ATPG in Alloy4:
Automatic Test Pattern Generation for combinational circuits in Alloy4

Samuel A. Kirk
http://www.samkirk.com
sak@samkirk.com
August 16, 2010

Abstract

The Alloy model given in this paper is a very simple example of how to do Automatic Test Program Generation (ATPG) for a stuck-at type fault model in an electronic digital electronic circuit. This example demonstrates that ATPG via Boolean differential equations is easy, elegant and very straight-forward when done in a desk-calculator style in Alloy4. It also shows that ATPG can properly be viewed as a form of Model-Driven Testing (MDT), aka Model-Based Testing (MBT).

"Examples are the test cases of our concepts."

1. Introduction

The problem of automatically generating a test pattern presupposes a particular kind of fault, i.e., a fault model. There are many different ways a circuit can fail and each such failure mode can be modeled. E.g., an open connection or break in the electrical conduit or path between two components. [Cite a text book reference here that lists over two pages of them.] A very common kind of fault is one in which the connector between two components are stuck hi (to Vcc, the power supply) or stuck lo (to ground). The ATPG problem is to construct a set of input vectors which will detect any such fault as indicated by getting a different output than had that fault not been present. Essentially, any solution entails a comparison between the behavior (output) of a fault-free circuit with a faulty one that makes a difference. Moreover, if there is no such input vector (as they call it), the fault is untestable.

ATPG is essential for expedient testing of combinational logic circuits. There are several different approaches. Mathematically, ATPG is a matter of performing the boolean differential of the circuit with respect to the fault. Using Boolean differences is not only mathematically sound but also an intuitively appealing way, to do ATPG using the paradigm of everyday circuit debugging:

> Suppose that the x input is suspected of being fault. (Perhaps the bonding wire between the x pin and the IC bonding pad is broken.) A reasonable test for this condition would be to change the signal on x and observe the f output to see f changes in response to the x change. Of course, the other input signal must be hold fixed at values that make f dependent on x. If f is not vacuous in x, such values must exist." (McCluskey, 1986, pp. 177-178 Section 5.5, "Boolean Difference")

However,

> "Early test generation algorithms such as boolean difference and literal proposition were not practical to implement on a computer." ("Automatic test pattern generation - Wikipedia," n.d.)

I content that with Alloy4, this is no longer the case for boolean difference as demonstrated by my very simple example.

Samuel A. Kirk, http://www.samkirk.com.

The ATPG problem is an NP-complete one. As pointed out by (Stephan, Brayton, & Sangiovanni-Vincentelli, 1996, p. 1), this was proved by (Ibarra & Sahni, 1975) using satisfiability (SAT). (See also (Biere & Kunz, 2002) for an interesting account of the contention between ATPG and SAT.) It turns out that Alloy is essentially compiler for a SAT solver:

> The Alloy Analyzer is therefore a constraint solver for the Alloy logic. In its implementation, however, it [Alloy] is more of a compiler, because, rather than solving the constraint directly, it translates the constraint into a boolean formula and solves it using an off-the-shelf SAT solver. (Jackson, 2006, p. 150)
>
> The choice of SAT solver can be set in the Alloy Analyzer's preference panel. For most analyses, the choice of the SAT solver doesn't matter. MiniSat and ZChaff are good choices for small problems; for larger problems, Berkmin seems to perform best. ["Which SAT solver should I tell Alloy to use?" in ("Alloy FAQ," n.d.)]

So, Alloy guarantees the computational power required to ATPG because it uses very good SAT solvers.

This paper contends that a simple application of Alloy4 will do the Boolean differential approach to ATPG very nicely as shown by the example below.

