TAO: Regular Expression based High-Level Testability Analysis and Optimization

Srivaths Ravi, Ganesh Lakshminarayana and Niraj K. Jha

Department of Electrical Engineering
Princeton University, Princeton, NJ 08544

Abstract

In this paper, we present TAO, a novel methodology for high-level testability analysis and optimization of register-transfer level controller/data path circuits. Unlike existing high-level testing techniques that cater restrictively to certain classes of circuits or design styles, TAO exploits the algebra of regular expressions to provide a unified framework for handling a wide variety of circuits including application-specific integrated circuits, application-specific programmable processors, application-specific instruction processors, digital signal processors and microprocessors. We also augment TAO with a design-for-test framework that can provide a low-cost testability solution by examining the trade-offs in choosing from a diverse array of testability modifications like partial scan or test multiplexer insertion in different parts of the circuit. Test generation is symbolic and, hence, independent of bit-width. Experimental results on benchmark circuits show that TAO is very efficient, in addition to being comprehensive. The fault coverage obtained is above 99% in all cases. The average area and delay overheads for incorporating testability into the benchmarks are only 3.3% and 1.1%, respectively. The test application time is comparable to that associated with gate-level sequential test generators.

1 Introduction

With the increasing use of hierarchical design styles to tackle the size and complexity of modern VLSI circuits, there exists a need for evolving a framework for testability analysis and optimization that can cater to a range of architectural choices like application-specific integrated circuits (ASICs), application-specific programmable processors (ASPPs), application-specific instruction processors (ASIPs), digital signal processors (DSPs) and microprocessors, as well as design styles encompassing diverse features like multi-bit-width data paths and control-data chaining. While gate-level sequential test generators can handle these circuits, their high test generation time and low fault coverage renders them impractical. In this paper, we present TAO (stands for Testability Analysis and Optimization), a technique for high-level testability analysis and optimization that possesses the efficiency and speed of high-level testing and the wherewithal to handle a wide range of circuits and design styles.

Like many previous high-level test generation techniques, we use the *precomputed test set approach* for test generation. This approach significantly speeds up the test generation process by ensuring justification and propagation of the precomputed test set for a module-under-test (MUT) at the register-transfer level (RTL). The concept of precomputed test sets was proposed for acyclic RTL circuits in [1] and was combined with symbolic testability analysis during behavioral synthesis of ASICs in [2]. The methodology was extended to RTL descriptions of ASICs, ASPPs and ASIPs in [3, 4, 5]. These approaches do not easily extend across architectural choices or design styles. One approach for the test generation of instruction set processors[1](ISPs) was proposed in [6], wherein cost-based justification/propagation path selection was employed by testability analysis to generate test programs. However, cost-based testability analysis cannot capture all possible symbolic justification/propagation paths. This can trigger unwarranted design-for-test (DFT) hardware addition, leading to unnecessary area and delay overheads.

We present a comprehensive new strategy for testability analysis based on the *algebra of regular expressions*. Our testability analysis methodology tackles complete RTL controller/data path circuits and captures all justification/propagation paths by adopting the following approach. First, we construct equations for symbolic paths identified in the RTL data path description for testing a module in the data path. Then, we solve the equations to obtain *regular expressions* representing these paths. Instruction assembly, if needed, is shown to be a simple follow-up to these steps. In addition, we exploit the testability analysis to identify hard-to-test modules and then carry out DFT modifications, if necessary, for testability enhancement. We provide a general framework that can output a low-cost DFT solution by examining the trade-offs of carrying out a variety of DFT modifications, like partial scan and test multiplexer insertion, in different parts of the given circuit. Lastly, we use the test architecture model proposed in [4, 5] to test the controller separately.

The rest of the paper is organized as follows. Section 2 presents an example which illustrates that regular expressions provide a convenient and powerful means for testability analysis. Section 3 formalizes the notion of regular expression based testability analysis. Section 4 provides an integrated solution to the DFT problem within this setting. Section 5 gives an algorithmic description of the methodology outlined in Sections 3 and 4, while Section 6 presents experimental results. The conclusions follow in Section 7.

2 Perspective on Testability Analysis

In this section, we motivate the key ideas presented in this paper through an example. We first establish that the *completeness* of the test environment[2], *i.e.*, the ability of the test environment to completely capture the set of justification and propagation paths has a profound impact on symbolic test generation. We then illustrate that regular expressions provide a compact representation for test environments and form a natural solution to the problem of symbolic testability analysis.

Consider *SimpleRTL*, the RTL data path shown in Figure 1. Let us first conduct a naive symbolic testability analysis for the MUT, decrementer SUB. To test SUB, we can generate a test environment by the following set of actions.

- Load $R1$ from *PI-port* (*R1*←*PI-port*),
- Load $R3$ with A (*R3*←*R1+1*),
- Load $R2$ from *PI-port* while register $R3$ is in the hold mode (*R3*←*R3* and *R2*←*PI-port*),
- Load $R2$ with S while $R3$ is in the hold mode (*R3*←*R3* and *R2*←*S*).

[1] We will be using the term ISPs to refer to DSPs and microprocessors.

[2] The term *test environment* refers to a set of symbolic justification and propagation paths for a MUT, and was first introduced in [2].

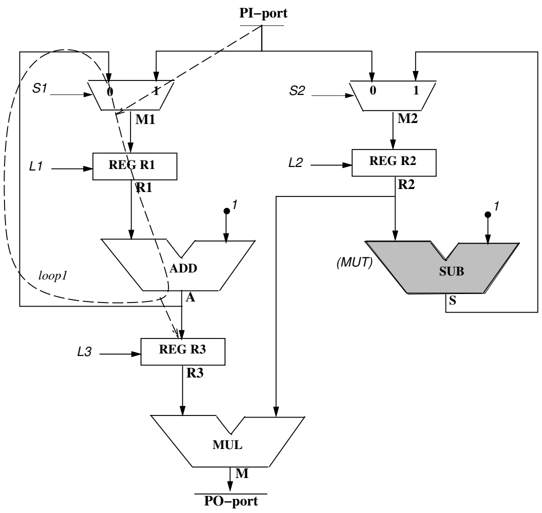


Figure 1: An RTL data path called *SimpleRTL*

This cycle-by-cycle behavior of the test environment can be concisely captured as follows:

$$E \;=\; (R1 \leftarrow PI\text{-}port,\ R3 \leftarrow R1+1,\ R3 \leftarrow R3 \text{ and } R2 \leftarrow PI\text{-}port,\ R3 \leftarrow R3 \text{ and } R2 \leftarrow S)$$

If complete controllability over the control signals exists, the test environment E is sufficient to directly help generate the values of the control signals needed in different cycles to test SUB. However, *complete controllability* over the control signals may not be possible in most practical cases, since the control flow in the datapath is usually dictated by a controller. Let us consider a controller for *SimpleRTL* that behaves like an ISP. That is, the controller takes an instruction as its input and outputs the appropriate set of control signals. Figure 2 shows a hypothetical instruction set for *SimpleRTL*, consisting of four instructions $I0$, $I1$, $I2$ and $I3$. Each instruction consists of one or more micro-instructions. For example, $I0$ consists of four micro-instructions while $I1$, $I2$ and $I3$ include one micro-instruction each. Each micro-instruction is a set of micro-operations (MOPs). In Figure 2, MOPs describe the status of registers $R1$, $R2$ and $R3$ in the datapath. The format adopted for MOPs, which are loads in registers, is *LoadRegister*(*Source*), and for MOPs, which are holds on registers, is *HoldRegister*. For example, $I1$ is characterized by the micro-instruction "*LoadR1*(*PI-port*) *LoadR2*(*PI-port*) *HoldR3*" which implies that $R1$ and $R2$ load from *PI-port* while $R3$ holds its previous value.

We will now show that the control path to $R3$ captured by E cannot be mapped to any instruction in the given instruction set. From the instruction set description, we can see that instruction $I0$ is the only instruction where $R3$ loads. Instruction $I0$ works as follows. In the first cycle of $I0$, $R1$ loads from *PI-port*, while $R3$ holds its previous value. In the second cycle, $R1$ is incremented

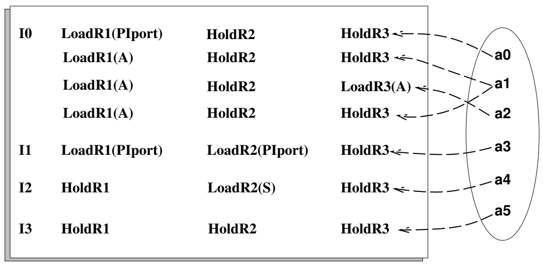


Figure 2: Instruction set of *SimpleRTL*

through loop1 (indicated by dotted lines in Figure 1) with $R3$ still in the hold mode. In the third cycle, both $R1$ and $R3$ load the output of the incrementer. The fourth cycle of $I0$ is identical to its second cycle. $R2$ remains on hold in all the four cycles. In other words, the control paths for $R1$ and $R3$, as provided by $I0$, consists of (*R1←PI-port* and *R3←R3, R1←R1+1* and *R3←R3, R1←R1+1* and *R3←R1+1, R1←R1+1* and *R3←R3*). In the case of SUB, the control path captured in E for controllability of $R3$ is (*R1←PI-port, R3←R1+1, R3←R3, R3←R3*). We can see that this path cannot be mapped to the control path provided by $I0$. In such a case, we may be tempted to resort to measures like testability insertion [3] or controller resynthesis [7, 8] to generate the requisite control path since testability analysis precludes other options. On the other hand, by merely expanding the testability analysis procedure to generate a test environment, that captures *all* possible symbolic justification and propagation paths, we can pre-empt unnecessary test overheads. Let us consider the following sequence of events in the datapath:

- Any sequence of MOPs is executed.
- $R1$ loads from *PI-port*.
- $R1$ increments and/or $R3$ holds multiple times.
- $R3$ loads A.
- Any sequence of MOPs is executed.

