Search-Space Optimizations for High-Level ATPG

Jorge Campos and Hussain Al-Asaad
Department of Electrical and Computer Engineering
University of California, Davis, CA
E-mail: {jcampos, halasaad} @ece.ucdavis.edu

Abstract

Our mutation based validation paradigm (MVP) is a validation environment for high-level microprocessor implementations. To be able to efficiently identify and analyze the architectural states (prospect states) that can possibly satisfy a set of constraints during MVP's test generation, we need to reduce the search space in the analysis process as early as possible. In this paper, we present some optimizations in the search space that speed up the overall test generation process.

1. Introduction

Complete high-level microprocessor implementations have a large design space, often too large for modern deterministic automatic test pattern generation (ATPG) methods. In our previous research [1], we have defined a method to generate highly effective input stimuli by targeting the set of constraints that potentially exposes the largest number of modeled errors for any given ATPG iteration; this method, however, requires deterministic ATPG. Common high-level ATPG methods do not always perform deterministic ATPG [2][3], and the ones that do rely on first synthesizing portions of the high-level microprocessor implementation [4]. These methods are not efficient or effective enough because they circumvent the task of generating one test sequence for one set of constraints directly from the high-level information, a requirement when generating input stimuli that satisfy the most effective ATPG goals at any given time frame.

In our previous research [1][5][6], we have developed the methods that implement MVP (Mutation-based Validation Paradigm), our validation environment for high-level microprocessor implementations. MVP is able to handle complete implementations because it only uses high-level information, and only uses the hardware description language (HDL) information relevant to the set of constraints when identifying all relevant architectural states. We are defining an architectural state of the circuit that satisfies the set of constraints under consideration as a *prospect state*.

Generating input stimuli that satisfy a set of constraints requires the solver to identify all prospect states for each time frame, and eliminate the prospect states that can not be used to produce a test sequence. This is a problem for modern superscalar microprocessor implementations because of their inherently large state space. Therefore to be able to efficiently identify and analyze the architectural states (*prospect states*) that can possibly satisfy the set of constraints, we need to reduce the search space in the analysis process as early as possible.

MVP relies heavily on its basic algorithm that extracts and solves prospect states when satisfying constraints, when identifying which signals act as registers, and when identifying the finite state machine (FSM) for any given register. It is therefore important that prospect states are extracted and solved as efficiently as possible, because doing so optimizes MVP as a whole. Section 2 discusses some relevant background information. Section 3 describes the limitation of our initial algorithm for solving a prospect state, which was the motivation for the work presented in this paper. Section 4 describes the test circuit developed for analyzing the performance-limiting scenario, and Section 5 describes the methods used to optimize our solver. Finally, the results are discussed in Section 6.

2. Background

For our validation paradigm [6], we have created the notion of a prospect state to define an architectural state in terms of a set of constraints. For each constraint, there is a data dependency of its corresponding signal to the set of primary inputs and internal registers, and there is a set of control requirements that allow that data dependency to become active.

For a prospect state generated from multiple constraints, a non-false control requirement signifies that an architectural state exists which may be able to satisfy the set of constraints. Furthermore, a non-false set of data dependencies means that a solution for that architectural state has been generated.

The data dependencies and control requirements of a prospect state are each represented as independent data dependency graphs (DDGs). Each DDG has a tree structure with the following sequence of layers starting at the root node: Boolean OR operators, Boolean AND operators, relational operators, computational operators, and literals/identifiers. Given that Boolean OR operators are always at a layer above Boolean AND operators, a DDG in a prospect state has a disjunction of conjunctions structure.

All DDG nodes operate on an explicit range of values, which we are implementing using our `ValueRange` derived classes. Identifiers initially hold the range of values specified by their corresponding type definition, and literals hold the exact value specified by the hardware description. Given that the root DDG node returns a Boolean range of values, a DDG is solved by forcing a true value onto the root node. The solver algorithm reduces the range of values returned from the root node to true by reducing the range of values of all identifiers in the DDG accordingly. An identifier found to have an invalid range of values after solving a DDG signifies that its statement (rooted by an ancestor relational operator) contradicts with a statement elsewhere within the same conjunction (rooted by an ancestor Boolean AND operator). The range of values in that conjunction therefore cannot be satisfied.