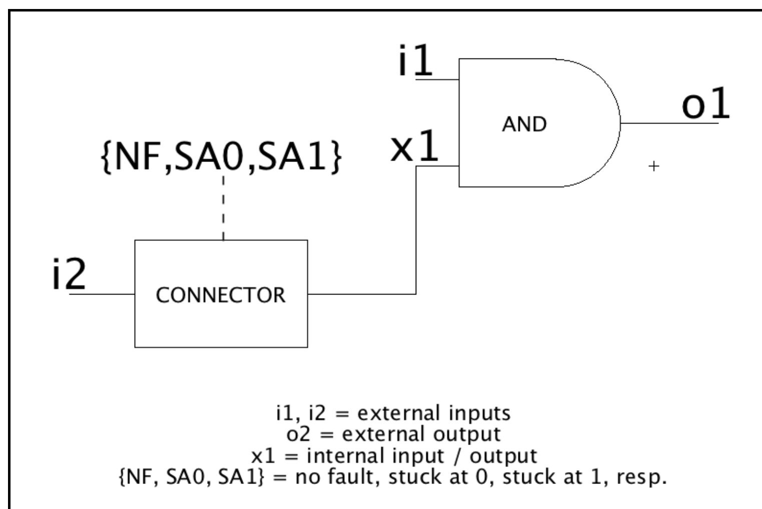

Fig 1. - Electronic Circuit Modeled with Stuck-At Fault

2. Modeling Stuck-At Faults in Alloy - `Connect`

The faults in this Alloy model are stuck-at-one and stuck-at-zero and they apply to a CONNECTOR. In the traditional, electronic domain-specific diagram [Fig 1], the CONNECTOR is drawn as a box. We can imagine it to be a path of any shape, so long as its source, an input terminal here, is connected with its destination, an input of the AND gate.[1] If the connector itself is grounded anywhere, all the inputs are stuck at zero. And likewise for stuck at one. This is reflected by the dotted line between the expression "{NF, SA0, SA1}" and the CONNECTOR itself indicates a meta-relationship such that:

- For no fault, NF, the connector behaves ideally.

1 Although not used in this very simple example here, fan-outs with this connector model are allowed to feed multiple inputs from a single output. Imagine a connector in the shape of a blob that has a single point of connection on its left (for the input to the connector) and several on its right (for the fanned outputs of the connector).

- For a stuck at 0 fault, SA0, the connector conveys a boolean value of 0 to its output no matter what its input.
- Likewise, for a stuck at 1 fault, SA1, the connector conveys a boolean value of 1 to its output no matter what its input is.

Corresponding to the CONNECTOR in the domain-specific diagram is the essential part of this Alloy model: the `Connect` function (line 10).

Thus, the fault model is given in this Alloy application as an explicit, discrete component of the model that can be disabled or otherwise configured for one of two fault modalities. More than one instance of the `Connect` fault model is allowed to model multiple stuck at faults. In fact, you can mix stuck-at-0 with stuck-at-1 faults. But each instance of a fault is specified precisely one way.[2]

Stuck fault modeling is said to apply to structural tests. I.e., the target of the tests are the connections between the components, not the components themselves. (In practice, the pins of IC packages are included.) We are thus said not to be component testing but rather, structural testing. The components themselves are *assumed* to be fault-free even if they really aren't. If the component truly has a fault that mimics a structural fault, structural testing is mute about that.

3. Modeling Circuit Structure in Alloy - `Network`

There is one-to-one correspondence between the labels listed in the legend of the circuit diagram [Fig 1] and the parameters of `Network` predicate (line 21). [See "APPENDIX – Source Code Listing" for all line number references.] The `Network` predicate takes all the nodes, both external and internal, that define the content of the circuit along with an additional parameter that specifies what kind of fault that specific a given instance of that circuit is to obey.

The `Network` predicate combines instances of the `Connect` function and any Boolean functions from the boolean module (opened in line 3) to comprise the specific circuit under test (lines 26 and 27). In this very simple circuit, the Boolean function used is the `And` function (line 26) and corresponds to the AND gate in the upper right-hand section of the diagram.

