Synthesis of Controllers for Full Testability of Integrated Datapath-Controller Pairs

Joan Carletta Mehrdad Nourani Christos Papachristou
Department of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, OH 44106

Abstract

This work facilitates the testing of datapath-controller pairs in an *integrated* fashion, with datapath and controller tested together in a single test session. Such an approach requires less test overhead than an approach that isolates datapath and controller from each other during test. The ability to do an integrated test is especially important when testing core-based embedded systems. The key to the approach is a careful examination of the relationship between techniques for controller synthesis and the types of gate level controller faults that can occur. A method for controller synthesis is outlined that results in a fully testable controller, so that full fault coverage of the controller can be achieved without any need for isolation during test.

1 Introduction

This work addresses the problem of testing systems that consist of interacting datapaths and controllers. Typically, datapaths and controllers are synthesized and tested independently. The majority of synthesis systems work by first designing the datapath based on the desired behavior, and then implementing the controller independently based on control flow information. Similarly, testing of datapaths and controllers is often done independently. However, even if the datapath and controller are designed such that they are 100% testable taken separately, when the two are taken in combination the testability may be severely degraded [6]. Few, if any, synthesis tools address the issue of how to test datapath and controller in an integrated way. Our work is motivated by two main issues: (a) the need to test an entire system, composed of an interacting datapath and controller, realistically and at speed, without neglecting the interface used for communicating between the two; and (b) the area overhead advantage obtained by observing controller faults through the datapath registers, rather than adding test hardware at the interface so as to be able to observe the controller outputs directly.

In this work, we build on previous work on testing datapaths in the context of a datapath-controller pair [14], and on techniques for observing controller faults through the datapath registers, so that the controller can also be tested in the context of the pair [13]. Here, we attack the problem of controller testing in an integrated system environment by looking at controller synthesis issues, so that the controller can be designed in such a way that it is easily tested in the context of a datapath-controller pair. This will require a careful consideration of the meaning of "don't care" specifications for the controller. Careful treatment of "don't cares" in logic optimization has been considered by many researchers for different purposes, including testing [5], control optimization [1], BDD optimization [12] and formalism [2]. References [11] and [4] discuss methods for high performance controller synthesis.

2 Background

Behaviorally, the datapath is represented by a *data flow graph* (DFG), in which nodes represent operations such as addition and multiplication, and edges represent the transfer of data. Structurally, the datapath consists of arithmetic logic units (ALUs), multiplexers, registers, and busses, and is responsible for all data computations. Figure 1 shows the active components in a typical control step and their corresponding control signals. We note that the approach presented in this paper is not restricted to the exact form shown in this figure; for example, the work also applies to datapaths with more than one multiplexer along the path of a single register-to-register transfer.

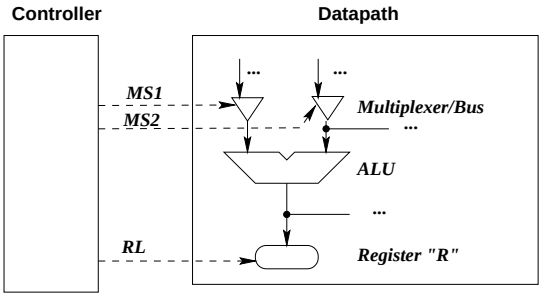


Figure 1: One datapath register and the control signals that affect it.

Behaviorally, the controller is viewed as a state diagram that specifies the control steps in which the various operations in the data flow graph are done. For this work, controllers are implemented structurally as finite state machines using random logic. The controller guides the datapath in making computations by supplying it with control signals governing which path is selected through each multiplexer and which registers are loaded at each control step.

We classify stuck-at faults internal to the controller into several groups [13]. Faults that never affect the input-output behavior of the *synthesized* controller in normal mode are *controller-functionally redundant* (*CFR*), while faults that affect the output of the controller in at least one control step when the controller is running in normal mode are *controller-functionally irredundant* (*CFI*). The work in [7] shows that controller resynthesis can be used to remove CFR faults if they are a concern. CFI faults are further divided into two subgroups. *System-functionally irredundant* (*SFI*) faults change the input-output behavior of the controller-datapath pair *as a system*. One example of an SFI fault is a stuck-at zero on a register load line in a control step when the register is supposed to receive the result of a computation; this will cause a noticeable problem, since the result of the computation will never be written and will therefore be lost. *System-functionally redundant* (*SFR*) faults do not change the input-output behavior of the system as a whole, even though they do affect the input-output behavior of the controller. One example of a system-functionally redundant fault is a fault that affects a multiplexer select line only in those control steps when the multiplexer is idle; in these idle control steps, the multiplexer select line has a "don't care" specification.

