Sequential Automatic Test Pattern Generation by Constraint Programming

Sebastian Brand

CWI
P.O. Box 94079
NL-1090 GB Amsterdam
The Netherlands
Sebastian.Brand@cwi.nl
www.cwi.nl/~sbrand/

Abstract. The problem of automatic test pattern generation for sequential digital circuits is considered from the point of view of constraint programming. A classification of existing logics used for ATPG is described, and the ideas are consequently extended to the case of sequential ATPG. An implementation parametric in several logics is shown to be a useful prototype for constraint programming-based sequential ATPG.

1 Introduction

The problem of automatic test pattern generation (ATPG) for digital circuits is a hard problem in electric engineering. The production process for digital circuit is not error-free. The share of error-free chips can be as low as few percent. Therefore every produced chip needs to be subjected to a test: for a number of input patterns compare the expected against the actual output. ATPG is the problem of finding a small number of test patterns that identify a high number of possible faults.

For combinational circuits (circuits without loops), ATPG by constraint programming (CP) has been examined before [HSD92,Sim89]. We built on this work but apply here CP to sequential circuits, those circuits that have loops and states. We compare several models, and show that modern, tool-supported CP is well applicable to such a task. The implementation is tested with some examples from the literature and from benchmarks from the electrical engineering community.

2 Sequential ATPG

2.1 Digital Circuits

For ATPG, a digital circuit lends itself well to be modelled as a network of constraints, due to its inherent binary value structure. The constraints are the logical gates *not*, *and*, ... Stateless combinational circuits can be seen as computing an output vector of bits from an input vector of bits, in on step. A sequential circuit can be seen as originating in a combinational one where some (previous) output lines are connected to some (previous) input lines via a memory element (a flipflop). The memory element passes the value at its input to its output in the next clock cycle[1]. Thus, memory elements represent the state.

We consider here ATPG at the logical gate level. ATPG is also possible at the register transfer level, a higher abstraction.

Fault Model. The usual fault model is the single *stuck-at* fault. One supposes a line as permanently set to one value, 0 or 1, independent of its input. A testset covering all single stuck-at faults also covers many other faults, although not time-dependent ones. We assume further that the circuit clock is faultless.

Test Pattern. A faulty circuit is identified as such if after a sequence of input patterns (separated by a clocktick, which we will ignore from now on), the subsequently observed output differs from the output to be expected due to the structure of the circuit. This is the most widely used single observation time testing. There exist also faults and circuits for which multiple observation time testing – observing along several clockticks – is needed.

2.2 Test Generation

In order to compute test patterns through possibly several clock ticks, or steps forward in time, one needs to attach time information to values in the circuit. To this end, one models a sequential circuit by its unfolding into single timeframes: delete the memory elements and transform these lines into *pseudo-input* and *pseudo-output* lines (as opposed to *primary* output lines). Then a copy of the now combinational circuit can be added forwards (and backwards), such that the respective pseudo-input and -output lines are connected. *[picture of this needed]*

How many unfoldings into *timeframes* are needed depends on the fault at hand. The upper limit is exponential in the number of memory elements. There are circuits and faults for which this limit is reached (consider a bit counter and a fault in its highest output bit: the counter needs to count up to its maximum to set the potentially faulty bit to a defined value). It is common to set a limit on time spent or number of timeframes used, to avoid getting stuck in such computations.

Moreover, there are circuits containing faults that cannot show up for any input sequence. These faults are redundant in the true sense, there existence does not influence the circuit functionality. Consequently, an ATPG system can not compute a test sequence.

Therefore, ATPG systems often do not reach 100% fault coverage.

[1] sequential circuits are clocked, or can be modelled has being so

2.3 Sequential ATPG as a dynamic CSP

Sequential ATPG, unlike combinational, is thus in principle an optimisation problem. Practical systems faced with today's large circuits ($100,000$ gates and upwards), however, usually confine themselves to finding the first solution, with heavy reliance on heuristics. Even then computation times are in the order of magnitude of several hours. We do not deal with these specialised heuristics here, but concentrate on the model instead.