The above sequence of events is also a control path for $R3$ and includes the control path captured in E and that in $I0$. Such a control path can be written as (any MOP multiple times, *R1←PI-port, R1←R1+1* and/or *R3←R3* multiple times, *R3←R1+1,* any MOP multiple times), where "multiple times" implies zero or more occurrences. We can rewrite this path using a *regular expression* as (any MOP*.(*R1←PI-port*).((*R1←R1+1*)*.(*R3←R3*)*+((*R1←R1+1*) and (*R3←R3*))*)*.(*R3←R1+1*). any MOP*). Note that "and" in the above expression denotes conjunction of two events and is not a regular expression operator.

3 Testability Analysis

In this section, we formally evolve our approach towards testability analysis based on regular expressions. First, we examine the notions of controllability and observability of a module embedded in the data path using regular expressions. Then, we explain our approach to testability analysis. We develop and present this formalism with the help of our running example, *SimpleRTL*.

In Section 2, we saw that the instruction set and the test environment can be conveniently described using micro-instructions. We are, therefore, motivated to develop our formalism for testability analysis using micro-instructions. With the understanding that micro-instructions constitute the alphabet of regular expression based testability analysis, we introduce the following definitions.

Definition 1: A *control path*, CP_{M_i}, is a set, consisting of strings of micro-instructions that can control the output of a module, M_i, to any desired value. ∎

Definition 2: A *control path*, $CP_{M_i \leftarrow c}$, is a set, consisting of strings of micro-instructions that can control the output of a module, M_i, to value c. ∎

Definition 3: A *path equation* for CP_{M_i} relates CP_{M_i} to the control paths representing the inputs of M_i. ∎

The allowed set of operations for manipulating sets of strings consists of *concatenation*(.), *star operation* ($*$), *union* ($+$), *intersection* ($\cap$) and *complementation* ($'$). These operations are described in detail in [9]. Note that $+$ is also used to denote simple addition. The exact meaning of this symbol will be clear from the context. While a control path naturally describes the notion of controllability, we can define the notions of observability and test environment as follows.

Definition 4: *Observability* of the output of a module M_i is achieved if we can derive a set of strings of micro-instructions that can transfer the output of M_i to a primary output port. ∎

Definition 5: A *test environment* for a MUT is a set of strings of micro-instructions that describes all possible symbolic paths for testing a MUT, *i.e.*, control its inputs and observe its subsequent response. ∎

Our procedure for testing an RTL element (e.g., functional units, multiplexers, registers, etc.) in the data path involves writing and solving path equations to derive its test environment. We now describe this methodology in the context of our motivational example *SimpleRTL*. As before, the objective will be to test the decrementer, SUB. Since path equations, by definition, are formulated using control paths, we will first illustrate the process of deriving a control path in Example 1.

Example 1: Let us determine CP_{R1}, the control path for $R1$ in the data path shown in Figure 1. The objective here is to capture CP_{R1} in terms of the control path for *PI-port*, $CP_{PI\text{-}port}$, since primary inputs are fully controllable.

The first step in this process is to write a path equation for CP_{R1}. Since a register can either hold its previous value or load from its input, the path equation for CP_{R1} can be written as

$$CP_{R1} = CP_{R1}.HoldR1 + CP_{M1}.LoadR1 \tag{1}$$

The first term on the right hand side of Equation (1), namely $CP_{R1}.HoldR1$, tells us that an existing path for $R1$ followed (concatenated) by MOP $HoldR1$ is a control path for $R1$. The second term indicates that CP_{M1}, the control path representing the multiplexer output *M1*, followed by the load operation, is also a control path for $R1$. Hence, CP_{R1} can be expressed as the union of these two terms. The path equation for CP_{M1} is expressed as follows.

$$CP_{M1} = CP_A.Sel1' + CP_{PI\text{-}port}.Sel1 \tag{2}$$

In the above equation, $Sel1'$ and $Sel1$ are MOPs representing selection of ports 0 and 1 of the multiplexer, respectively. The output of an incrementer can be controlled directly by controlling its input. Therefore, the control path for A, CP_A, is simply given as

$$CP_A = CP_{R1} \tag{3}$$

Using Equations (2) and (3), we can rewrite Equation (1) as

$$\begin{aligned} CP_{R1} &= CP_{R1}.HoldR1 + CP_{R1}.(Sel1'.LoadR1) \\ &\quad + CP_{PI\text{-}port}.(Sel1.LoadR1) \\ Or,\ CP_{R1} &= CP_{R1}.HoldR1 + CP_{R1}.LoadR1(A) \\ &\quad + CP_{PI\text{-}port}.LoadR1(PI\text{-}port) \end{aligned} \tag{4}$$

Equation (4) captures the control path for $R1$ as a function of the control paths of *PI-port* and $R1$. The presence of CP_{R1} in the right hand side of the equation captures the loop structure *loop1* and the hold operation. At this stage, we may recall that the alphabet Σ for our analysis consists of the micro-instructions $a0$, $a1$, $a2$, $a3$, $a4$ and $a5$ shown in Figure 2. From the figure, we can see that $HoldR1$, $LoadR1(A)$ and $LoadR1(PI\text{-}port)$ are MOPS of micro-instructions $a4$ or $a5$, $a1$ or $a2$, and $a0$ or $a3$, respectively. Therefore, Equation (4) becomes

$$\begin{aligned} CP_{R1} &= CP_{R1}.(a4 + a5) + CP_{R1}.(a1 + a2) \\ &\quad + CP_{PI\text{-}port}.(a0 + a3) \\ Or,\ CP_{R1} &= CP_{R1}.(\Sigma - a0 - a3) \\ &\quad + CP_{PI\text{-}port}.(a0 + a3) \end{aligned} \tag{5}$$

Let us rewrite the above equation by annotating the micro-instructions with the MOPS performed by them. We introduce notation ax_y to denote the MOP on register *Ry* in the data path, for micro-instruction ax. For example, $a3_1$ specifies MOP $LoadR1(PI\text{-}port)$ for micro-instruction $a3$. The need for this notation will become clearer later. Thus, Equation (5) becomes

$$CP_{R1} = CP_{R1}.(\Sigma - a0 - a3)_1 + CP_{PI\text{-}port}.(a0 + a3)_1 \tag{6}$$

We can see that this iterative construct can be written in the familiar regular expression form [10], $R = Q + RP$, with $R = CP_{R1},\ Q = CP_{PI\text{-}port}.(a0 + a3)_1,\ P = (\Sigma - a0 - a3)_1$

Since $R = Q.P^*$ is a solution to $R = Q + RP$ by Arden's rule, we have

$$CP_{R1} = CP_{PI\text{-}port}.(a0 + a3)_1.(\Sigma - a0 - a3)_1^* \tag{7}$$

∎

In Example 1, we introduced the procedure for deriving the control path for an element in the datapath. In particular, we saw that the notion of controllability of the output of an element is easily captured by its control path. Likewise, we will show in Example 2 that the control path also captures the notion of observability of the output of an element.

Example 2: Consider the objective of deriving an expression for the observability of the output of the decrementer, SUB, in Figure 1. Since *PO-port* is directly observable, the task reduces to determining an expression for $CP_{PO\text{-}port}$ in terms of CP_S, the control path for the output of the decrementer. In other words, we must find an expression for $CP_{PO\text{-}port}$ while treating node S like a primary input port. From Figure 1, we can see that

$$CP_{PO\text{-}port} = CP_{MUL} \tag{8}$$

The control path for the multiplier, CP_{MUL}, consists of the set of micro-instructions that propagates values from one input of the multiplier when the other input is controlled to 1. This is captured by the following path equation.

$$CP_{MUL} = CP_{R3\leftarrow 1} \cap CP_{R2} + CP_{R3} \cap CP_{R2\leftarrow 1} \tag{9}$$

In a similar manner, we can write the path equation for CP_{R2} as follows:

$$\begin{aligned} CP_{R2} &= CP_{R2}.HoldR2 + CP_S.(Sel2.LoadR2) \\ &\quad + CP_{PI\text{-}port}.(Sel2'.LoadR2) \end{aligned} \tag{10}$$

Since our objective is to express $CP_{PO\text{-}port}$ as a function of CP_S, we need to consider only the following parts of Equations (9) and (10).

$$CP_{MUL} = CP_{R3\leftarrow 1} \cap CP_{R2} \tag{11}$$

$$CP_{R2} = CP_{R2}.HoldR2 + CP_S.(Sel2.LoadR2) \tag{12}$$

Using micro-instructions $(\Sigma - a3 - a4)$ and $a4$ to respectively account for MOPS $HoldR2$ and $(Sel2.loadR2)$, *i.e.*, $LoadR2(S)$, we can solve this equation using Arden's rule to obtain the following solution for CP_{R2}.