It is the SFR faults that cause difficulty when attempting to test a controller without isolating it from the environment of the system. These faults can not be observed indirectly, through the registers of

the datapath; their effects can be seen only directly at the controller output. We found that for traditionally synthesized controllers, it can easily be the case that 20% of the faults in the controller are SFR. The focus of this paper is to show how controller synthesis can be done in such a way that no faults in the controller will be SFR.

3 Synthesis and the Location of Fault Sites

The key to the method outlined in this paper is an understanding of the relationship between how the controller is synthesized and the stuck-at fault sites in the resulting hardware. Our goal is to avoid having fault sites that correspond to system-functionally redundant faults. As a preliminary step, we explore the effect of various faults on a single piece of output logic from a finite state machine synthesized using a standard sum-of-products minimization technique. In Figure 2, we show an example piece of gate level logic, $F = \overline{S_3}\,\overline{S_1} + S_2\,\overline{S_1}\,S_0 + S_3\,\overline{S_2}\,S_1$, along with the corresponding Karnaugh map.

Under the single stuck-at fault model, faults in a sum-of-products can occur in one of three categories of places:

Category 1. At the output of the OR gate.

Category 2. At an input to the OR gate (the output of an AND gate).

Category 3. At an input to an AND gate.

Considering that a stuck-at fault can be either stuck-at one or stuck-at zero, we have six different types of faults to consider. Some types are equivalent. The types are:

Category 1 stuck-at zero. This type, represented by fault *A* in Figure 2, causes the output to always be zero, so that all prime implicants drop out of the K-map. This is shown in Figure 3(a).

Category 1 stuck-at one. Equivalent to Category 2 stuck-at one. This type, represented by fault *B* in Figure 2, causes the output to always be one, so that in effect the prime implicants grow to cover all of the K-map. This is shown in Figure 3(b).

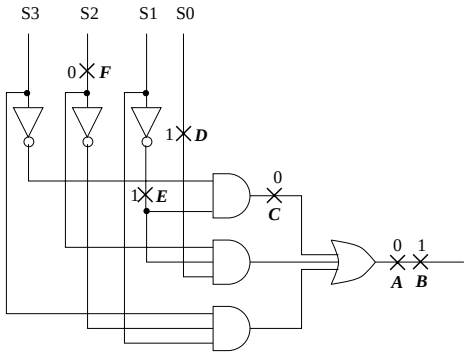
Category 2 stuck-at zero. Equivalent to Category 3 stuck-at zero. This type causes a single prime implicant (the one computed by the AND gate whose output is stuck-at zero) to drop out of the sum-of-products, in effect being removed from the K-map. For example, a category 2 stuck-at-zero fault shown as fault *C* in Figure 2 causes the $\overline{S_3}\,\overline{S_1}$ product to drop completely out of the sum-of-products, as shown in Figure 3(c).

Category 3 stuck-at one. This type causes a single variable to drop out of a product term in the sum-of-products. In effect, one prime implicant (the one computed by the AND gate whose input is stuck) grows in size by one dimension. For example, a category 3 stuck-at one fault shown as fault *D* in Figure 2 causes the $S_2\,\overline{S_1}\,S_0$ product to change to $S_2\,\overline{S_1}$. The effect on the K-map, shown in Figure 3(d), is for the prime implicant to grow to cover four cells instead of the intended two.

A stuck-at zero fault of any category causes at leat one prime implicant to drop out of the sum, while a stuck-at one fault of any category causes at least one prime implicant to grow in at least one dimension.