Controlling the unfolding. There are several options here (forward-only, backward-only, forward-reverse, ...). We choose an adapted iterative-deepening search with forward-reverse processing, i. e., setting a limit on the number of unfoldings that is increased as needed. Within one step, timeframes are added as needed and possible, with a preference of adding forward.

Fault Propagation and Justification. The first step in making a fault effect observable at the output is to excite it. For a stuck-at-0 fault this is done by forcing the line to 1 by an appropriate input sequence. The fault effect then exists internally, and must now be propagated to some output. To this end, the gates along a possible path of the fault effect must be controlled accordingly. Consider a stuck-at-0 fault at the input of an *or* gate. If the other input is a 0 then the fault effect will appear at the output as well.

This process of *fault effect propagation* leaves a number of *objectives*, such as the 0 at the *or*-input. Finding appropriate circuit inputs (and possibly pseudo-inputs) is the *justification* phase.

3 A Taxonomy of Logics

There are a number of ways to enhance the underlying binary logic $\{0, 1\}$, and to use these logics (collection of values and their behaviour with respect to the gates). We describe here several possibilities, some from the literature, some new.

3.1 Logic type options

The type of logic to be used depends on the type of variable that takes values in the logic. We differentiate between variables in an imperative language, and logic variables.

Domain Logic. Such a logic to be used for a logic variable contains precisely the base values that the variable can be instantiated to. The domain being a singleton coincides with the variable being instantiated. Domains of variables get only smaller. These are the types of logics one uses in a (finite domain) constraint solver with domain reduction functions.

Assignment Logic. In the imperative world variables are always instantiated and used with destructive assignment. An assignment logic equivalent to some domain logic can be constructed by defining a constant for every every subset of the domain logic. These constants are the values that are subsequently assigned to the variables. It is clear that the partial order corresponding to the respective subset order is needed to control consecutive assignments to one variable.

Historically, it is interesting to note the ATPG systems developed in electrical engineering started with assignment logics without being fully aware of the relationship to domain logics via the subset constants. Early algorithms for combinational circuits used only some subset constants, and when the value model was transferred to sequential ATPG, the remaining constants had to be added for completeness of search, due to repeated effects of faults in consecutive timeframes [Mut76].

One Layer/Two Layers. The necessity to consider at the same time the good and the faulty circuit, which are different but only slightly, leaves two possibilities. One can construct two constraint networks with two sets of variables (e. g., with a boolean domain) representing the good and the faulty circuit, such that in the faulty circuit the faulty line is replaced by a fixed 0 or 1. They are linked via their common inputs. Or one can use only one incarnation of the circuit as gate constraints to which a special fault site constraint is added. Then one needs a compound logic, the cartesian product of a base logic; such as $\{0/0,\ 0/1,\ 1/0,\ 1/1\}$ representing the combination good/faulty value in the circuit.

3.2 Value options

Several new symbols to carry extra line information were introduced to extend the binary logic $\{0, 1\}$.

Value x. This is used in assignment logics to denote the unknown assignment, or "'one of $0, 1$". It can be replaced by $0, 1$, or it can stay x in the pseudo-inputs at the oldest timeframe, where it expresses that *any* previous state is acceptable to start the test sequence at hand. Domain logics have an (a number of) implicit x value(s): any non-singleton domain.

This value is used for instance in [Mut76], which shows that this 9-valued logic is an assignment logic, and should not be used as a domain logic.

Value u A refinement on the concept of irrelevant previous state is using u (uncontrollable) for this purpose while using x or a non-singleton domain for currently unspecified values. A u can also be *propagated into* the circuit, which allows earlier detection whether fault propagation or justification is possible with the currently last timeframe, and what input values are good choices. A u behaves similar to $x = 0/1$, so for instance: $or(u, 0, u)$, $or(u, 1, 1)$.