3. Problem Definition

Each prospect state is defined by two components: (*i*) the data dependency of its constraints onto all registers and primary inputs of the circuit under test, and (*ii*) the control requirements that allow for that prospect state to become active. Each of these components is implemented as a DDG, therefore both components can be solved by using the same algorithm.

Prospect states are generated and solved throughout the ATPG process, making the algorithms that restructure and solve the DDGs to be MVP's limiting factor. Therefore optimizing the worst-case scenario for the algorithm that restructures and solves a DDG will have a significant impact on MVP's overall performance.

Case statements are commonly used in hardware descriptions to describe the functionality of an FSM. Therefore it is expected that some case statements in a hardware description will be significantly large. Also, it is expected that some signals (in particular, control signals) will have a separate assignment statement within each block of the case statement. These large case statements will be the limiting factor for MVP's performance because such a signal will require the corresponding assignment statement and control requirements to be analyzed for every block in the case statement.

If a case statement contains a `"when others =>"` block, its control requirements (guard) will be the conjunction of the negated guards of all explicit cases. This block's control requirements will therefore be a conjunction of disjoining operators, where a disjoining operator is an operator that imposes a disjoint range of values onto any identifier operand. Our goal is to convert this graph into a disjunction of conjunctions such that each conjunction defines a contiguous range of values for all discrete identifiers within it, therefore we must restructure the DDG into an equivalent graph that is free of disjoining operators.

Restructuring the graph into our desired form forces us to recursively replace each sub-tree rooted at a disjoining operator with an equivalent tree that is free of disjoining operators, but is bigger in size. An inequality operator $A \neq B$ is replaced by a disjunction of relational operators $((A < B) \text{ OR } (A > B))$, which unfortunately is a complete tree with twice the number of leaf nodes than the original. The size complexity is exacerbated by the modified graph's conjunction of disjunctions structure. Performing a brute-force restructuring process to convert this graph into a disjunction of conjunctions through the use of DeMorgan's Theorem produces a graph that is exponential in size in terms of the number of disjoining operators. This size complexity quickly becomes a burden because restructuring requires an exponential runtime complexity, and soon thereafter becomes a limitation because it may easily consume all available memory.

Figure 1(a) depicts an example DDG produced by the `"when others =>"` block of a case statement for signal *A*, such that the guards for the case statement's two explicit cases are: (*i*) `"when 1 =>"` and (*ii*) "`when 3 =>"`. Figure 1(b) shows the restructured DDG (with no optimization) using the method described earlier. We next estimate the size of the restructured DDG that represents the control requirements for the `"when others =>"` block of a case