Because the only place where we can have fanout in a sum-of-products structure is directly at the inputs, the only other kind of faults that we have to worry about are the faults at the fanout sources themselves. These faults will act as multiple category 1 faults. For example, a stuck-at one fault shown as fault *E* on Figure 2 will cause two prime implicants to each grow in the S_1 dimension, as shown in Figure 3(e). The stuck-at 0 fault shown as fault *F* on Figure 2 will cause one prime implicant to drop out of the sum, and one to grow in the S_2 dimension, as shown in Figure 3(f).

An understanding of the relationship between logic level synthesis, the types of single stuck-at faults that can occur, and the effect of those



(a) gate level circuit with fault sites marked.

S1 S0 \ S3 S2	00	01	11	10
00	1	1	0	0
01	1	1	1	0
11	0	0	0	1
10	0	0	0	1

$F = \overline{S_3}\,\overline{S_1} + S_2\,\overline{S_1}\,S_0 + S_3\,\overline{S_2}\,S_1$

(b) fault-free K-map

Figure 2: An example sum-of-products with designated stuck-at faults.

S1 S0 \ S3 S2	00	01	11	10
00	***0***	***0***	0	0
01	***0***	***0***	***0***	0
11	0	0	0	***0***
10	0	0	0	***0***

$F = 0$

(a) for fault *A*

S1 S0 \ S3 S2	00	01	11	10
00	1	1	***1***	***1***
01	1	1	1	***1***
11	***1***	***1***	***1***	1
10	***1***	***1***	***1***	1

$F = 1$

(b) for fault *B*

S1 S0 \ S3 S2	00	01	11	10
00	1	1	0	0
01	1	1	***0***	0
11	0	0	0	1
10	0	0	0	1

$F = \overline{S_3}\,\overline{S_1} + S_3\,\overline{S_2}\,S_1$

(c) for fault *C*

S1 S0 \ S3 S2	00	01	11	10
00	1	1	***1***	0
01	1	1	1	0
11	0	0	0	1
10	0	0	0	1

$F = \overline{S_3}\,\overline{S_1} + S_2\,\overline{S_1} + S_3\,\overline{S_2}\,S_1$

(d) for fault *D*

S1 S0 \ S3 S2	00	01	11	10
00	1	1	0	0
01	1	1	1	0
11	***1***	***1***	***1***	1
10	***1***	***1***	0	1

$F = \overline{S_3} + S_2\,S_0 + S_3\,\overline{S_2}\,S_1$

(e) for fault *E*

S1 S0 \ S3 S2	00	01	11	10
00	1	1	0	0
01	1	1	***0***	0
11	0	0	***1***	1
10	0	0	***1***	1

$F = \overline{S_3}\,\overline{S_1} + S_3\,S_1$

(f) for fault *F*

Figure 3: K-maps for the example in the presence of the designated faults.

faults on the functionality of the controller is crucial to developing a synthesis-for-testability approach. When a controller is tested in an isolated environment, with its outputs directly observed, any change in its functionality can be readily observed. However, when the controller is tested as an integral part of a datapath-controller pair, any change in functionality must be observed indirectly by observing the effect of the change on the way that the datapath functions.

4 Effect of Don't Care Specifications

Figure 1 shows a datapath fragment with two multiplexers, an arithmetic logic unit, and a register, along with the control lines for which logic must be synthesized.

4.1 Don't Cares in Multiplexer Select Lines

In the general case, a register in the datapath will be loaded in only some of the control steps. In any control steps in which the register is not loaded, we don't care about the values of the multiplexer select lines. We also do not care about the values of the multiplexer select lines in any control states that are not ordinarily used, i.e., that are not reachable from the reset state. Such control states are present when the number of control steps required for the datapath is not an exact power of two.

A change in any "care" value on a multiplexer select line will be observable not only directly at that line, but also at the output of the datapath. That is because the change will cause the datapath to do a computation with incorrect data. Unless the datapath itself is designed to do redundant computations (a case that is beyond the scope of this paper, and will cause a great deal of testability problems), supplying incorrect data for even an intermediate computation will affect the final computation made by the datapath, provided that computations are done for a reasonably large range and number of input test patterns.