`Network` defines a specific circuit, i.e., an instance of a conceptually more general circuit. Although Alloy is adept in expressing predicates at the level of theorems, we are not using that capability here. In focusing to a specific circuit under test (rather than a properties of circuits in general), we are writing in what I like to call the calculator style (as opposed to a theoretic style) in Alloy.[3]

Observe that there is only one definition of the circuit under test, viz., `Network`, and that it may be configured one of three different ways via the `Fault` parameter.

---

2 "You can't have it every which way," as Dr. Turnbull, former chair of The Ohio State University Philosophy Dept, used to say.

3 Such "theorems" would include checking that the circuit under test is a DAG (Directed Acyclic Graph) and thus doesn't contain any feedback loops. Another would check that the circuit is connected and thus does not have any islands or orphans. These predicates (and/or functions) would best be packaged as a separate module that the end-user would open in a single line of code (similar to line 3) and thus keep his application uncluttered with the "theoretic" support module. Of course, the graph module in Alloy is an excellent candidate for such a module! [TBD- check that there is such a module by that name.]

4. Doing the Analysis in Alloy - `Compare`

As explained in the first paragraph of the Introduction, a Boolean differential analysis entails a comparison between the faulted and non-faulty circuit that makes a difference in the output with a commonality in the external inputs. Since the fault itself is a connector, the impacted fault site gets modeled as an internal node, in this case, x1, an input to the AND gate from a terminal (external) input. Internal nodes connected to the faulty/stuck connector may very well behave differently in the faulty vs non-faulty case.

The `Compare` predicate (line 29) sets down all these conditions. Here, we use the convention in Alloy for primed variables, viz., `O1'`, `I1'`, `I2'`, and `X1'`, to represent the values of the nodes in the faulted version of the circuit. For the fault-free circuit, we use the unprimed counterparts viz., `O1`, `I1`, `I2`, and `X1`, to represent the non-faulty values. Please note that on the circuit diagram, the corresponding variables are written in corresponding lower case, viz., o1, i1, i2 and x1.

In effect `Compare` lays down two rules:

1. Lines 34 and 35 ensure the external inputs are the same; whereas
2. Line 37 ensures the outputs are different.

Lastly, lines 38 and 39 instantiate the non-faulty and faulty circuits, resp. In this example, line 39 induces a Stuck-at-1 fault. By changing SA1 to SA0 in that line, we would induce a Stuck-at-0 fault instead into the model. Mathematically speaking, we are performing a Boolean differential with respect to internal node variable X, the fault site, since there is both a `X1` and `X1'` with no restrictions placed upon them, i.e., X is free.

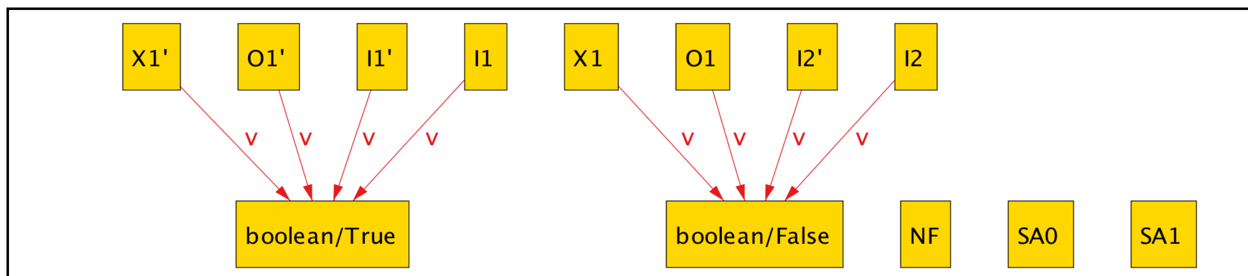

Fig 2. - Executed Alloy Model Results – A Unique Instance

5. Getting & Checking Results – Running the Alloy Model

When running this model, we get the results in the Alloy instance diagram [Fig 2]. The test vector reads as:

I1 = True, I2 = False ==> O1 = False if a stuck-at-1 is present on node x, O1 = True if fault-free.