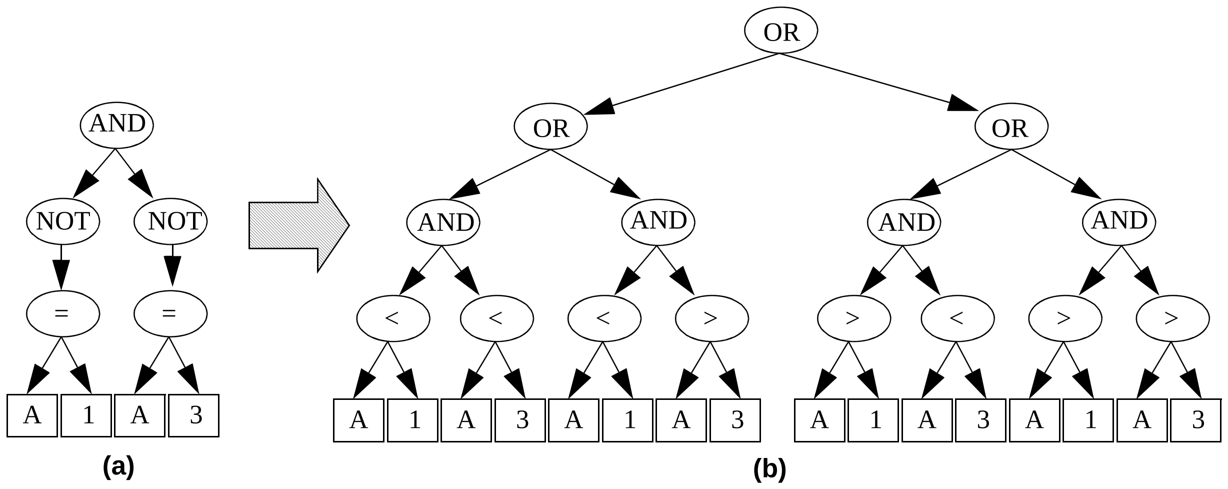

Figure 1 A sample DDG: (a) Before and (b) after restructure (with no optimization).

statement with *n* explicit cases.

On a case statement with *n* explicit cases, the `“when others =>”` block will have a guard that is a conjunction of *n* inequality operators. Each inequality operator is replaced as follows: $(A \neq B) \Rightarrow ((A < B) \text{ OR } (A > B))$. So a conjunction of *n* inequality operators will be a conjunction of disjunctions, and each disjunction will have 2 operators. An example of this conjunction of disjunctions is (($A < 1$) OR ($A > 1$)) AND (($A < 3$) OR ($A > 3$)) AND (($A < 4$) OR ($A > 4$)).

We then perform DeMorgan's Theorem to propagate the OR operators above the AND operators starting at the end of the tree. The operators in each disjunction will be duplicated many times as they are being placed into conjunctions (distributive property). Every conjunction in the final tree will have one relational operator from each original disjunction, and hence, each conjunction will have *n* operators. In order to determine how many operators there are in the final tree, we need to determine the number of final conjunctions, in other words, how many combinations we can have such that exactly one relational operator is taken from each original disjunction (either > or <). For each original disjunction, there are 2 choices and since there are *n* original disjunctions, then there are 2^n possible combinations. With 2^n conjunctions in the final tree and each conjunction has *n* operators, therefore there are $n \times 2^n$ operators. Each operator has 2 leaves, so there are $n \times 2^{n+1}$ leaves. Since a complete tree with *k* leaves has $(2k - 1)$ nodes and since the OR-AND part of the DDG is a complete tree, then the overall DDG has a total of $(3n + 2)x2^n - 1$ nodes.

Figure 7 shows the exponential runtime of the solver due to the brute-force restructuring process discussed in this section, under the line labeled *Reduce None*.

4. Experimental Setup

Our focus for this paper is to optimize the *restructure-and-solve* process for large case statements. We wish to analyze the performance of our solver when a constraint is placed on a signal that appears on every block of the case statement (including the `“when others =>”` block), such that only the `“when others =>”` block is able to satisfy the constraint. This will force our ATPG algorithm to analyze a large set of possible scenarios (prospect states) as it identifies the input requirements. We wish to study this test scenario because it will force the restructure-and-solve process to completely analyze the `“when others =>”` block’s control requirements, as it will not reduce to *false*. Figure 2 shows the VHDL description of the used test circuit, where multiple instances were made such that each instance had a distinct number of explicit cases in the case statement.

These test circuit instances were then converted into C++ library elements by using our simple parser. We created a simple ATPG test bench for each test scenario such that the same constraint was inserted into the ATPG unit for all scenarios. Figure 3 shows this simple test bench for a test circuit with 100 explicit cases in the case statement, and illustrates the simplicity of inserting constraints into the ATPG unit prior to extracting the ATPG implications. Multiple constraints can be inserted into the ATPG unit prior to extracting a solution; the multi-threaded solver does not initiate until `extract_globalStateSet()` is called.

```
type num_type is ( zero, one, two, ...);
entity MTV05 is
    port (      in_val:     in  num_type;
                clk:        in std_logic;
                out_val:    out integer);
end;
architecture MTV05_ARCH of MTV05 is
begin
    converter : process( clk, in_val )
    begin
        if clk'event and clk = '1' then
            case in_val is
            when one =>    out_val <= 1;
            when two =>    out_val <= 2;
                .
                .
                .
            when others => out_val <= 0;
            end case;
        end if;
    end process;
end MTV05_ARCH;
```