A change in a "don't care" value on a multiplexer select line, while observable directly at the select line, will not be observable at the datapath output. While such a change will cause a change in signal values local to the area around the multiplexer and arithmetic logic unit, those changes will never be saved to the register and propagated further. This means that in order to ensure that the controller output logic computing the multiplexer select lines is fully testable in an integrated datapath-controller environment, we must be sure that no single stuck-at fault in that logic affects only don't care values. Fortunately, to do so requires only that we follow the same minimization technique that we would follow to minimize the area of the output logic for a single line: we choose prime implicants in the K-map to be as large as possible without covering any zeroes. Then, removing any prime implicant (as any stuck-at zero fault of category 1, 2, or 3 would do) will certainly change a "care" specification; the prime implicant would not have been included in the minimal sum if the only unique cells that it had to contribute to the sum were don't care cells. In addition, growing any prime implicant (as any stuck-at one fault of category 1, 2, or 3 would do), will also certainly change a "care" specification; by definition, a prime implicant can not be expanded in any dimension without circling a "care" 0-cell.

What this means is that if we synthesize separate output logic for each one of the multiplexer select lines so as to minimize the area of that separate logic, taking don't cares until full account, we will end up with multiplexer select logic that is fully testable in a integrated datapath-controller environment; any fault will affect not only controller functionality, but also functionality of the system as a whole.

4.2 Don't Cares in Register Load Lines

We now explore the meaning of don't care specifications on the register load lines of the datapath. We again use the example fragment of a datapath in Figure 1. In some control steps, the schedule specifies that the register should be loaded to record the result of a computation in the arithmetic logic unit. Such a load is termed *necessary*. Each necessary load becomes a "care" 1-cell in the K-map specification for the register load line logic.

For control steps in which no load is specified by the schedule, loading is not necessary. If, by looking at the register transfer that takes place when a load is done, we discover that the load overwrites some needed data with garbage, thereby disrupting the functionality of the datapath, we term the load *malignant*. If, however, the load, while unnecessary, is harmless in the sense that the load does not change the functionality of the datapath, we term the load *benign*. A load that overwrites a live variable with other data is clearly malignant. However, a load that changes the contents of a register in a time step when that register holds no live variable essentially writes garbage over garbage, and is benign. Further, it is possible for a load to overwrite a live variable, but with another copy of the same data. This happens when the multiplexer select lines still point to the same data that was used when originally writing the data. Such loads are also benign.

For control steps in which a register load is malignant, the corresponding cell of the K-map for the register load line must be a "care" 0-cell, to ensure that the load is not done. For control states in which a register load is benign, the corresponding cell of the K-map for the register load line is a "don't care" *d*-cell.

We can now synthesize separate output logic for each one of the register load lines, taking the don't cares into full account in the same manner that we did for the multiplexer select lines. Because any fault in the resulting sum-of-products implementation will cause a prime implicant to either drop out completely or grow by one or more dimensions, this technique ensures that any fault will affect a "care" specification for the register load line. A fault that causes a prime implicant to drop out will cause some necessary load to not be done, which will be observable even at the datapath outputs, because the results of some computation important to the datapath functionality will not ever be written. A fault that causes a prime implicant to grow will cause a malignant load to be done, which will disrupt the datapath functionality and therefore also be easily observed at the datapath outputs. We remark here that the algorithm synthesizes the multiplexer select lines separately from the register load lines, and therefore may miss the opportunity for multi-output minimization.

5 The Basic Synthesis Approach

The overall process for synthesizing controller output logic so that it can be fully tested within an integrated datapath-controller pair is described next. We are given a scheduled data flow graph, a RTL datapath, and a state encoding for the controller. The algorithm first uses the pattern of register loads in the schedule to determine where the 0-cells, 1-cells, and *d*-cells are in the K-maps for the multiplexer select lines, and then synthesize the multiplexer select lines, using separate logic for each line. The synthesis uses the traditional approach for minimizing area by taking full advantage of the don't care cells. Note that this approach ensures that any single stuck-at fault in the multiplexer select line logic will cause a change in a "care" condition, and therefore be detectable.