$$CP_{R2} = CP_S.a4_2.(\Sigma - a3 - a4)_2^* \tag{13}$$

From Equations (8), (11) and (13), we have

$$CP_{PO\text{-}port} = CP_{R3\leftarrow 1} \cap CP_S.a4_2.(\Sigma - a3 - a4)_2^* \tag{14}$$

Using the methodology outlined in Example 1, we can write the following expression for $CP_{R3\leftarrow 1}$

$$\begin{aligned} CP_{R3\leftarrow 1} &= CP_{PI\text{-}port}.(a0 + a3)_1.(\Sigma - a0 - a3)_1^*.a2_3 \\ &\quad .(\Sigma - a2)_3^* \end{aligned} \tag{15}$$

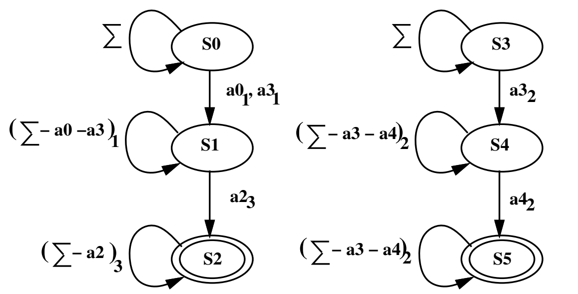
Equation (14) now becomes

$$\begin{aligned} CP_{PO\text{-}port} &= CP_{PI\text{-}port}.(a0 + a3)_1.(\Sigma - a0 - a3)_1^*.a2_3 \\ &\quad .(\Sigma - a2)_3^* \cap CP_S.a4_2.(\Sigma - a3 - a4)_2^* \end{aligned}$$

Since any sequence of operations can follow $CP_{PO\text{-}port}$ without affecting our objective ($CP_{PO\text{-}port}$ implies that observability has been achieved), the above expression concatenated with Σ^* is a complete representation for the observability of the output of the decrementer as a function of CP_S. In other words,

$$\begin{aligned} CP_{PO\text{-}port} &= ((CP_{PI\text{-}port}.(a0 + a3)_1.(\Sigma - a0 - a3)_1^*.a2_3 \\ &\quad .(\Sigma - a2)_3^*) \\ &\quad \cap (CP_S.a4_2.(\Sigma - a3 - a4)_2^*)).\Sigma^* \end{aligned} \tag{16}$$

∎



(a) (b)

Figure 3: (a) NFA_P that accepts P , and (b) NFA_Q that accepts Q

As mentioned earlier, our high-level testing model requires controllability of the inputs of a MUT and subsequent observability of its outputs. Having discussed the notions of controllability (Definition 1) and observability (Definition 4) in terms of the control path in Examples 1 and 2, we can now derive the test environment (Definition 5) for a MUT by the procedure illustrated in Example 3.

Example 3: Let us derive a micro-instruction level test environment for the decrementer, SUB, in Figure 1. In Example 2, we obtained an expression (Equation (16)) for the observability of the output of SUB. If we now obtain an expression for CP_{R2} in terms of $CP_{PI\text{-}port}$, the task of controlling the input of the MUT will be accomplished. Using the methodology outlined in Example 1, we can write the following equation for CP_{R2}.

$$CP_{R2} = CP_{PI\text{-}port}.a3_2.(\Sigma - a3 - a4)_2^* \tag{17}$$

Since SUB is a decrementer, we have

$$CP_S = CP_{R2} \tag{18}$$

From Equations (16), (17) and (18), it follows that

$$\begin{aligned} CP_{PO\text{-}port} \;=\; & (CP_{PI\text{-}port}.(a0 + a3)_1.(\Sigma - a0 - a3)_1^*.a2_3 \\ & .(\Sigma - a2)_3^* \cap \; CP_{PI\text{-}port}.a3_2.(\Sigma - a3 - a4)_2^* \\ & .a4_2.(\Sigma - a3 - a4)_2^*).\Sigma^* \end{aligned} \tag{19}$$

Equation (19) captures the micro-instruction level test environment for the MUT. To move towards instruction level test generation, we need to simplify the above equation and obtain a closed-form expression. We now describe this process of simplification. We first note that $CP_{PI\text{-}port}$ is basically Σ^*, since controllability of *PI-port* is unaffected by the choice of operations. Also, Equation (19) is of the form, $R = (P \cap Q).\Sigma^*$, with

$$\begin{aligned} P &= \Sigma^*.(a0 + a3)_1.(\Sigma - a0 - a3)_1^*.a2_3.(\Sigma - a2)_3^* \\ Q &= \Sigma^*.a3_2.(\Sigma - a3 - a4)_2^*.a4_2.(\Sigma - a3 - a4)_2^* \end{aligned} \tag{20}$$

In short, we have two regular expressions P and Q and want to obtain a regular expression for $P \cap Q$. The basic idea is to take the non-deterministic finite automata (NFA) for P and Q, NFA_P and NFA_Q, and combine them into a new NFA, $NFA_{P\cap Q}$. ∎

We will now use a modified version of the 5-tuple definition of an NFA [9] to formally compute the intersection machine $NFA_{P\cap Q}$.

Definition 6: An **NFA** is a 5-tuple $(Q, \Sigma, \delta, start, F)$, where, Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \rightarrow Q \times A$ is a one-to-many mapping called the transition relation with A referring to the *annotation* set , $start \in Q$ is the start state, and $F \subseteq Q$ is a set of accept states. ∎

The above definition modifies the standard notion of the transition relation in the sense that a transition from one state to another now carries information about the resources locked up by the MOP(s), as indicated by the *annotation* set. Also, note that the transition relation is a one-to-many mapping implying that the state can transition from one state to different next states on the same input (the standard notion of non-determinism). In addition, the transition relation is defined over Σ rather than $\Sigma \cup \epsilon$, where ϵ refers to the null string. If we now consider the regular expressions for P and Q in Equation (20), they can be represented by the two *NFAs* shown in Figures 3(a) and 3(b), respectively. Suppose that $NFA_P = (Q_p, \Sigma, \delta_p, start_p, F_p)$ and $NFA_Q = (Q_q, \Sigma, \delta_q, start_q, F_q)$ are the NFAs accepting P and Q, respectively. Then, the NFAs can be mathematically summarized as follows.

- $Q_p = \{S0, S1, S2\}$ and $Q_q = \{S3, S4, S5\}$
- By definition, $\Sigma = \{a0, a1, a2, a3, a4, a5\}$
- $start_p = \{S0\}$ and $start_q = \{S3\}$
- $F_p = \{S2\}$ and $F_q = \{S5\}$
- The transition relation can be defined as follows. Transition from state $S0$ to $S1$ occurs on an input symbol $a0$ with annotation set $\{1\}$. Hence, $\delta_p(S0, a0) = (S1, 1)$. Similarly, $\delta_p(S0, \Sigma) = (S0, NULL)$, $\delta_p(S1, a2) = (S1, 1)$, $\delta_p(S1, a2) = (S2, 3)$, and so on.

With this formalism, we are now ready to construct $NFA_{P\cap Q} = (Q, \Sigma, \delta, start, F)$ shown in Figure 4 as follows.

1. $Q = \{(X, Y) | X \in Q_p \text{ and } Y \in Q_q\}$.
   In other words, the Cartesian product of $\{S0, S1, S2\}$ and $\{S3, S4, S5\}$ defines this set.
2. $start = (start_p, start_q)$.
   Hence, $(S0, S3)$ is the start state of $NFA_{P\cap Q}$.
3. δ, the transition relation is defined as follows. For each $(X, Y) \in Q$ and each $a \in \Sigma$, we consider all possible specifications (due to one-to-many mapping) of $\delta_p(X, a) = (NS_X, ann_X)$ and $\delta_q(Y, a) = (NS_Y, ann_Y)$. Then, we define

   $$\delta((X, Y), a) = ((NS_X, NS_Y), \{ann_X \cup ann_Y\})$$

   whenever $\delta_p(X, a)$ and $\delta_q(Y, a)$ are both specified, and ann_X and ann_Y share **no** common resources. In all other cases, the transition relation is left undefined.