We can also see that the rules laid down in the previous section are obeyed:

1. I1' = I1 = True
2. O1' = True, O1 = False ==> O1' ≠ O1

By clicking again for another instance in Alloy, you'll discover that you don't get any. This means that this test vector is unique.

We can also observe the organization of the model using the Meta-Model view of the model in Alloy4 [Fig 3] as a classical class hierarchy.

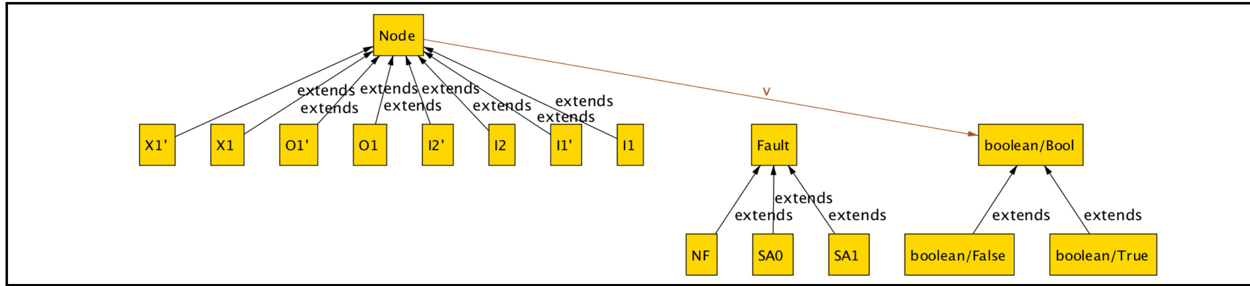

Fig 3. The Alloy Meta-Model – A Classical Class Hierarchy

6. Conclusion

Kindly note that you are spared the inconvenience of getting endless redundant solutions, i.e., solutions with the same value. Did you have to do any preliminary circuit minimization or other simplification? No. I contend that this is true for far more complex circuits when modeled in Alloy as done in this paper.

And did you see a SAT solver? No. But there indeed was one lurking magnificently behind the scenes in your behalf!

7. Future Work

Now that we've done a model of one fault for a very simple circuit, we would like to do it for more faults in bigger circuits and consider the resulting set of all such vectors. Questions typically posed in ATPG with regard to test efficiency are:

- *What is the minimum set of a test vectors?*
- *What then is the smallest subset of test vectors that will test all the (testable) faults? Is there more than one?*
- *Does it matter in what order we apply the vectors?*

8. Related Work

In (Bushnell & Agrawal, 2000, p. 161"Algorithm Types," sec 7.1.6), "Symbolic - Boolean Difference" is listed at just one type of ATPG algorithms. The others given there are:

- Exhaustive
- Random - Used with Algorithmic Methods
- Path Sensitization Methods
- Boolean Satisfiability and Implication Graph Methods (p. 164)

The Boolean Satisfiability Method is one that allows a rather direct application of a SAT solver. The modeling technique in Alloy I have given is far more indirect in as much as the SAT solvers are completely transparent to the end user. The Implication Graph Method is an approach using graphs to simplify the circuit so that the application of the SAT solver is more manageable. Note that Alloy4 has its own built-in method of simplification, a legacy it inherited from a earlier program called Kodkod. (“Kodkod,” n.d.)

(Wang, Wu, & Wen, 2006, p. 167 section 4.3) have a good presentation of the theoretical background on Boolean difference.

(Larrabee, 1992) is the classic approach to applying SAT solvers to boolean differences to do ATPG.

(Yang, Cheng, & Wang, 2004) improve upon SAT-based ATPG by utilizing the longest sensitizable path in the circuit under test.

(Mourad & Zorian, 2000, p. 61 Section 3.2.2 "Boolean Difference") contains a brief, 2-page summary of Boolean difference.

(McCluskey, 1986, p. 179 Table 5.5-1 "Properties of the Boolean Difference") is a tidy summary that should be on Wikipedia.