Next, the algorithm synthesizes the register load lines. The major work here is to decide for each control step and each register whether a load is necessary, malignant, or benign. Note that this can not be done until the multiplexer select lines have been synthesized, since the result depends on exactly what register transfer would be done were a load to take place. If the schedule shows a register loading at the control step, the load is necessary. If the register is not holding a live variable during the control step, the load is benign. If the register is holding a live variable, and either the multiplexer select values or the variables in the source registers have changed, the load is malignant. If the register is holding a live variable, and another load

will serve simply to re-load the same value a second time, the load is benign. Necessary, malignant, and benign loads correspond to 1-cells, 0-cells, and *d*-cells in the K-map for the register load line, respectively. The last step of the algorithm is to synthesize the register load lines, using separate logic for each line. This synthesis uses the traditional approach for minimizing area by taking full advantage of the don't care cells. Again, this approach ensures that any single stuck-at fault in the register load line logic will cause a change in a care condition, either not loading a register when it should be, or doing an extra load that writes garbage over important data. In either case, data important to the calculation performed by the datapath is lost, so the datapath's functionality is disrupted, and the fault is testable.

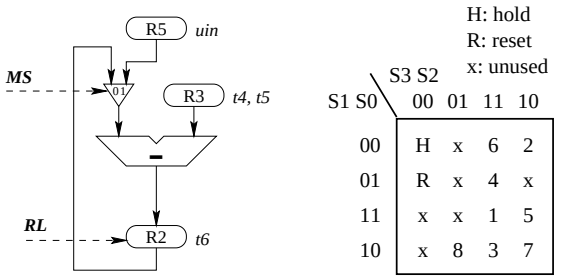
We now illustrate the idea behind the synthesis technique using a fragment of the datapath for an example that implements a differential equation solver. Figure 4(a) shows one arithmetic logic unit (a subtractor) from the datapath, along with the surrounding multiplexers and registers. Figure 4(c) shows the lifespans of the variables bound to the registers, along with the part of the schedule that pertains to the arithmetic logic unit. The arithmetic logic unit performs scheduled computations in two control steps; in control step 4, it computes $t6 \leftarrow uin - t4$ using register transfer $R2 \leftarrow R5 - R3$, and in control step 5, it computes $uvar \leftarrow t6 - t5$ using register transfer $R2 \leftarrow R2 - R3$.

From the schedule and binding information, we know that *MS*, the multiplexer select signal for the datapath fragment, must be a "1" in control step 4, and a "0" in control step 5. In all other control steps, we do not care what the value is, because register *R2* will not be loaded. For this example, we have chosen the arbitrary state encoding shown in Figure 4(b), where the number in the K-map cell tells which control step is mapped to that cell. As a result, the K-map for *MS* is as shown in Figure 5(a). After minimizing the logic, we arrive at the expression $MS = \overline{S_1}$. Note that were we to choose another expression for *MS*, we would be adding some redundant logic to the circuit; we would require extra gates that created "0" or "1" values in control steps where we do not care what the created value is. This is very likely to happen if we do multiple output minimization to minimize logic over a number of multiplexer select lines.

Once we know the exact synthesized expression for the multiplexer select lines, we can determine whether register loads will be necessary, malignant, or benign. It is necessary to the functionality of the datapath to load register *R2* in control steps 4 and 5. In control step 6, loading register *R2* is malignant. We know this by first looking at the K-map for for the *MS* line, and noting that the synthesized value for *MS* is a 1. This means that were a load done in control step 6, it would implement the register transfer $R2 \leftarrow R5 - R3$. Referring to the chart of variable lifespans, we see that at control step 6, *R5* holds the live variable *uin*, and the last variable written to *R3* was *t5*, so (in the absence of multiple faults) *R3* will still hold *t5*, even though the variable is no longer live. This means that an extra load of *R2* in control step 6 would store *uin - t5* in *R2*, overwriting the live variable *uvar* ($= t6 - t5$) with an erroneous value. This erroneous value will be used instead of *uvar* in subsequent computations, disrupting the datapath's functionality.

We now use the analysis of the effect of register loads to fill a K-map for the register load line with an appropriate specification. Necessary register loads correspond to care 1-cells. Malignant register loads correspond to care 0-cells. Benign register loads correspond to don't care cells. The K-map for the register load line *RL* of our example is shown in Figure 5(b). In order to synthesize a fully testable controller, we again use the traditional synthesis method for minimizing area, taking the don't cares into full account. As a result, we choose $RL = S_0$.