   For example, consider $(S0, S3) \in Q$. We are given that $\delta_p(S0, a0) = (S1, 1)$, $\delta_p(S0, a3) = (S1, 1)$, $\delta_p(S0, \Sigma) = (S0, NULL)$, $\delta_q(S3, a3) = (S4, 2)$, $\delta_q(S3, \Sigma) = (S3, NULL)$. If we now apply the above definition, we can write the transition relation δ, when the machine is in $(S0, S3)$, as follows.

   $$\begin{aligned} \delta((S0, S3), \Sigma) &= ((S0, S3), NULL) \\ \delta((S0, S3), a0) &= ((S1, S3), \{1\}) \\ \delta((S0, S3), a3) &= ((S1, S3), \{1\}) \\ \delta((S0, S3), a3) &= ((S0, S4), \{2\}) \end{aligned} \tag{21}$$

   From Figures 3(a) and 3(b), we can see that when NFA_P is in $S0$ and NFA_Q is in $S3$, there are four possible next state *destinations*, namely, from $(S0, S3)$ to $(S0, S3)$, $(S0, S4)$, $(S1, S3)$ and $(S1, S4)$. However, Equation (21) defines the transition relation for only three of them. This is because the transition from $(S0, S3)$ to $(S1, S4)$ due to $a3$ is invalidated by annotations 1 and 2 to the transitions from $S0$ to $S1$ and $S3$ to $S4$, respectively. In particular, MOPs $a3_1$ and $a3_2$, which represent $LoadR1(PI\text{-}port)$ and $LoadR2(PI\text{-}port)$, respectively, require simultaneous access to *PI-port*, thus giving rise to a conflict. Since our definition of the transition relation requires annotations to be conflict-free, the transition relation remains undefined for the transition from $(S0, S3)$ to $(S1, S4)$. In this way, a potential conflict due to reconvergence is successfully handled.
4. Machine $NFA_{P\cap Q}$ must accept its input if both NFA_P and NFA_Q accept this input. Hence, the accept states of $NFA_{P\cap Q}$ are those pairs wherein both NFA_P and NFA_Q are in an accept state. Thus, $F = \{(X, Y) | X \in F_p \text{ and } Y \in F_q\}$

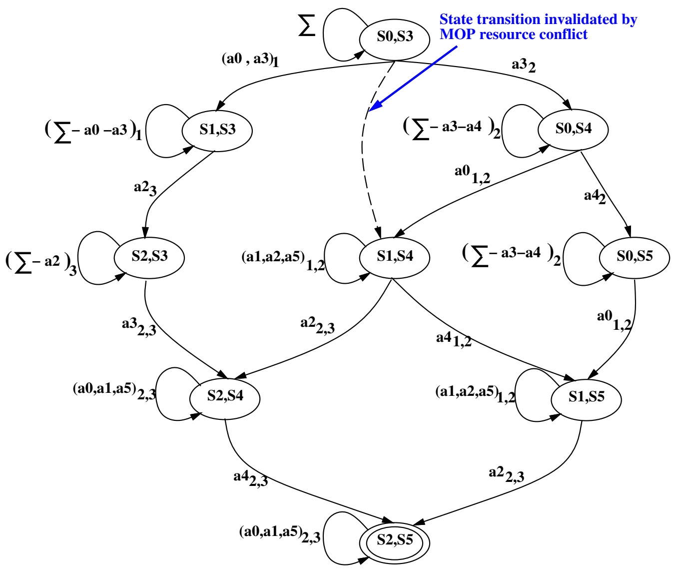


Figure 4: *NFA*$_{P \cap Q}$, an NFA that accepts $P \cap Q$

Having constructed *NFA*$_{P \cap Q}$, as shown in Figure 4, we can write its equivalent regular expression as follows.

$$\begin{aligned} CP_{PO\text{-}port} &= \Sigma^*\{(a0+a3)_1.(\Sigma - a0 - a3)_1^*.a2_3.(\Sigma - a2)_3^* \\ & \quad a3_{2,3}.x^*.a4_{2,3} + a3_2.y^*.(a0_{1,2}.z^*.(a2_{2,3}.x^* \\ & \quad .a4_{2,3} + a4_{1,2}.z^*.a2_{2,3}) + a4_2.y^*.a0_{1,2}.z^* \\ & \quad .a2_{2,3})\}.x^*.\Sigma^* \\ where \;\; x &= (a0+a1+a5)_{2,3},\; y = (\Sigma - a3 - a4)_2, \\ z &= (a1+a2+a5)_{1,2} \end{aligned}$$

In the above equation, we note that the regular expression derived for $CP_{PO\text{-}port}$ has Σ^* concatenated at its end. This is to account for the fact that any sequence of micro-instructions can follow once the observability objective has been achieved. In this way, a regular expression based representation of the micro-instruction level test environment for SUB has been obtained.

Since *SimpleRTL* has an ISP-like controller, the testability analysis carried out so far is not sufficient to generate the final test suite for the MUT. Our objective is, therefore, to map the test environment comprising of micro-instructions to a sequence of valid instructions that form the input to the controller. If we view the test environment as a sentence composed of strings and an instruction as a well-formed word, our task reduces to parsing a sentence into meaningful words. In other words, the procedure for generating the desired assembly of instructions is a simple lexical analysis, as is illustrated by the following example.

Example 4: Consider the task of generating a set of instructions that test the decrementer, SUB, in Figure 1, using the regular expression for the test environment, $CP_{PO\text{-}port}$, determined in the previous example. The instruction generation process can be viewed as a two-part exercise, which is outlined next.

1. The intersection machine, Z, generated from the NFAs for the test environment and the instruction set, gives the set of valid strings belonging to the instruction set that yield the test environment. Now, any set of instructions I that is generated from the instruction set Σ_I defined in Figure 2 can be written as follows:

$$\begin{aligned} \Sigma_I &= \{I0,\; I1,\; I2,\; I3\}, \quad I = \Sigma_I^* - \{\epsilon\} \\ Or, I &= (a0.a1.a2.a1 + a3 + a4 + a5)^* - \{\epsilon\} \end{aligned}$$

The procedure for computing the intersection machine $Z = I \cap CP_{PO\text{-}port}$ is as explained in the previous example and is not shown here.

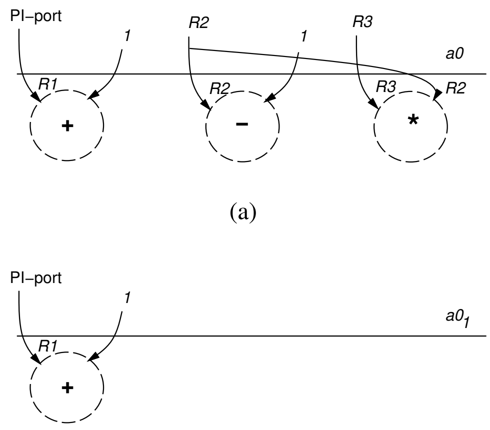
2. Note that Z, obtained in the previous step, will be composed of strings defined over the alphabet Σ of micro-instructions. However, transforming Z into a sequence defined over Σ_I is simply the task of parsing the sequence of micro-instructions into a set of syntactically correct programs, which can be easily performed by a lexical analyser like *lex* [11]. We then obtain

$$\begin{aligned} Z &= \Sigma_I^*.(I0.(I1+I2+I3)^*.I1.I3^*I2 + I1.(I3 + \\ & \quad I0)^*.I0.I3^*.I2 + I1.(I3+I0)^*.I2.(I3+I0)^* \\ & \quad .I0).\Sigma_I^* \end{aligned}$$

Expression Z represents many test programs for testing SUB. For example, one possible solution is $I0.I1.I2$ which we obtain from parsing $CP_{PO\text{-}port}$ into the string E given below.

$$E = a0_1.a1_1.a2_3.a1_3.a3_{2,3}.a4_{2,3} = I0.I1.I2 \tag{22}$$

Clearly, E is contained in $(a0.a1.a2.a1.a3.a4)$ which belongs to I, and hence $I0.I1.I2$ is an acceptable solution. ∎



(a)

(b)

Figure 5: CDFG for micro-instructions (a) $a0$ and (b) $a0_1$

So far, we have illustrated the procedure for deriving an instruction-level test environment for a MUT. This test environment fixes the inputs to the controller and decides the cycle-by-cycle behavior of the data path. In order to complete the test generation process, we must also fix the values necessary at the input ports of the data path in the different clock cycles. Since a control-data flow graph (CDFG) is a convenient model for describing the cycle-by-cycle behavior of a circuit, we can map the test environment to a CDFG. To do so, our methodology is to concatenate CDFG descriptions of the micro-instructions that form the test environment. We now describe the complete process with the help of the following example.

Example 5: In Example 4, we showed that $I0.I1.I2$ (Equation (22)) is a valid instruction-level test environment for decrementer SUB. Consider now the objective of fixing the values in the different clock cycles at *PI-port*, the input port of *SimpleRTL*. We can satisfy this objective by mapping the micro-instruction level test environment E (Equation (22)) to a CDFG.

For this purpose, we again observe that every micro-instruction description defines a CDFG. For example, micro-instruction $a0$ described in Figure 2 can be directly mapped to the CDFG shown in Figure 5(a). Now, consider $a0_1$, the first micro-instruction in E. Since the annotation describes the resources used by the data path, we use the annotation to do an AND *operation* with the CDFG to obtain the relevant subgraph. In other words, $a0_1$ is described by the CDFG shown in Figure 5(b).