(Abramovici, Breuer, & Friedman, 1990) contains no discussion on boolean differences. However, chapter 6, p. 181, "Testing for Single Stuck Faults" is a good exposition of all the practical methods for the stuck-at fault model testing other than Boolean differences.

Alloy4 is a Java 1.5 application. ("Alloy (specification language) - Wikipedia," n.d.)is a good presentation and will tell you where to go to download Alloy (which is as slick as it can be). I used Alloy Analyzer 4.1.10 (build date: 2009/03/19) on a Mac OS 10.4.11 running dual G4's (the "silver drawer" Power Mac). Note: If Alloy4 required Java 1.6, I would have further problems running it since Apple doesn't support Java 1.6 on my particular Mac (which is considered a little out of date).

(Jackson, 2006) is the goto book if you're really going to do Alloy. Here:

http://softwareabstractions.org/sample-chapters/appendix.pdf

you can read the appendix from this book where Jackson dares to compare Alloy with its competitors, Z, B, VDM and OCM, as represented by their leading practitioners by having them write the model in their respective modeling language for a common specification. Doing a comparison like this is very difficult. Jackson accommodates them with great hospitality. I found this appendix to be sufficiently compelling to buying the book!

I have found that Chapter 2 is the key to understanding the rest of the book. Pay close attention to Chapter 2 and see if you can puzzle out its relationship to the rest of the book. The examples there are examples of what I like to call the "calculator style" of Alloy rather than the more general, abstract "theorem or theoretic style" of Alloy that are generally true of examples through the book. The ATPG application of Alloy I have given is in the calculator style. I could have written predicates that would do a great job of checking the validity of the circuit under test, but I did not do so in the interests of keeping the model simple. See footnote 3.

9. Acknowledgements

Mark Utting's work in Model-Based Testing (MDT) as been a primary inspiration to me to study more advanced methods in testing even though my domain in testing targets hardware rather than software. The book (Utting & Legeard, 2007) on MDT has been generally inspirational to me. The paper (Aydal, Paige, Utting, & Woodcock, 2009) used Alloy as a model checker for test generation purposes and was particularly inspiring for getting this paper done.

This paper was written in Open Office v3.0.1 ("OpenOffice.org," n.d.). With the Zotero ("Zotero," n.d.) plug-in, I have experienced the most profound ease (and low cost) yet in writing this paper. Zotero has taken what used to be the most painful part of writing a paper, viz., the references, and made them a pleasure!

I found Jennifer Widom's web page (Widom, 2009) on writing a technical article of substantial help in my moment of need and hope she may find that I have largely complied.

10. References

Abramovici, M., Breuer, M. A., & Friedman, A. D. (1990). *Digital systems testing and testable design*. IEEE press New York.

Alloy (specification language) - Wikipedia. (n.d.). . Retrieved August 11, 2010, from http://en.wikipedia.org/wiki/Alloy_(specification_language)

Alloy FAQ. (n.d.). . Retrieved August 11, 2010, from http://alloy.mit.edu/faq.php

Automatic test pattern generation - Wikipedia. (n.d.). . Retrieved August 5, 2010, from http://en.wikipedia.org/wiki/Automatic_test_pattern_generation

Aydal, E. G., Paige, R. F., Utting, M., & Woodcock, J. (2009). Putting formal specifications under the magnifying glass: Model-based testing for validation. In *Proceedings of 2nd International Conference on Software Testing, Verification, and Validation, Denver, Colorado, USA* (pp. 131–140).

Biere, A., & Kunz, W. (2002). SAT and ATPG: Boolean engines for formal hardware verification. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design* (p. 785).

Bushnell, M. L., & Agrawal, V. D. (2000). *Essentials of electronic testing for digital, memory, and mixed-signal VLSI circuits*. Springer Netherlands.

Ibarra, O. H., & Sahni, S. K. (1975). Polynomially complete fault detection problems. *IEEE Transactions on Computers*, *100*(24), 242–249.