For all control steps other than steps 4, 5, and 6, there is no live variable stored in *R2*. This means that any extra loads done in these control steps are benign; they do not disrupt the computation being performed by the datapath. We remark that it is also possible to have a benign load that simply re-writes the value of a computation an extra time, although that case does not show up in this example.

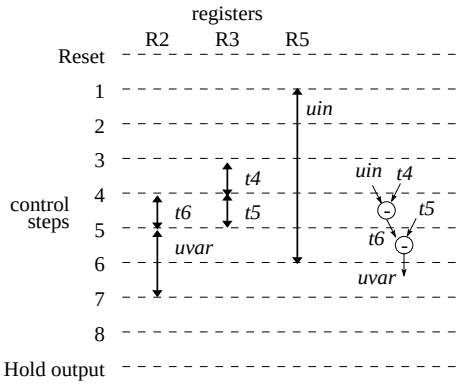


(a) datapath fragment.

H: hold
R: reset
x: unused

S1 S0 \ S3 S2	00	01	11	10
00	H	x	6	2
01	R	x	4	x
11	x	x	1	5
10	x	8	3	7

(b) state encoding.



(c) variable lifespans and schedule fragment.

Figure 4: A fragment of an example that implements a differential equation solver.

S1 S0 \ S3 S2	00	01	11	10
00	d	d	d	d
01	d	d	1	d
11	d	d	d	0
10	d	d	d	d

(a) K-map for MS line.

S1 S0 \ S3 S2	00	01	11	10
00	d	d	0	d
01	d	d	1	d
11	d	d	d	1
10	d	d	d	d

(b) K-map for RL line.

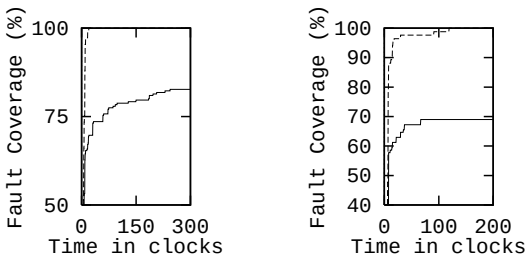
Figure 5: Synthesis for the example.

6 Results

In this section, we show results using two example circuits. Our first example implements a differential equation solver and is a standard high level synthesis benchmark, and our second example is another high level benchmark known as the FACET example [8]. The circuits have been synthesized from high level descriptions using the SYNTEST synthesis system [10]. The output of SYNTEST is a register transfer level datapath and state diagram controller. Logic level synthesis for the traditional approach is done using the ASIC Synthesizer from the COMPASS Design Automation suite of tools [3], using a finite state machine implementation for the controller and based on a 0.8-micron CMOS library. Logic level synthesis for the proposed approach, which eliminates system-functionally redundant faults within the controller, is done using the algorithm of Section 5. The test pattern generation registers (TPGRs) necessary for built-in self-test (BIST) are synthesized using COMPASS's Test Compiler. Fault coverage curves are found for the resulting logic level circuits using AT&T's GENTEST fault simulator [9]. GENTEST uses a single

	traditional synthesis	SFR-free synthesis	
Diffeq	671	336	(a) in transistors.
FACET	607	230	
Diffeq	962x632	713x444	(b) in λ^2.
FACET	938x544	513x444	

Table 1: Size of controllers for the two examples.



(a) differential equation solver. (b) FACET example.

Figure 6: Fault coverage curves for the controllers using traditional synthesis (solid line) and SFR fault-free synthesis (dashed line).

stuck-at fault model. The probability of aliasing within the MISRs is neglected, as are faults within the TPGRs and other test circuitry. Although the datapath and controller are tested together, we have separated out fault coverage curves and area figures for the controller to clarify the results.

As the basis of comparison, we show fault coverage and area results for controllers synthesized both in the traditional way and using our proposed technique. Table 1 shows the relative sizes of the traditionally-synthesized and SFR-free controllers. It is not surprising that the SFR-free controllers are significantly smaller than the ones synthesized with a traditional approach and a gated clock scheme. The gated clock scheme demands that all unnecessary loads, both malignant and benign, be synthesized to be zeroes. This scheme saves power, since it loads registers (and changes the signals at the inputs to the logic driven by the registers) only when necessary for the functionality of the datapath. However, additional logic must be used to ensure that the loads are not done; there are no "don't cares" in the register load specifications. By the same token, the power consumed by the systems using the SFR-free controllers will be significantly higher than those using the traditionally synthesized, gated clock controllers, not because of the power consumed in the controllers themselves, but because of power consumed in the datapath as a result of unnecessary loading. Thus, the main tradeoff is between testability and power consumption.