The u value is important for domain logics in sequential ATPG for precisely the purpose of marking uncontrollable pseudo-inputs of the oldest timeframe.

Values *d*, *dnot*. These abbreviate 1/0 and 0/1 resp., in a one-layer logic. These values originate at the fault site and need to be propagated to some primary output, where they indicate a fault effect. The definition of gates needs then to be correspondingly, e. g., $or(0, d, d)$, $or(1, d, 1)$.

Since as shown d can be masked away as well, it is useful to use targeted variable and value selection in the fault effect propagation phase in order to drive a d-path to some output.

Value *e*. The idea stems from [HSD92] and is the following: In fully instantiated circuit, the fault effect d will trace a path from the fault site to a primary output. If some line participating in this path occurs in more than one input to a gate, i. e., if it branches, then all other branches contain (copies of) the d as well. These copies may or may not reach a primary output, it suffices that one does. The value e represents such expendable fault effects. Particular *fanout* gates are introduced for this reason, such that they convey precisely one d only (thus they constitute a choice point). The binary case is $\mathit{fanout}(I, O_1, O_2)$:

$$\mathit{fanout}(d, d, e), \mathit{fanout}(d, e, d), \mathit{fanout}(1, 1, 1), \mathit{fanout}(0, 0, 0)$$

All gate definitions should then equally be such that d does not disappear. Thus, for instance, $or(0, e, e)$, $or(0, d, d)$, $or(1, e, 1)$, but not $or(1, d, 1)$. Then d and e provide precisely one d-path from the fault site to some output if one exists.

4 A CP implementation

We implemented a sequential ATPG system as a constraint logic program in ECL^{i}PS^e and `CHR`, with the help of the RGA(W) generator of [AM01]. The following logics were considered:

***3u*:** 9-valued $\{0, 1, u\}^2$, a two-layered domain logic

9du*:** 9-valued {0, 1, d, *dnot*, u/u, $u/0$, $u/1$, $0/u$, $1/u$, u/u}, a one-layered domain logic (note that 1 is really 1/1, etc.) Observe the correspondence to ***3u, the cartesian product of which is ***9du***.

For ***9du*** and ***3u***, the fault effect propagation is directed by following the structure of the circuit from the fault site to an output with variables that are labelled with d (or *dnot*).

***11deu*:** 11-valued $\{0, 1, d, dnot, e, enot, u/u, \ldots\}$ one-layered domain logic

Justification for all three logics is done by labelling with the smallest domain first heuristic for variable selection. (Again, we dispense with other strong heuristics developed in the electrical engineering community, for instance [FS83] or controllability/observability measures, at this time, because we are interested more in the constraint propagation.)

4.1 Generating Propagation Rules

If one wants to use a logical gate as a constraint, one needs to derive some constraint propagation mechanism for it. For this case we employ the results of [AM01]. From the description of the gate, two types of rules can be computed: 1) membership rules that enforce arc-consistency, and 2) equality rules that enforce rule-consistency, a weaker local consistency than arc-consistency. All rules are domain reduction rules with a condition: instances in the case of equality rules, subsets in the case of membership rules.

The gate descriptions themselves are all successively, and building on each other, generated with the simple boolean *and* and *not* at the root. A description for the behaviour of u is added, and a filtering of the generated gate tuples with respect to the d value (who must not be created nor disappear from a normal gate). From correspondences such as $or(X, Y) = not\ and(not\ X, not\ Y)$ etc., one can compute all other gate definitions. Also, for equality rules, definitions for gates with more than two inputs can be computed. For membership rules this is not necessary, as it is easy to prove that arc-consistency is established as well by stating the decomposition directly as a set of constraints.

benchmark

4.2 A Configurable ATPG System

The resulting constraint program is parametric in the logic, the circuit, and the faults it can take, with only a small part of the code depending on the logics. Upon calling, it returns test patterns for each specified fault, or "aborted", which occurs once the unfolding maximum or the time maximum has been reached. These are necessary to capture hard faults.

