PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 36, NOS. 3-4, PP. 319-334 (1992)

AUTOMATIC TEST GENERATION BASED ON CONSTRAINTS[1]

K. TILLY*[2] , GY. ROMÁN**, L. SURJÁN**

*Department of Applied Physics
Delft University of Technology, The Netherlands
**Department of Measurement and Instrument Engineering
Technical University of Budapest,
H-1521 Budapest, Hungary

Received: May 13, 1993

Abstract

It seems to be a very hard task to enhance the properties of widespreadly used automatic test pattern generation algorithms. Experiences show that achievements are sometimes not worth the effort. In the authors' opinion this fact stems from the basically 'algorithm oriented' nature of research made in the past. A new experimental framework is presented for the problem, considering network representation and search control algorithms as equally important parts. The network is represented by object- oriented data-flow networks, the search control algorithm is based on constraint satisfaction, and a special kind of dependency directed backtracking which we call constraint slackening. Similar methods were proved to be very useful in automatic system diagnosis by DAVIS (1985) and others, although have not been introduced to testing yet. This paper summarises the basic notions of constraint satisfaction, the potential advantages of using it for building test generation systems, and shows implementational details of a test generation system, based on constraints. Experiences of the run-time tests show that constraint-based test generation can be highly efficient.

Keywords: automatic test pattern generation, constraint, data-flow networks, object-oriented programming.

Introduction

There are several known algorithms for automatic test pattern generation in digital circuits as described by ROTH (1966), KIRKLAND (1988), FUJIWARA (1985) and others, which are rather efficient and widespreadly used. Although when we try to enhance their properties (e.g. speed or coverage rate), or extend them to other domains (like from gate level to functional testing or from combinational to sequential circuits) we have to face extreme difficulties, where extensive efforts must be made for acceptable gains. The authors are convinced that this is concerning not only

[1]This project was sponsored by the grant OTKA 5-771.

[2]On the leave from the Department of Measurement and Instrument Engineering, Technical University of Budapest

the difficulties of the problem itself (which naturally plays a distinct role here), but also with the representational and programming tools chosen for building test generation systems. In this article we try to show that — beside other relevant advantages mentioned later — the efficiency of traditional test generation algorithms can be significantly increased if we use a carefully designed and appropriately implemented software environment.

To characterise the properties of such an environment, we consider as a starting point the two basic problems in a test generation system:

- The first one is to find a representation method to describe the network for which we want to generate test patterns.
- The second problem is to control the test generation process. To achieve this we need algorithms to efficiently perform extensive graph search using the previously described network. It is also important that the control algorithms must avoid most unnecessary search, since automatic test pattern generation is NP-complete.

Methods used until now have paid most attention to the construction of the control algorithms, while considering the network representation problem as a solved one.

A basic experience of the software engineering community shows that to efficiently solve difficult problems (as automatic test pattern generation obviously is) is practically only possible by using software tools dedicated to the needs of the current task. So we viewed the problem from this perspective and we tried to establish an environment dedicated to automatic test generation.

It is obvious that digital networks are built of just a few types of blocks, although a given network can contain as many instances of these elements, as needed. This property can be very easily handled in an object-oriented environment, where element types (like NAND gates) can be modelled by distinct object classes, while individual gates can easily be produced by generating instances of these classes.

This way we can represent nodes of the network as object instances, but what to do with the connections (wires) between the nodes? Data-flow networks can help BIEGL (1988). As it is known, data-flow networks contain elementary processing units in their nodes and a data item is ordered to any of their arcs. Any node has inputs and outputs. If a node has enough valid data on its inputs, the node can be started, it performs a specific transformation on the input data and sets the values of its outputs. This also means that some of the nodes connected to the output of this node may be started. Such a network offers the possibility of concurrent run-time scheduling and shows a very strong analogy with electric circuits.