We can repeat this process for all the micro-instructions in E and concatenate the CDFGs so obtained to construct the CDFG shown in Figure 6. From the CDFG, we can see that a value 1 at the left input of $*$ requires an *all-1's* vector at *PI-port* in the first

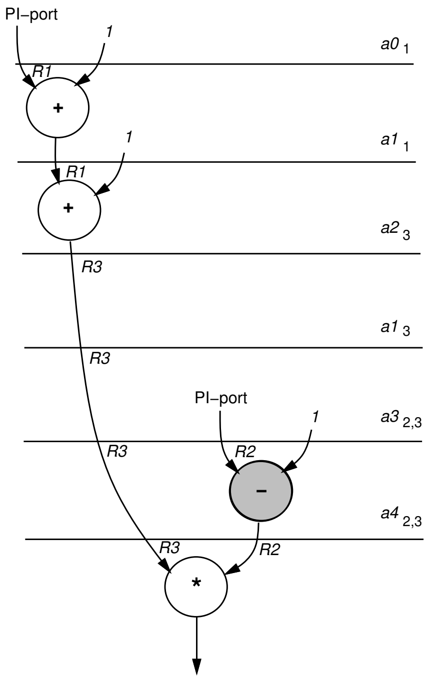


Figure 6: CDFG for test environment E

control step, while applying a test vector v to SUB requires v at *PI-port* in the fifth control step. In fact, these conditions on *PI-port* complete the test suite for SUB. In general, justifying system inputs on CDFGs is a simple exercise that can be systematically done by the method employed in [2]. ∎

4 DFT Using Symbolic Testability Analysis

Certain modules in a circuit can be hard-to-test. Designers often add DFT enhancements to the circuit to make such modules testable. Several DFT approaches have been proposed to aid the process of test generation for sequential circuits. Among them, full or partial scan based techniques [12, 13] are commonly used. However, the area and delay overheads incurred in scan-based designs are not always small. High test application time is also an area of concern in such designs. More recently, DFT techniques at the RTL have been proposed [3, 14, 15, 16]. These approaches selectively apply a DFT enhancement to a circuit ignoring possible benefits from a judicious application of a combination of these techniques. For example, the DFT approaches in [3] and [14] are based solely on test multiplexer addition and partial scan, respectively. We are, therefore, motivated to devise a framework that can systematically suggest DFT enhancements after examining the trade-offs of choosing/rejecting a DFT modification, be it partial scan or test multiplexer addition, somewhere in the given circuit. We next present an outline of our methodology for DFT enhancements.

- The first step is to identify hard-to-test modules in the given circuit. This is easily done by our testability analysis method presented in Section 3, which finds a module hard-to-test if and only if its test environment is found to be null.
- Next, we must identify areas in the circuit suitable for DFT addition. In other words, we must be able to identify candidates that satisfy the circuit constraints like area and delay.
- Now, we can construct a testable circuit from the given circuit by adding the DFT modifications generated in the previous step. We can again perform testability analysis to derive test environments for the hard-to-test modules. We must now determine a minimum cost subset of these modifications which results in a non-null test environment.

We will now discuss the methodology in complete detail with the help of a running example.

Example 6: Consider the RTL data path of *Checker*, an ASIP shown in Figure 7. Ignore the candidate DFT links shown in dotted lines for the time being. *Checker* has two input ports, $In1$ and $In2$, and one output port Out. In addition, a programmable instruction ROM (not shown in the figure) sources the control signals $S = \{S1, S2, S3, S4, L1, L2\}$. Thus, these control signals are completely controllable. In addition, control signals $S5$, $S6$ are, respectively, the output of comparators $COMP_<$ and $COMP_>$ in the data path.

Suppose the MUT is the multiplier MUL. Consider the process of deriving a test environment for this MUT. We can apply the methodology outlined in Section 3 to write a system of path equations as follows:

$$CP_{Out} = CP_{MUL}, \quad CP_{MUL} = CP_{R1} \cap CP_{R2} \qquad (23)$$

If we now write the path equations for CP_{R1} and CP_{R2}, we obtain

$$\begin{aligned} CP_{R1} &= CP_{M5}.\Sigma_{LoadR1(M5)} + CP_{K_0}.\Sigma_{LoadR1(K_0)} \\ &\quad + CP_{Out}.\Sigma_{LoadR1(Out)} + CP_{R1}.\Sigma_{HoldR1} \\ CP_{R2} &= CP_{M6}.\Sigma_{LoadR2(M6)} + CP_{K_1}.\Sigma_{LoadR2(K_1)} \\ &\quad + CP_{Out}.\Sigma_{LoadR2(Out)} + CP_{R2}.\Sigma_{HoldR2} \qquad (24) \end{aligned}$$

The path equations in Equation (24) are written keeping in mind that *Checker* is an ASIP and complete controllability over the control signals in S exists, and that micro-instructions still constitute our alphabet. In other words, Σ_{HoldR1} and $\Sigma_{LoadR1(M5)}$, for example, refer to the set of all micro-instructions that allow MOPs $HoldR1$ and $LoadR1(M5)$, respectively. Also, note that the number of micro-instructions for an ASIP is exponential in the number of control signals which, however, does not affect the symbolic testability analysis carried out. Proceeding with our analysis, we can write the following set of path equations.

$$\begin{aligned} CP_{M5} &= (CP_{K_2} + CP_{In1}).CP_{S5}, \\ CP_{M6} &= (CP_{K_3} + CP_{In2}).CP_{S6} \\ CP_{S5} &= CP_{R1} \cap CP_{R2}, CP_{S6} = CP_{R1} \cap CP_{R2} \qquad (25) \end{aligned}$$

If we now consider the path equations for CP_{R1} and CP_{R2}, control paths CP_{K_0} and CP_{K_1} are null (ϕ). This is because the corresponding nodes in the data path represent constant nodes and, hence, are not controllable to an arbitrary value. Similarly, we set CP_{K_2} and CP_{K_3} to ϕ. Also, the terms containing CP_{Out} in the path equations for CP_{R1} and CP_{R2} can be ignored, since any value at Out is directly observable and, hence, there arises no need for propagation of Out. The system of equations in Equations (24) and (25) now becomes

$$\begin{aligned} CP_{R1} &= CP_{M5}.\Sigma_{LoadR1(M5)} + CP_{R1}.\Sigma_{HoldR1} \\ CP_{R2} &= CP_{M6}.\Sigma_{LoadR2(M6)} + CP_{R2}.\Sigma_{HoldR2} \\ CP_{M5} &= CP_{In1}.CP_{S5}, \quad CP_{M6} = CP_{In2}.CP_{S6} \\ CP_{S5} &= CP_{R1} \cap CP_{R2}, CP_{S6} = CP_{R1} \cap CP_{R2} \qquad (26) \end{aligned}$$

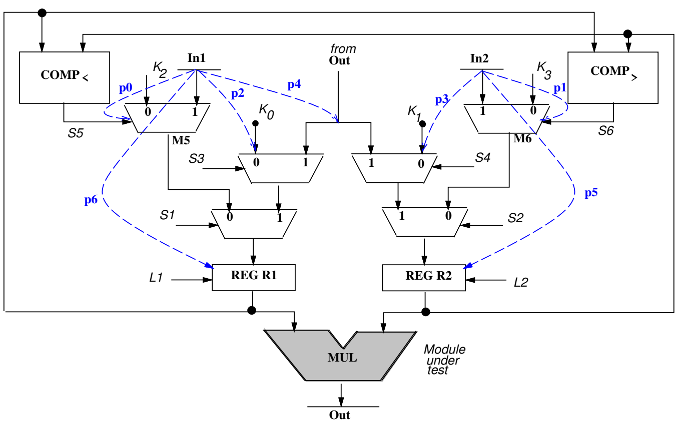


Figure 7: RTL description of an ASIP called *Checker*, with candidate DFT links in dotted lines

Solving the equations iteratively, starting with $CP_{R1} = CP_{R2} = \phi$, we get

$$CP_{R1} \;=\; \phi, \;\; CP_{R2} \;=\; \phi$$

This means that $CP_{Out} \;=\; \phi$. Thus, the test environment for MUL is ϕ due to the non-existence of control paths to $R1$ and $R2$. Hence, MUL is hard to test. ∎

Since timing is a critical specification, we examine delay constraints for a DFT modification by using an RTL true delay estimator [17]. We can then determine whether adding certain DFT hardware violates delay constraints in some portion of the circuit. Suppose we find that DFT modification p_x violates delay constraints, we then place a Boolean clause (p'_x) in a set of constraint relations called $Constraint\_set$. The condition $p_x = 0$ indicates absence of the DFT modification p_x, while $p_x = 1$ indicates the presence of p_x. If any clause database C includes $Constraint\_set$, satisfiability of C (and hence $Constraint\_set$) implies that DFT modification p_x is absent. Often, we discover DFT modifications which, when used simultaneously, violate delay constraints. For example, DFT hardware addition, say, of links p_x or p_y in a circuit may not violate the allowed slack in circuit delay, while adding **both** p_x and p_y may do so. This can occur in the simple case when p_x and p_y are serially connected, causing the delay numbers to add up. We can then express such a constraint as the Boolean condition $(p_x + p_y) \cap (p'_x + p'_y)$ and append the relation to $Constraint\_set$. Again, if $Constraint\_set$ must be satisfied, the above condition guarantees that both p_x and p_y cannot be present. We will describe $Constraint\_set$ in more detail later in the paper. We are now ready to analyze the impact of applying DFT modifications to a given RTL circuit.

Example 7: For the ASIP *Checker* considered in Example 6, let us suppose that we are given the option of adding DFT modifications so as to provide one or more of the links in a set $P = \{p0, p1, p2, p3, p4, p5, p6\}$, as shown in Figure 7. Assume that P was generated by the pre-processing steps as explained above, and that $p0$, ..., $p4$ refer to links added by test multiplexer addition along the corresponding dotted lines in Figure 7. In addition, links $p0$ and $p1$ require the presence of one 1-bit latch apiece. Link $p6$ puts register $R1$ on the scan path from $In1$, while link $p5$ puts register $R2$ on the scan path from $In2$. Our initial objective is to conduct a testability analysis of *Checker*, assuming that **all** the links in P are provided.