4.3 Additional Constraints

For ***3u*** and ***9du*** it needs to be ensured that the fault effect reaches the output. This could be done by a test in the form of a constraint, since the labelling heuristic in the fault propagation phase takes care of that already. Here, however, we use a constraint that, once only one output is possible to show the fault effect, enforces the fault effect there. Constraint propagation can then lead to further assignments that might specify the fault effect path more.

Another example for a useful and easily addable constraint is state repetition. In the sequence of states a test input pattern creates, there should be no repetitions. If they are then the enclosed subsequence can be cut out, and the remaining test pattern will still make the fault observable. So a constraint that forces all states to be different is useful. The implementation contains such a constraint equivalent to the *not-equal* relation for every pair of states. Note that one has an *alldifferent* constraint here, for whose simple form strong consistency algorithms have been studied, which might be applicable here, too.

4.4 Benchmarks

We tested the implementation against some example circuits, including some circuit descriptions from the ISCAS'85 benchmark set. The result for two small circuits for each logic, and both types of rules (m: membership, e: equality), with the number of generated rules for *and* in that case, is (seconds):

	3u, e	***3u***, m	***9du***, e	***9du***, m	***11deu***,e	***11deu***, m
muth	0.23	0.48	0.50	45.0	0.65	390
s27	0.77	0.54	1.20	68.4	9.80	420
# rules	16	18	134	1294	153	4656

The fault coverage is in both cases 100% (all faults can be discovered). Both circuits contain about 10 gates, and the length of the test sequences is between one and three.

According to these results, performance is better for the two-layered logic with twice the variables but smaller domains, and less rules. Note in particular that the program instances for ***3u*** and ***9du*** correspond very closely to each other (variable domains, control of labelling). Among the higher-valued logics, there is a strong advantage of equality rules, which enforce a weaker consistency but need more labelling.

A larger circuit, s641, with about 400 gates (including ones with more than two inputs) finished computation after 23 hours with equality rules for ***3u***. The result were test patterns for 101 faults, while the remaining 226 faults were aborted; the fault coverage is thus 28%. For this experiment, the timeout and unfolding limits were chosen very low, in order to get any result for the circuit. Several other larger circuits had comparable performance. These experiments demonstrate first of all that the principle of the approach works for larger benchmark circuits.

Further, the experiment times seem to show some correlation between time needed and number of rules. This makes sense once one realises how the rules are scheduled in CHR: in general every rule is tried once on every instance of a gate constraint. This is not necessary since elements can be removed from domains just once, rules do double work. We plan to look closely at this issue, possibly replacing `CHR`. Another rule-related issue worth examining is that of finer control of propagation, such as directing it forwards or backwards only, at a time.

5 Conclusions

We showed that sequential ATPG is feasible by constraint programming. Several models are possible, including a conceptually elegant one such as ***11deu***. Constraint propagation procedures can be generated by general existing methods, which also allow the choice of local consistency, with substantial effects. The benchmarks indicate that simple logics are faster, although the type of rules and their scheduling seem to play an important role, which requires more research.

6 Acknowledgement

Eric Monfroy provided me with valuable and extensive comments on the problem. To Krzysztof Apt, who suggested the topic, I am especially thankful for continued inspiration.

References

[AM01] Krzysztof R. Apt and Eric Monfroy. Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming*, 2001.

[FS83] Hideo Fujiwara and Takeshi Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, 32(12):1137–1144, 1983. multiple backtrace.

[HSD92] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58(1-3):113–159, December 1992.

[Mut76] Peter Muth. A nine-valued circuit model for test generation. *IEEE Transactions on Computers*, 25(6):630–636, 1976.

[Sim89] Helmut Simonis. Test generation using the constraint logic programming language CHIP. In Maurizio Levi, Giorgio; Martelli, editor, *Proceedings of the 6th International Conference on Logic Programming (ICLP '89)*, pages 101–112, Lisbon, Portugal, June 1989. MIT Press.