Data flow networks are efficient tools for simulation, though test generation requires more than that, since in this latter case the direction of in-

formation flow on the arcs (wires) of the network is not given (the data-flow graph is not directed). This requirement has some important consequences. We can only refer to inputs and outputs of a node as bi-directional pins — although we characterise the original direction of pins as input or output. A more difficult problem is that nodes (or more precisely the object classes of nodes) must be constructed in a special way, so as to make them able to accept input values on any of their pins and supply output values to any other pins. These facilities can be handled by means of constraint propagation techniques, which have gained an increasing importance in AI research in the last few years and offer just the required properties.

In the next sections we summarise the basic notions of object-oriented data-flow and constraint propagation systems and point out their advantages for building automatic test pattern generators. In the last part we describe and evaluate our first implemented constraint-based test pattern generator for combinational logic circuits.

Basic Concepts

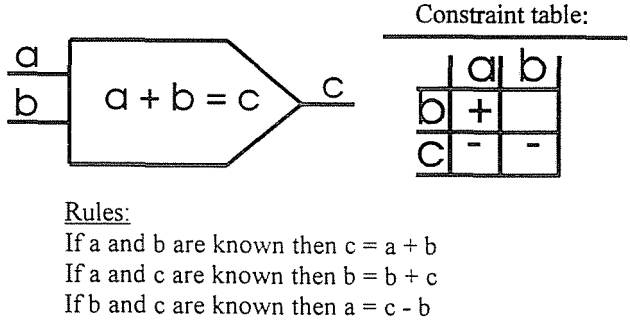


Fig. 1. Representation of an elementary constraint

The most straightforward way to define elementary constraints is by simple (algebraic) equalities (or inequalities) containing an operator and a set of constants and variables as shown in *Fig. 1.* Any (sometimes more than one) of these variables can be unknown, while others hold given values. Compound constraints or sets of constraints can be represented by constraint networks, whose nodes are constraints and the arcs hold variable values. *Figs 2* and *3* show a classical example by STEELE (1980).

The constraint satisfaction problem itself is to find a set of values of unknown variables, which (together with the known ones) satisfy any constraint in the network. There are several real-world problems (like qual-

itative modelling of physical systems, automatic diagnosis DAVIS (1985), equation solving, dynamic resource allocation problems or automatic design) which can be characterised by means of constraints.

The only problem with the simple graph representation stated in the previous section is that a given variable can take place in more than two constraints, so a single arc is not sufficient to represent a constraint variable.

A good solution to this problem is to use hyper graphs as stated by MONTANARI (1991), where a hyper arc can connect as many nodes as required.

In practice hyper graphs can be represented by using two kinds of nodes: actor nodes (which hold constraints) and data nodes (which hold variable values) STEELE (1980), BIEGL (1988). In this model of constraint networks no nodes with the same type can be adjacent, though to a data node any number of actor nodes can be connected and vice versa to an actor node any number of data nodes can be attached (*Fig. 2*).

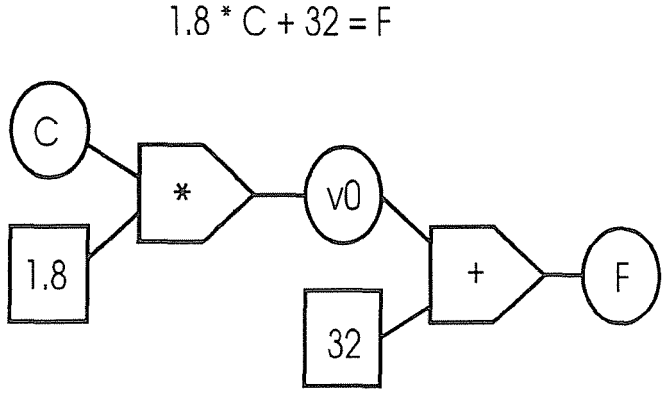


Fig. 2. Representation of constraint networks

The stated representation method is in theory rather simple, although traditional programming tools do not support it, since for example the assignment operation which is closely related to constraints is unidirectional, so the left side has no affect on the right side, while the right side has affect on the left side only at the very moment of the evaluation of the given statement. On the other hand, the implementation of the data-flow graph itself is not supported at all.