Jackson, D. (2006). *Software Abstractions*. MIT Press.

Kodkod. (n.d.). . Retrieved August 11, 2010, from http://alloy.mit.edu/kodkod/

Larrabee, T. (1992). Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *11*(1), 4–15. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.2529&rep=rep1&type=pdf

McCluskey, E. J. (1986). *Logic Design Principles: With Emphasis on Testable Semicustom Circuits*. Prentice Hall.

Mourad, S., & Zorian, Y. (2000). *Principles of testing electronic systems*. Wiley-Interscience.

OpenOffice.org. (n.d.). . Retrieved August 16, 2010, from http://www.openoffice.org/

Stephan, P., Brayton, R. K., & Sangiovanni-Vincentelli, A. L. (1996). Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *15*(9), 1167–1176. Retrieved from http://www.eecs.berkeley.edu/~brayton/papers/tegus.ps

Utting, M., & Legeard, B. (2007). *Practical model-based testing: a tools approach*. Morgan Kaufmann Pub.

Wang, L. T., Wu, C. W., & Wen, X. (2006). *VLSI test principles and architectures: design for testability*. Morgan Kaufmann Pub.

Widom, J. (2009, December 4). Tips for Writing Technical Papers. Retrieved August 16, 2010, from http://infolab.stanford.edu/~widom/paper-writing.html

Yang, K., Cheng, K. T., & Wang, L. C. (2004). TranGen: a SAT-based ATPG for path-oriented transition faults. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference* (pp. 92–97). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.1606&rep=rep1&type=pdf

Zotero. (n.d.). . Retrieved August 16, 2010, from http://www.zotero.org/

APPENDIX – Source Code Listing

The (unnumbered) source code file may be downloaded from http://www.samkirk.com.

```
1    // This is a very simple example of ATPG for a digital combinational circuit using Alloy
2    // CC-BY, v0.1 by Samuel A. Kirk, http://www.samkirk.com

3    open util/boolean

4    abstract sig Fault { }
5    one sig NF, SA0, SA1 extends Fault { }  // no fault, stuck at 0, stuck at 1

6    abstract sig Node { v: Bool }
7    one sig O1, I1, I2, X1 extends Node { }      // The pristine (unfaulted) version of the circuit
8    one sig O1', I1', I2', X1' extends Node { }  // The faulted version of the circuit.
9        // O = Output, I = Input, X = internal (non-observable) node

10   fun Connect [input: Bool, f: Fault] : Bool {
11   // Connect -  passes along the value from its input if there's no fault (NF).  Otherwise,
12   // it passes along True
13   //    if the fault injected is stuck at one (SA1), or False if stuck at zero (SA0)
14   // input is the value it receives
15   // the "returned" value is the value the Connector "outputs"
16       ( f = NF) => input else
17       ( f = SA1 ) => True else
18       ( f = SA0 ) => False else
19       none
20       }

21   pred Network [ f: Fault, o1, i1, i2, x1: Node ] {
22   // Network - predicate defining the structure of the circuit using Boolean functions and Connect.
23   // The inputs are given as parameters and are used but not "assigned" or otherwise defined.
24   // This predicate is invoked for both the faulted and the non-faulted circuit in a single run by
25   // the caller.

26       o1.v = And [ i1.v, x1.v ]
27       x1.v = Connect [ i2.v, f ]
28       }

29   pred Compare( ) {
30   // Is there a single set of inputs, I1 and I2, for the faulted and unfaulted circuit such that
31   // the output, O1, differs when there's no fault (NF)
32   // as compared to when there is a fault (SA1 or SA0) ?

33       // faulted inputs = non-faulted inputs.
34       I1.v = I1'.v
35       I2.v = I2'.v

36       // faulted outputs != non-faulted outputs
37       O1.v != O1'.v

38       Network[ NF, O1, I1, I2, X1 ]
39       Network[ SA1, O1', I1', I2', X1' ]
40       }

41   run Compare for 3
```