Fault coverage results for the examples are shown in Figure 6. Fault coverage is for the controller only. On the fault coverage graphs, the vertical axes show fault coverage as the percentage of controller faults detected, and the horizontal axes show time as a function of clock cycles. During each of the tests, the controller is run in normal mode, and the only points used for observation are those normally available as outputs of the system, i.e., the data outputs of the datapath, and the "done" signal created by the controller to signal that a computation is complete. Thus, the controller is tested as a truly integrated part of the datapath-controller pair; it is not isolated from the system in any way during the test.

Clearly, the use of gated clocks can cause testability problems that make it impossible to achieve 100% testability of the controller without separating the controller from the datapath during test. Synthesizing logic to force the load lines to be zeroes in all unnecessary control steps, even those in which the actual value of the load line does not matter, results in more logic. The additional logic is redundant when the controller and datapath are viewed as a single, integrated system.

7 Conclusions

This work presented an approach for the synthesis of finite state machine controllers that results in controllers that are fully testable even without separating the controller from its environment (i.e., from the datapath that it controls) during the test. One key to the approach is an understanding of the nature of system-functionally redundant (SFR) faults, which can not be caught with an integrated testing technique. The second key is a study of the effect of logic synthesis on whether or not synthesized logic contains SFR fault sites. By providing a way to test a datapath-controller pair without intervening test hardware, this work also comes one step closer to the successful test of core-based embedded systems; for these systems, it is simply not possible to add test hardware after the fact.

References

[1] R. Bergamaschi, D. Lobo and A. Kuehlmann, "Control Optimization in High-Level Synthesis Using Behavioral Don't Cares," in *Proc. Design Auto. Conf.*, June 1992, pp. 657-661.

[2] D. Brand, R. Bergamaschi and L. Stok, "Don't Cares in Synthesis: Theoretical Pitfalls and Practical Solutions," *IEEE Trans. on CAD*, April 1998, pp. 285-304.

[3] Compass Design Automation, "User Manuals for COMPASS VLSI V8R4.4," Compass Design Automation, Inc., 1993.

[4] A. Crews and F. Brewer, "Controller Optimization for Protocol Intensive Applications," *Proc. EURO-DAC Conf.*, 1996.

[5] S. Devadas, H. Ma and A. Newton, "Redundancies and Don't Cares in Sequential Logic Synthesis," *Journal of Electronic Testing: Theory and Applications*, Jan. 1990.

[6] S. Dey, V. Gangaram and M. Potkonjak, "A Controller-Based Design-for-Testability Technique for Controller-Datapath Circuits," *Proc. Int'l. Test Conf.*, October 1995.

[7] F. Fummi, D. Sciuto, and M. Serra, "Synthesis for Testability of Large Complexity Controllers," *Proc. Int'l. Conf. on Computer Design*, pp. 180–185, October 1995.

[8] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers: Boston, MA, 1992.

[9] AT&T, "User Manuals for GENTEST_S 2.0," AT&T Bell Laboratories, 1993.

[10] H. Harmanani, C. Papachristou, S. Chiu and M. Nourani, "SYNTEST: An Environment for System-Level Design for Test," *Proc. EURO-DAC 92*, Sept. 1992.

[11] A. Hertwig and H. Wunderlich, "Fast Controllers for Data Dominated Applications," *Proc. Int'l. Test Conf.*, 1997.

[12] Y. Hong, P. Beerel, J. Burch and K. McMillan, "Safe BDD Minimization Using Don't Cares," *Proc. Design Auto. Conf.*, June 1997.

[13] M. Nourani, J. Carletta and C. Papachristou, "A Scheme for Integrated Controller-Datapath Fault Testing," *Proc. Design Auto. Conf.*, pp. 546–551, June 1997.

[14] C. Papachristou and J. Carletta, "Test Synthesis in the Behavioral Domain," *Proc. Int'l. Test Conf.*, pp. 693–702, October 1995.