Equation (23) remains unchanged, while the set of path equations in Equation (26) get modified as follows:

$$\begin{aligned}
CP_{R1} &= CP_{M5}.\Sigma_{LoadR1(M5)} + CP_{In1}.\Sigma_{LoadR1(In1)\ via\ p2} \\
&\quad + CP_{In1}.\Sigma_{LoadR1(In1)\ via\ p4} + CP_{R1}.\Sigma_{HoldR1} \\
&\quad + CP_{In1}.\Sigma^{|ScanLoadR1(In1)\ via\ p6|}_{ScanLoadR1(In1)\ via\ p6} \\
CP_{R2} &= CP_{M6}.\Sigma_{LoadR2(M6)} + CP_{In2}.\Sigma_{LoadR2(In2)\ via\ p3} \\
&\quad + CP_{In2}.\Sigma_{LoadR2(In1)\ via\ p4} + CP_{R2}.\Sigma_{HoldR2} \\
&\quad + CP_{In2}.\Sigma^{|ScanLoadR2(In2)\ via\ p5|}_{ScanLoadR2(In2)\ via\ p5} \\
CP_{M5} &= CP_{In1}.(CP_{S5} + \Sigma_{In1bit\ via\ p0}) \\
CP_{M6} &= CP_{In2}.(CP_{S6} + \Sigma_{In2bit\ via\ p1}) \\
CP_{S5} &= CP_{R1} \cap CP_{R2}, \;\; CP_{S6} = CP_{R1} \cap CP_{R2} \qquad (27)
\end{aligned}$$

The modifications in Equation (27) merely reinforce the changes in the MOPs that arise due to the presence of a DFT enhancement. For example, $\Sigma_{LoadR1(In1)\ via\ p2}$ indicates that control signals must be asserted so as to allow register $R1$ to load from primary input port $In1$ using the DFT link $p2$. Similarly, $\Sigma^{|ScanLoadR1(In1)\ via\ p6|}_{ScanLoadR1(In1)\ via\ p6}$ accounts for the scan operation on $R1$, which consumes a fixed number of cycles given by $|ScanLoadR1(In1)\ via\ p6|$. This system of equations can now be solved iteratively by setting $CP_{R1} \;=\; \phi$ and $CP_{R2} \;=\; \phi$. We then obtain using Arden's rule

$$\begin{aligned}
CP_{R1} &= (CP_{M5}.\Sigma_{LoadR1(M5)} + CP_{In1}.\Sigma_{LoadR1(In1)\ via\ p2} \\
&\quad + CP_{In1}.\Sigma_{LoadR1(In1)\ via\ p4} \\
&\quad + CP_{In1}.\Sigma^{|ScanLoadR1(In1)\ via\ p6|}_{ScanLoadR1(In1)\ via\ p6}).\Sigma^*_{HoldR1} \\
CP_{R2} &= (CP_{M6}.\Sigma_{LoadR2(M6)} + CP_{In2}.\Sigma_{LoadR2(In2)\ via\ p3} \\
&\quad + CP_{In2}.\Sigma_{LoadR2(In1)\ via\ p4} \\
&\quad + CP_{In2}.\Sigma^{|ScanLoadR2(In2)\ via\ p5|}_{ScanLoadR2(In2)\ via\ p5}).\Sigma^*_{HoldR2} \\
CP_{M5} &= CP_{In1}.\Sigma_{In1bit\ via\ p0} \\
CP_{M6} &= CP_{In2}.\Sigma_{In2bit\ via\ p1} \\
CP_{S5} &= \phi, \;\; CP_{S6} = \phi \qquad (28)
\end{aligned}$$

Substituting for CP_{M5} and CP_{M6}, we can rewrite the equations obtained in Equation (28) and Equation (23) as follows.

$$\begin{aligned}
CP_{R1} &= (CP_{In1}.\Sigma_{In1bit\ via\ p0}.\Sigma_{LoadR1(M5)} \\
&\quad + CP_{In1}.\Sigma_{LoadR1(In1)\ via\ p2} \\
&\quad + CP_{In1}.\Sigma_{LoadR1(In1)\ via\ p4} \\
&\quad + CP_{In1}.\Sigma^{|ScanLoadR1(In1)\ via\ p6|}_{ScanLoadR1(In1)\ via\ p6}).\Sigma^*_{HoldR1} \\
CP_{R2} &= (CP_{In2}.\Sigma_{In2bit\ via\ p1}.\Sigma_{LoadR2(M6)} \\
&\quad + CP_{In2}.\Sigma_{LoadR2(In2)\ via\ p3} \\
&\quad + CP_{In2}.\Sigma_{LoadR2(In1)\ via\ p4} \\
&\quad + CP_{In2}.\Sigma^{|ScanLoadR2(In2)\ via\ p5|}_{ScanLoadR2(In2)\ via\ p5}).\Sigma^*_{HoldR2} \\
CP_{Out} &= CP_{R1} \cap CP_{R2} \qquad (29)
\end{aligned}$$

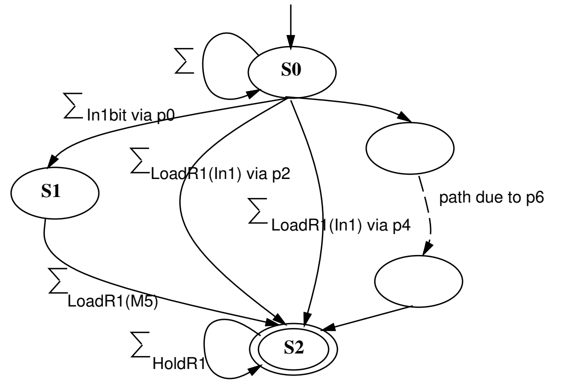


Figure 8: NFA that accepts CP_{R1}

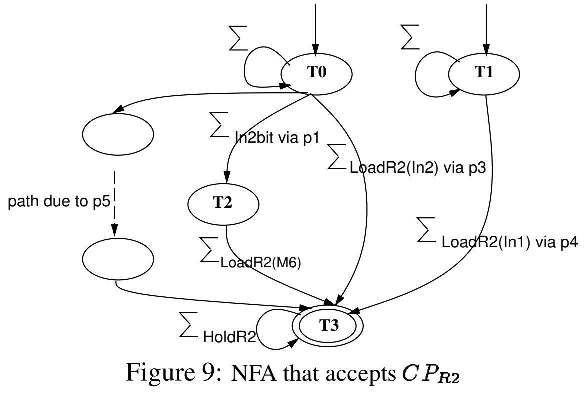


Figure 9: NFA that accepts CP_{R2}

From the above equations, we can derive the NFAs for CP_{R1} and CP_{R2}, as shown in Figures 8 and 9. Then, we can construct the intersection machine to yield the NFA for CP_{Out}, as shown

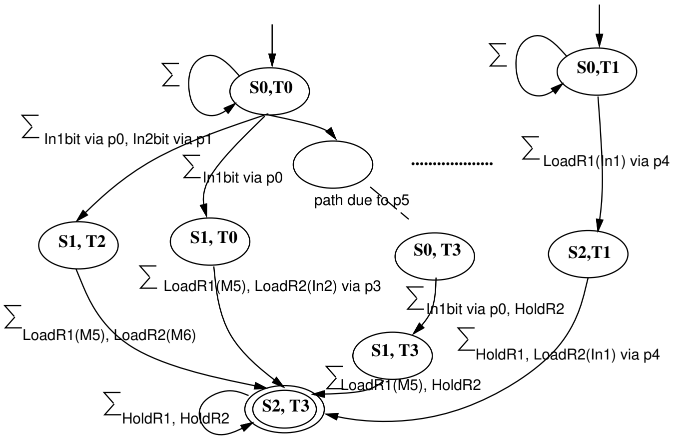


Figure 10: Portion of the NFA that accepts CP_{Out}

in Figure 10. This NFA has many accepting paths, which constitute a non-null test environment for MUL. Hence, we conclude the iteration at this juncture. ∎

The analysis in Example 7 defines the test environment for MUL in a DFT enhanced version of *Checker*. However, our analysis is not yet complete since we are interested in determining a test environment that employs a minimum cost subset of P. To achieve this objective, we exploit the isomorphic nature of algebras of language functions and Boolean functions [18]. Specifically, we convert a regular expression into an equivalent Boolean function such that a minimum cost satifying assignment for the Boolean function determines the desired minimum cost subset of P. The reduction uses the NFA for a test environment resulting from a post-DFT testability analysis, as in Example 7, to write down a Boolean function, as explained in the following example.