Figure 2 The test circuit used.

```
#include "mtv05_100.h"
#include <mATPG/pStateSet.h>
#include <iostream>

using namespace std;

int main()
{
    mtv05 cut;
    cut.insert_constraint("MTV05/out_val", "0");
    pStateSet *s = cut.extract_globalStateSet();

    if (s) { //if solution exists:
        cout<<"A solution was extracted!"<<endl;
        cout<<"A set with "<< s->list_size();
        cout<<" prospect states was generated.\n";
        s->print_pStateSet_prefix(cout);
    }
    else cout<<"No solution was extracted.\n";
}
```

Figure 3 ATPG testbench.

The graphs under Figure 7 demonstrate the gradual improvements in our solver's runtime complexity for the test bench set described above after each series of optimizations. The initial performance of our solver demonstrated by the graph labeled "Reduce None" forced us to re-think our solver strategy, and demonstrates the necessity for the optimizations presented in this paper. All tests have been performed on a Dual 2.5GHz G5 workstation under OS X Tiger using gcc 4.0. MVP has been implemented as a library using GNU's autotools (`autoconf`, `automake`, `libtool`) in 15K lines of C++ code. The test circuits similar to Figure 2 were converted from VHDL into C++ library elements, such that they implement stand-alone ATPG units that make use of MVP's libraries. Each test bench was run on top of the BSD *time* utility to determine the time used to execute each test bench only.

5. Optimizations

In this section, we describe the optimizations performed on DDGs in order to reduce the size of the ATPG search space.

5.1. *Unconditionally False* Sub-Trees

The performance limitation described in Section 3 was caused by the complexity of restructuring a DDG into a disjunction of conjunctions that does not contain disjoining operators. Analyzing the restructured graph of Figure 1 brought to our attention the presence of sub-trees that can be removed early in the restructuring process because they evaluate to *false*. We can say that these sub-trees are *unconditionally false*. We can identify trees that are unconditionally false by attempting to force a Boolean *true* value onto any Boolean operator or relational operator. A sub-tree will only be able to satisfy the *true* value if the range of values imposed onto all identifiers at that sub-tree intersects with the range of values imposed on corresponding identifiers at all other sub-trees of the same conjunction. These operators will return a value of SUCCES if successful, and will return a value of FAIL otherwise.

Applying DeMorgan's Theorem to propagate a Boolean OR operator above a Boolean AND operator begins with a DDG sub-tree rooted at a Boolean AND operator and results in an equivalent sub-tree that is rooted at a Boolean OR operator. In order to optimize the algorithm that restructures a DDG, we had the Boolean AND operator perform a `reduce()` operation on the sub-tree rooted at the Boolean OR operator that was generated by applying DeMorgan's Theorem. This `reduce()` operation recursively travels down to all relational operators and attempts to force a *true* value onto these operators. It replaces any of these relational operators with a Boolean *false* literal if a FAIL value is received in return. The `reduce()` algorithm reduces Boolean AND and Boolean OR nodes that are connected to a Boolean literal (an unconditional value) accordingly as it recursively returns back to the node it was called on.

