point to a leaf node is the union of edge types of the children edges of the node to which the edge points. For example, the type of left edge of node LD = 1 in Figure 2 is ISE (which means Initialization and State Edge) because the node COUT = 500 to which it is connected has a left edge whose type is SE and a right edge whose type is IE.

The MOT is built for every process statement in the behavioral description of a module. This preserves the concurrency of the process statement. The VHDL conditional constructs, if-then-else-if-and case statements within the process statement are analyzed to generate the MOT. Nested conditional constructs are handled.

After extracting all of the necessary information from the intermediate representation, compiler optimization techniques, namely copy and constant propagation and dead code elimination, are used to compact information stored in the MOT.

Signal/variable assignment statements (referred to as statements from now on) at MOT nodes may contain internal signals (signals and variables that are not module ports). Such signals are eliminated to ensure that test knowledge is in terms of module ports and does not contain any information related to temporary signals or variables used in the design description. Relevant signal relationships between module ports and internal signals in the MOT nodes are used for the elimination process.

4 Test knowledge generation

Test generation for sequential circuits (especially targeted at switch level faults) can be very complex, time consuming and even impractical for large circuits due to line justification and error propagation involving multiple time frames. Test knowledge which guides line justification and error propagation by pruning the search space is obtained using information stored in the MOT in conjunction with structural connections between various modules in the design. The different pieces of test knowledge generated using the MOT for each module in the design are:

1. Initialization Modes which enable initialization of any module output to zero or to constant values.
2. Propagation Modes which correspond to identity modes of the module.

3. Hold Modes which correspond to any module output retaining its previous state.

4. State Modes in which the new state of any module output depends on its previous state.

The information corresponding to each mode generated contains a set of constraints on the module inputs which needs to be satisfied in each time frame and the action of the mode.

Modes are directly extracted from the MOT using the algorithm shown in Figure 5. If the modes contain constraints on module outputs, they are eliminated by using the algorithm shown in Figure 6. This ensures that the constraints are restricted to module inputs.

```plaintext
find_fund_modes(MOT, constraint_list) {
    if (statement list at the node is not empty)
        for (each statement in the statement list)
            Add modes, if any to the relevant mode list
        endfor
    endif
    if (left child is a leaf node)
        Generate modes using constraint_list
    else
        constraint_list.append(node-constraint)
        find_fund_modes(MOT_left, constraint_list)
    endif
    if (right child is a leaf node)
        Generate modes using constraint_list
    else
        constraint_list.append(node-constraint)
        find_fund_modes(MOT_right, constraint_list)
    endif
} end find_fund_modes()
```

Figure 5: Algorithm to find fundamental modes

```plaintext
find_advanced_modes(MOT, fund_modes) {
    for (each fund_mode, m_i)
        if (constraints involve module outputs)
            Search MOT to eliminate constraint on module output
            if (such a mode, m_j, is found)
                Combine m_i and m_j to update m_i
            else
                Unrealizable mode
            endif
        endif
    endfor
} end find_advanced_modes()
```

Figure 6: Algorithm to find advanced modes

corresponds to A being zero and B being a symbolic value.

4.2 Recognition of Shift operation

VHDL does not provide an operator to specify the Shift operation of modules and hence cannot be directly recognized. ATKET identifies the Shift operation (serial-serial, serial-parallel, and parallel-serial) automatically by using statements in each block of the VHDL process statement. For example, the technique used to identify the serial-parallel Shift operation is described below.

Let IM be an input vector of module M and OM be an output vector of M. Let 'b' and 'd' represent two specific bit positions of OM and 'c' represent a specific bit position of IM. If there is a statement OM[b] = OM[b] followed by a statement, OM[b] = IM[c] in the same block (indicates same time frame), a serial-parallel Shift operation is recognized.

The information extracted about the Shift operation is whether the Shift is to the left or to the right and the number of bits which can be shifted in from the module input without losing any bits.

4.3 Combining module modes

For each module M in the design, the modes of each of the subcomponents in M are combined to generate useful test knowledge relevant to module M.

The information used to combine the modes of modules in the design include the structural connectivity between modules, existence of feedback loops in
the circuit, constraints on module inputs due to some signals being permanently tied to a fixed value, and constraints on outputs of modules connected to inputs of other modules.

5 Test knowledge example

Test knowledge for the example circuit [7] shown in Figure 7 was automatically generated in about 1.0 CPU second on a SUN 4/110. The research prototype of ATKET has been implemented in C++ (about 7000 lines of source code) using object-oriented techniques.

The behavior of the Counter and Adder shown in Figure 7 are described in Figures 2 and 3. The Reset of the Counter is connected to the Overflow of the Adder. REG.SLT controls the output of the Register on the positive edge of CLK. A '0' value of REG.SLT results in an hold operation for the Register while a '1' value results in a latch operation. RST initializes the output of the Register to zero.

![Figure 7: Test knowledge example](image)

The modes generated for the Counter and Adder are shown in Figures 8 and 9. The modes generated for EG.CKT are shown in Figure 10. Constraints shown between two bold horizontal lines correspond to a mode. For example, in Figure 8, there are two initialization modes each spanning six time frames. The first initialization mode requires that the constraint \(CLK = 0\) be satisfied in the first time frame, set of constraints \(CLK = 1\), \(LD = 1\), \(IN = 0\), \(ADD.IN = 0\) and \(RST = 1\) be satisfied in the second time frame and so on.

It should be noted that each of the modes in Figure 8 correspond to the seven leaf nodes in the MOT shown in Figure 2. In Figure 9, the first two initialization modes correspond to the two leaf nodes in the MOT shown in Figure 3 while the propagation mode corresponds to the default mode associated with the '+' operator in the statement at the node of the MOT in Figure 3. The state and hold modes of EG.CKT are not shown in Figure 10 due to space considerations.

For another example, the EG.CKT in Figure 7 was modified such that the 4 bit wide ADD.IN port is permanently tied to GND. Only one interesting mode which is automatically generated for this case is shown in Figure 11. This mode is generated as a result of identifying that the realization of constraint, \(ADD.IN > 0\) in the second initialization mode of the

![Figure 9: Modes generated for the Adder](image)
Adder (shown in Figure 9) is not possible. The feedback loop in the circuit is exploited to generate the initialization mode for the Adder.

5.1 Using generated test knowledge

The use of test knowledge generated by ATKET can be demonstrated by considering two scenarios. One is when the automatic test generator is generating test patterns for a module, say M whose input is connected to the output of a module like the EG.CKT module. In this case, the modes in Figure 10 can be used to avoid backtracking through the EG.CKT module, thereby speeding up the test generation process.

The second scenario is when the automatic test generator is targeting faults inside a module, say the Register module in Figure 7. In this case, the Adder modes (since this is the only module that is connected directly to the Register) shown in Figure 9 can be used to speed up the test generation process.

6 Conclusions

Techniques to automatically generate test knowledge from structural and behavioral information in the VHDL description of a design have been introduced. The results obtained from a research prototype of the test knowledge extraction tool for an example circuit which is difficult to test were provided. The tool took only about 1.0 CPU second for the example circuit which indicates the ease with which test knowledge was generated.

We are in the process of extracting test knowledge from additional VHDL constructs. Also, techniques to generate more complicated test knowledge are being identified.

Acknowledgements

Interesting discussions with Desmond D’Souza are acknowledged. This research was supported in part by the Semiconductor Research Corporation under contract 91-DP-142.

References