Example 8: Resuming our DFT analysis of *Checker*, let us derive the Boolean function whose satisfying assignments reflect the DFT modifications necessary for enhanced testability. We base this conversion on the simple observation that a non-null test environment for a hard-to-test module is made possible only by the presence of DFT modifications. As the first step, we generate *Boolean_function* from the NFA for CP_{Out} shown in Figure 10. From the figure, we can see that one path from a start state to the accept state is :

$$One\_path \;=\; (S0, T0) \rightarrow (S1, T2) \rightarrow (S2, T3) \quad (30)$$

We consider the labels on the transitions that comprise *One_path* and extract a representative product term. This product term is defined as Πp_i, wherein variable p_i appears in the product term if and only if the label on some transition in *One_path* includes a reference to the DFT link p_i. Applying this correspondence, we get

$$Product\_term_{One\_path} \;=\; p0.p1 \quad (31)$$

In a similar manner, we can generate the different product terms to yield the following expression for *Boolean_function*.

$$Boolean\_function \;=\; (p0 + p2 + p4 + p6) .(p1 + p3 + p4 + p5) \quad (32)$$

Recall that the pre-processing step of using the true delay estimator to analyze the effect of the DFT enhancements in P on the timing behavior of *Checker* yields *Constraint_set*. We must, therefore, modify *Boolean_function* (by ANDing the Boolean conditions defined in *Constraint_set*) to reflect the possible constraints. Assume that *Constraint_set* consists of the solitary Boolean clause $p4'$. Note that we could have eliminated $p4$ from consideration in an earlier pre-processing step. However, this is not possible in most cases since a combination of DFT modifications (rather than a solitary modification like $p4$ here) jeopardize timing of the circuit (see discussion following Example 6). Then, *Boolean_function* becomes

$$Boolean\_function \;=\; (p0 + p2 + p4 + p6) .(p1 + p3 + p4 + p5).(p4') \quad (33)$$

∎

Having derived *Boolean_function* in Example 8, we are now ready to determine the solution to our DFT problem.

Example 9: In order to generate a minimum-cost DFT solution, we must determine a minimum-cost cover of *Boolean_function* obtained in Example 8. To determine such a cover, we define a cost function, namely, $Cost\_func : P \rightarrow \Re$. In other words, we associate a cost with each DFT enhancement. In this example, we do so by using the following simple rule-of-thumb. *If p_x represents a link added by DFT modification to an* N-*bit interconnect in a hypothetical circuit, we define* $Cost\_func(p_x) = N$. Applying this yardstick to P and assuming an 8-bit data path, we obtain

$$Cost\_func(p0) \;=\; Cost\_func(p1) \;=\; 1$$

$$Cost\_func(p2) \;=\; Cost\_func(p3) \;=\; Cost\_func(p4) \;=\; 8$$

$$Cost\_func(p5) \;=\; Cost\_func(p6) \;=\; 8$$

Finding the minimum cover for a Boolean function with a well-defined cost function is a well-known problem in computer-aided design (CAD) and efficient algorithms described in [19] can be used. Solving thus, we get $(p0 = 1, \; p1 = 1)$ as the min-cost satisfying asssignment for *Boolean_function*. We can clearly see that $\{p0, p1\}$ is our desired solution. This is because DFT links $p0$ and $p1$ are added by 1-bit test multiplexer-cum-latches, and do not violate the delay constraints defined by *Constraint_set*. ∎

5 The Algorithm

In this section, we formalize the methods described in Sections 3 and 4. Figure 11 shows the top-level pseudocode that we use to generate the system-level test set of an RTL controller/data path circuit. The procedure takes as its input an RTL circuit R with controller specification $R.C$ and data path connectivity $R.D$, design library L, and an integer *limit* which specifies the bound (time steps or iteration steps) on the subroutine for finding the solutions iteratively. Other inputs include a set of paths P for DFT enhancement and a constraint set of relations CC, which reflects the subset of paths, or combinations thereof, unavailable for DFT enhancement. Register scan and test multiplexer addition are the possible means considered for DFT enhancement. In addition, test multiplexer addition is restricted to inputs of registers and control inputs of RTL elements to define P concisely. A true delay estimator is used to define CC. It is assumed that we have pre-defined the alphabet for testability analysis and also that L includes path equations for the different elements in the circuit.

The first step in the test generation process is to conduct testability analysis of the different elements in the circuit. To do this, we invoke the procedure *Test_Anal* shown in Figure 12. *Test_Anal* works as follows. For each element M in the data path, we derive its test environment $TestEqn_M$ using the procedure *Generate_Path_Eqn*. *Generate_Path_Eqn* uses the data path connectivity information in $R.D$, the element path equations in L, and the

```
Procedure Test_Gen(RTL R, Design_lib L, limit,
          DFT_paths P, Constraint_clause CC){
   /* Conduct testability analysis on RTL R with
   /* data path R.D and controller spec R.C */
   Test_Anal(R.D,R.C,L,limit,P,CC);
   /* Assemble system-level test environment */
   SysTestEnviron = Transform(SolnEqn,R.C);
   /* Assemble test suite */
   TestSuite = Val_Anal(SysTestEnviron,
        SolnEqn,L);}
```

Figure 11: Procedure for testing the RTL circuit

methodology outlined in Section 3 for writing the necessary path equations to derive $TestEqn_M$. We then attempt to determine $SolnEqn_M$, a solution to $TestEqn_M$, iteratively. $SolnEqn_M$ is null if no solution exists or is not determinable within a preset number of steps, *limit*. We then mark off such an M as hard-to-test and store our observation in $HTTlist$. We now proceed to the next element and repeat the process. Once we have exhausted the elements, we examine the elements in $HTTlist$ using procedure *DFT_Anal*.

```
Procedure Test_Anal(RTL R, Design_lib L, limit,
            DFT_paths P, Constraint_clause CC){
  HTTlist = NULL;
  for (each element M in R.D) {
    TestEqn_M = Generate_Path_Eqn(M,L,R.D);
    SolnEqn_M = Iterative_Solve(TestEqn_M,limit);
    if (SolnEqn_M == NULL)
      Append(HTTlist,M);
  }
  if (HTTlist ≠ NULL)
    DFT_Anal(HTTlist,R,L,limit,P,CC);}
```

Figure 12: Procedure for testability analysis

```
Procedure DFT_Anal(HTTlist,R,L,limit,P,CC){
  Formula = CC;
  R.D_Enhanced = R.D ∪ P;
  for (each element M in HTTlist){
    TestEqn_M = Generate_PE(M,L,R.D_Enhanced);
    SolnEqn_M = Iterative_Solve(TestEqn_M,limit);
    if (SolnEqn_M == NULL)
      Append(GiveUplist,M);
    else
      Formula = Formula ∪ Clause(SolnEqn_M);
  }
  Subset_P = MinCost_Cov(Formula);
  Additions = (HTTlist − GiveUplist);
  Update_SolnEqn(SolnEqn,Additions,Subset_P);}
```

Figure 13: Procedure for DFT addition

The purpose of *DFT_Anal* is to determine $SolnEqn_M$ for elements in $HTTlist$ by using a subset of P, the set of paths for DFT enhancement. To do this, *DFT_Anal* conducts the testability analysis for every element M in $HTTlist$, using the DFT enhanced data path $R.D\_Enhanced$. $R.D\_Enhanced$ reflects the state of the data path assuming that all the paths in P are available. If the output of testability analysis, $SolnEqn_M$, is non-null, a Boolean formula is obtained using the procedure described in Section 4. A composite Boolean formula $Formula$ is then constructed from the Boolean formulae for the different elements in $HTTlist$ and the constraint clause CC. $Formula$ is subsequently fed to *MinCost_Cov*, a minimum cost cover finder, to obtain $Subset_P$, the paths needed for DFT enhancement. $SolnEqn$ is then modified to exclude the elements of $(P - Subset_P)$. Since some elements may still be untestable, we return a list containing such elements, namely, $GiveUplist$.

We now return to *Test_Gen* to complete the test generation process. With testability analysis successfully done, we can now determine the system test environment $SysTestEnviron$ as dictated by the controller specification $R.C$. Example 4 in Section 3 is symbolic of this step. It now remains to assemble the final test suite. We do this by subjecting $SysTestEnviron$ to a value analysis procedure [2] (see Example 5) that uses the precomputed test sets for the elements stored in L.

6 Experimental Results

In this section, we present experimental results obtained by applying TAO to some example circuits. We adopt the test architecture model presented in [4, 5] to carry out our testing scheme and to test the controller separately.

Table 1: Circuit Index

Ref.	Circuit	Type
1	Paulin	ASIC
2	GCD	ASIC
3	ASPP4	ASPP
4	Checker	ASIP
5	SimpleCPU	ASIP
6	Am2910	ISP
7	SimpleRTL	ISP

Table 2: Circuit and DFT hardware statistics

Ref.	BW	# literals	# FFs	# TMs
1	16	6826	116	1
2	16	1333	50	0
3	16	11030	117	0
4	NU	3760	32	2
5	16	17173	282	0
6	NU	4336	308	1
7	16	3723	50	0

Table 3: DFT hardware placement overheads

Ref.	*Area*			*Delay*		
	Orig. ($x10^3$)	*Mod.* ($x10^3$)	*Ovhd* (%)	*Orig.* (ns)	*Mod.* (ns)	*Ovhd* (%)
1	88.1	89.7	1.8	155.6	158.8	2.1
2	16.6	17.7	6.6	58.6	60.4	3.1
3	143.3	145.1	1.3	97.6	97.7	0.1
4	37.8	38.8	2.6	112.8	113.1	0.3
5	187.4	196.1	4.6	138.2	139.0	0.6
6	66.6	70.4	5.7	88.0	89.4	1.6
7	39.4	40.8	3.6	107.0	108.0	0.9

Table 1 gives the reference for the different circuits tested using our methodology. For example, reference number 1 is used to refer to the circuit *Paulin* and so on. The circuits tested include ASICs, ASPPs, ASIPs and ISPs. The ASICs and ASPPs were generated by the high-level synthesis system Genesis(area) [2], that targets area optimization, *i.e.*, the testability flag in Genesis was turned off. The ASIC *Paulin* is a differential equation solver and is data-dominated, while *GCD* computes the greatest common divisor of two numbers and is control-flow intensive. Both the circuits are described in detail in [4]. *ASPP4* is an ASPP taken from [5] that can emulate the behavior of either *Paulin* or *Tseng*. The ASIP *SimpleCPU* has been taken from [5, 20], while the ISP *Am2910* is described in [21]. The other examples, *SimpleRTL* and *Checker*, are as described in the previous sections.