This optimization effectively reduces the size complexity of the restructure process. Figure 4 depicts the graph of Figure 1 after it is restructured using this

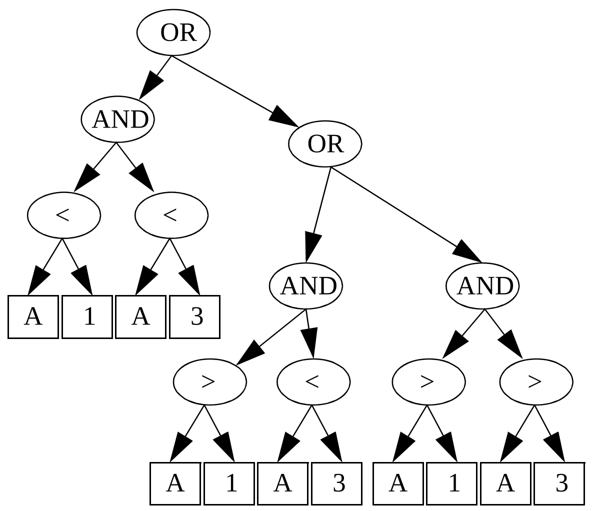

Figure 4 The sample DDG after reducing the unconditionally false sub-trees.

optimized algorithm. This optimization made the runtime complexity of the restructure process feasible, as it allows prospect states for most common constraints to be solved within several seconds.

However, a complex data dependency on a constraint will identify a series of signals whose value may depend on the circuit's architectural state. Such a data dependency will force the ATPG unit to analyze the same control requirements a series of times, once for each identified signal. If all of these identified signals are dependent on a large explicit FSM, the large case statement that implements the FSM will therefore be analyzed a series of times. This will multiply the computation time demonstrated by the second graph of Figure 7 (labeled *Reduce False*), and will render the runtime complexity of the restructure algorithm to be unacceptable.

5.2. *Unconditionally True* Sub-Trees

To further optimize our restructure algorithm, we similarly analyzed the restructured graph of Figure 4. This brought to our attention the presence of sub-trees that can be removed early in the restructuring process because they evaluate to *true*. These sub-trees occur when a comparison on an identifier does not reduce the range of values imposed on that identifier, therefore we can say that these sub-trees are *unconditionally true*. The reduced graph of Figure 4 is provided in Figure 5.

Enabling our `reduce()` operation to identify sub-trees that are unconditionally true required modifying a dyadic operator's method of solving its sub-tree. Now, a sub-tree returns EXPENDABLE if it is

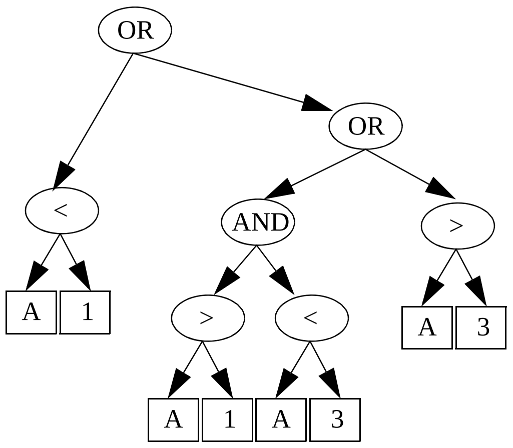

Figure 5 The sample DDG after reducing the unconditionally false (and true) sub-trees.

unconditionally true, returns FAIL if it is unconditionally false, and returns SUCCESS otherwise. An operator is EXPENDABLE if its child(ren) is(are) expendable, returns SUCCESS if no data contradiction is encountered, and returns FAIL otherwise. An identifier is EXPENDABLE if the range of values imposed on it encapsulates the range of values imposed onto the same identifier at a different sub-tree of the same conjunction, returns SUCCESS if the range of values imposed onto it intersect the range of values imposed onto the same identifier at a different sub-tree of the same conjunction, and returns FAIL otherwise. Similarly, a literal returns EXPENDABLE if the range of values imposed onto it matches its value exactly, and returns FAIL otherwise. Figure 6 shows how a relational operator is reduced based on whether it is unconditionally true or unconditionally false.

This optimization effectively reduces the size complexity and runtime complexity of the restructure process. The third graph of Figure 7 (labeled *Reduce True and False*) depicts the computation time of the optimized `reduce()` algorithm. With this new algorithm, solving the constraint of Figure 3 on the circuit of Figure 2 with 100 explicit cases took 0.11 seconds using a single thread of execution. Furthermore, its runtime complexity appears to be linear given that a case statement with 20 explicit cases required 0.02 seconds for a solution to be reached, and one with 50 explicit cases required 0.05 seconds.