The data paths were optimized at the logic level using SIS [22]. They were then technology-mapped with the *stdcell2_2.genlib* library. Instruction ROMs, wherever required, were generated using the ALLIANCE synthesis system [23]. The designs were then scaled down to the data path technology to get the complete area and delay figures of the circuits. Table 2 shows the size and DFT hardware statistics of the circuits obtained. The data path bit-width of the different circuits is given in Column 2. The dominant bit-width (ignoring control-data chaining) of all the circuits is 16, while *NU* stands for non-uniform bitwidth. Columns 3 and 4 show the literal-counts and the number of flip-flops present in the circuit, respectively. We found that our cost function always resulted in a test multiplexer [3] based DFT solution for enhancing circuit testability, if our testability analysis demanded a DFT enhancement. The number of test multiplexers used in the data path of the circuit is given in Column 5. This number just consists of the number of extra multiplexers in the data path and does not include the multiplexers used for testing the controller which are added by default to all circuits [4, 5], and which have been included while reporting the overheads in Table 3.

Table 3 shows the area and delay overheads for our DFT scheme. In Column 2, the original technology-mapped area of the circuits is given. This is a relative figure and, hence, has no units. Column 3 shows the area after DFT hardware addition. The area overheads are shown in Column 4. Columns 5, 6, and 7 show the corresponding delay numbers. The delay number represents the clock cycle time of the circuits in *ns*. The average area overhead is 3.3% and the average delay overhead is 1.1% (the average is calculated based on the total area/delay of all the examples for the original and modified cases). The delay overheads are negligible because the test multiplexers, if any, are all placed on non-critical paths (our DFT analysis precludes critical paths or combinations of non-critical paths that have the potential of becoming critical).

Table 4: Testability results

Ref.	*HITEC*				*TAO*	
	Orig. Ckt.		*Mod. Ckt.*		*Mod. Ckt.*	
	FCov (%)	*TApp* (cyc)	*FCov* (%)	*TApp* (cyc)	*FCov* (%)	*TApp* (cyc)
1	91.36	108	98.57	223	99.49	507
2	86.12	203	98.65	267	99.08	278
3	92.50	171	99.40	338	99.84	548
4	76.45	92	98.85	321	99.87	451
5	77.80	754	98.91	1140	99.60	1337
6	90.61	1342	96.58	1550	99.25	1782
7	98.20	359	99.37	367	99.91	422

In Table 4, testability results for the circuits are presented. We compare the performance of TAO against HITEC [24], an efficient gate-level sequential test generator. Column 2 shows the fault coverage obtained by running HITEC on the original data paths using the default controllers. For the ASIPs we placed an FFT (fast Fourier transform) algorithm as the original program into the instruction ROM. Column 3 shows the test application time in clock cycles required by HITEC to obtain the given fault coverage. In Columns 4 and 5, the corresponding results are given when HITEC is run on the circuits modified by our DFT hardware. The corresponding results obtained by running our algorithm on the modified circuits are shown in Columns 6 and 7, respectively. For our high-level test generator, the fault coverage numbers were obtained by fault simulating system-level test sets on the modified circuit using PROOFS [25]. Whereas TAO was used to generate the data path test set, HITEC was used to derive the controller test set from its gate-level description. Then, the two test sets were concatenated to derive our system-level test set. It can be seen that very high fault coverage results from applying TAO on a variety of circuits. The test application times are also comparable with HITEC. Note that test application times can be strictly compared when the fault coverages are similar since it is well-known that for detecting the last few faults, a gate-level sequential test generator may require very large test sequences.

7 Conclusions

In this paper, we presented a novel methodology for high-level testability analysis, DFT and test generation of RTL controller/data path circuits. Our algorithm, TAO, exploits the algebra of regular expressions to provide a unified setting for the testability analysis of a wide variety of circuits like ASICs, ASPPs, ASIPs, and ISPs. The architectural domain spanned by TAO is shown to include the domain handled by some of the earlier work in high-level testing [1]-[5]. In particular, these previous works could not handle ISPs. Our environment for testability analysis easily locates hard-to-test areas in the circuit and provides a framework for efficient DFT enhancements, whenever necessary. Experiments performed on several benchmark circuits indicate a high fault coverage with low DFT overheads.

References

[1] B. T. Murray and J. P. Hayes, "Hierarchical test generation using precomputed tests for modules," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 594–603, June 1990.

[2] S. Bhatia and N. K. Jha, "Genesis: A behavioral synthesis system for hierarchical testability," in *Proc. European Design and Test Conf.*, pp. 272–276, Feb. 1993.

[3] I. Ghosh, A. Raghunathan, and N. K. Jha, "Design for hierarchical testability of rtl circuits obtained by behavioral synthesis," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 1001–1014, Sept. 1997.

[4] I. Ghosh, A. Raghunathan, and N. K. Jha, "A design for testability technique of RTL circuits using control/data flow extraction," in *Proc. Int. Conf. Computer-Aided Design*, pp. 329–336, Nov. 1996.

[5] I. Ghosh, A. Raghunathan, and N. K. Jha, "Hierarchical test generation and design for testability of ASPPs and ASIPs," in *Proc. Design Automation Conf.*, pp. 534–539, June 1997.

[6] J. Lee and J. H. Patel, "Architectural level test generation for microprocessors," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1288–1300, Oct. 1994.

[7] S. Ravi, I. Ghosh, R. K. Roy, and S. Dey, "Controller resynthesis for testability enhancement of RTL controller/data path circuits," in *Proc. Int. Conf. VLSI Design*, pp. 193–198, Jan. 1998.

[8] S. Dey, V. Gangaram, and M. Potkonjak, "A controller-based design-for-testability technique for controller-datapath circuits," in *Proc. Int. Conf. Computer-Aided Design*, pp. 534–540, Nov. 1995.

[9] M. Sipser, *Introduction to the Theory of Computation*. PWS Publishing Company, Boston, 1997.

[10] Z. Kohavi, *Switching and Finite Automata Theory*. McGraw-Hill, New York, 1978.

[11] J. Levine, T. Mason, and D. Brown, *lex and yacc*. O'Reilly Associates, Sebastopol, 1992.

[12] K. T. Cheng and V. D. Agarwal, "A partial scan method for sequential circuits with feedback," *IEEE Trans. Comput.*, vol. 39, pp. 544–548, Nov. 1990.

[13] D. H. Lee and S. M. Reddy, "On determining scan flip-flops in partial-scan designs," in *Proc. Int. Conf. Computer-Aided Design*, pp. 322–325, Nov. 1990.

[14] J. Steensma, F. Catthoor, and H. De Man, "Partial scan at the register transfer level," in *Proc. Int. Test Conf.*, pp. 488–497, Sept. 1993.

[15] K. D. Wagner and S. Dey, "High-level synthesis for testability: A survey and perspective," in *Proc. Design Automation Conf.*, pp. 131–136, June 1996.

[16] S. Bhattacharya and S. Dey, "H-SCAN: A high level alternative to full-scan testing with reduced area and test application overheads," in *Proc. VLSI Test Symp.*, pp. 74–80, Apr. 1996.

[17] S. Bhattacharya, S. Dey, and F. Brglez, "Provably correct high-level timing analysis without path sensitization," in *Proc. Int. Conf. Computer-Aided Design*, pp. 736–742, Nov. 1994.

[18] J. A. Brzozowski and E. Leiss, "On equations for regular languages, finite automata, and sequential networks," *Theoretical Computer Science*, vol. 10, pp. 19–35, 1980.

[19] R. K. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Boston, 1984.

[20] P. Marwedel, *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, 1995.

[21] V. Chickermane, *Design and Synthesis for Testability using Architectural Descriptions*. PhD thesis, UIUC, 1993.

[22] E. Sentovich et al., "Sequential circuit design using synthesis and optimization," in *Proc. Int. Conf. Computer Design*, pp. 328–333, Oct. 1992.

[23] A. Greiner and F. Pecheux, "ALLIANCE: A complete set of CAD tools for teaching VLSI design," tech. rep., Laboratoire MASI/CAO-VLSI, Institut de Programmation, Universite Pierre et Marie Curie, Paris, 1993.

[24] T. Niermann and J. H. Patel, "HITEC: A test generation package for sequential circuits," in *Proc. European Design Automation Conf.*, pp. 214–218, Feb. 1991.

[25] T. Niermann, W. T. Cheng, and J. H. Patel, "PROOFS: A fast, memory efficient sequential circuit fault simulator," in *Proc. Design Automation Conf.*, pp. 535–540, June 1990.