6. Discussion

Performing the `reduce()` operation during the restructuring process results in two significant advantages. First, it allows a prospect state to be solved effi-

```
DDG *DDG_RelationalOperators::reduce()
{ //Attempt to force a value of true onto this relational operator:
  bind_result_type result = this->force_solution(new Boolean_ValueRange(true));
  switch (result) {
  case FAIL:              //If a contradiction was encountered (unconditionally false):
          disconnect();
          return (new DDG_BooleanLiteral(false))->connect();
          break;
  case EXPENDABLE:        //If this sub-tree did not produce useful results (unconditionally true):
          disconnect();
          return (new DDG_BooleanLiteral(true))->connect();
          break;
  case SUCCESS:           //This sub-tree is necessary:
          return this;
          break;
  }
}
```

Figure 6 Optimized `reduce()` algorithm for relational operators.

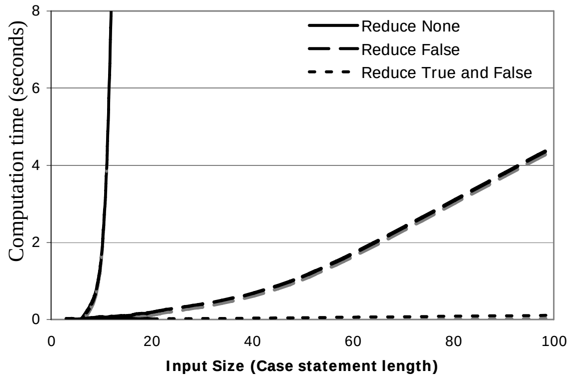

Figure 7 Simulation results.

ciently. Second, a prospect state's implications on the circuit's set of registers and primary inputs will have been identified once the restructure process terminates. As was mentioned earlier, the restructure process calls the `reduce()` algorithm on a sub-tree that was modified via DeMorgan's Theorem. It does this as a means to reduce this sub-tree before it performs DeMorgan's Theorem at higher levels in the graph (and thus, propagating the OR operators to the top of the tree). Calling `reduce()` will have the productive side-effect of forcing all conjunctions within that sub-tree to identify the range of values for all identifiers within it. This optimizes the solver because sub-trees are solved and reduced when they are small, and conjunctions that are joined into a greater conjunction via DeMorgan's Theorem can have any data contradictions immediately exposed based on each sub-graph's previously solved range in values.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0092867.

References

[1] J. Campos and H. Al-Asaad, "Mutation-based validation of high-level microprocessor implementations," *Proc. International High-Level Design Validation and Test Workshop*, 2004, pp. 81-86.

[2] A. Adir et al., "Genesys-Pro: Innovations in test program generation for functional processor verification," *IEEE Design and Test of Computers*, Vol. 21, pp. 84-93, March-April 2004.

[3] F. Corno et al., "Automatic test program generation: A case study," *IEEE Design and Test of Computers*, Vol. 21, pp. 102-109, March-April 2004.

[4] F. Corno, et al., "SymFony: A hybrid topological-symbolic ATPG exploiting RT-level information," *IEEE Transactions on Computer-Aided Design*, Vol. 18, pp. 191-202, February 1999.

[5] J. Campos and H. Al-Asaad, "Concurrent design error simulation for high-level microprocessor implementations," *Proc. AUTOTESTCON*, 2004, pp. 382-388.

[6] J. Campos and H. Al-Asaad, "MVP: A mutation-based validation paradigm," *Proc. International High-Level Design Validation and Test Workshop*, 2005, pp. 27-34.

[7] M. Abramovici, M Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, New York, 1990.

[8] G. Andrews, *Concurrent Programming: Principles and Practice*, Addison-Wesley Publishing Company, Menlo Park, California, 1991.