It means that at our current stage of available software tools some software development is required. The main task here is to implement the elements of the data-flow network and to create a run-time system to efficiently operate it. This run-time system would also incorporate the test generation control algorithms.

Elementary constraints are traditionally implemented by means of rules STEELE (1980), although we prefer constraint tables, because they are related to truth tables, this way it is easier to check consistency and table items can contain sophisticated procedures and compound data items as well which make this representation for our purposes more flexible (*Fig. 1*).

The simplest and basically very fast network control technique is local propagation, where any actor has only local information (i.e. the values of data nodes directly connected to it), and established to this information and its internal constraints the actor fills in the missing values around it. *Fig. 3* shows a classic example of STEELE (1980) to demonstrate this method. Arrows indicate the direction of information flow, values known at the beginning are underlined.

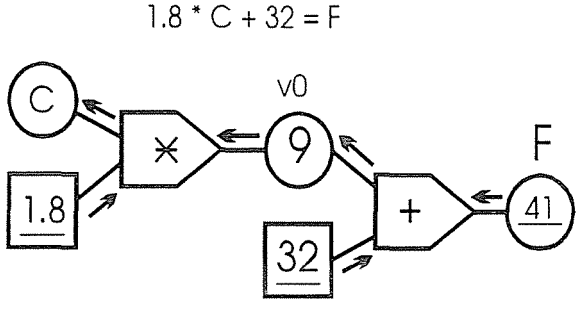


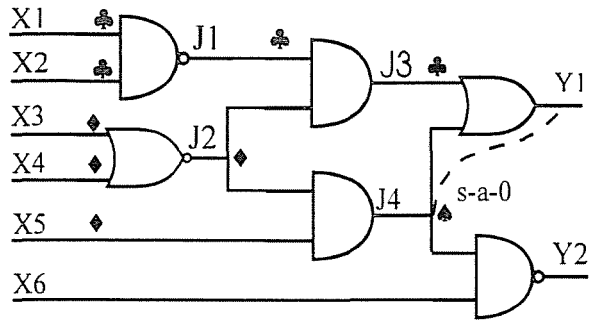
Fig. 3. An example of equation solving through local constraint propagation

Although when using this technique it is not trivial to handle cycles in the network, in our case fortunately it does not mean a very strict limitation (unlike in the case of solving equation sets with just one solution), since in test generation we have 'under determined' value sets.

Now let us consider an example taken from the field of combinational circuits to make clear the specific use of constraints and the basic steps of an ATPG algorithm (*Fig. 4*).

Although there is a very large range of practical differences, essentially any ATPG algorithm performs three basic steps:

- *Fault propagation*: A path must be found from the place of the actual fault to a primary output through connected gates to propagate the fault and make it observable. A fault is propagated through a functional element, if at one of its outputs there is a different value for the fault-free and for the faulty network.
- *Fault sensitization*: A primary input combination must be found which forces the opposite level of the given stuck-at (e.g. 1 must be forced for s-a-0) at the point of the given fault.



♠ site of fault ♦ fault sensitising junction
... D-path ♣ Junction to justify
D-drive: J4 — D → Y1 — D
Fault sensitisation: J4 — D → J2 — X5 — 1
J2 — 1 → X3 — X4 — 0
Justification: Y1 — D → J3 — 0 →
→ J1 — 0 → X1 — X2 — 1

The generated test for the fault:

X1	X2	X3	X4	X5	X6
1	1	0	0	1	X

Fig. 4. A simple ATPG problem

– *Justification*: During the previous two steps a set of internal junctions also get values. These must be recursively 'justified' by finding (tracing backwards to the primary inputs) a consistent set of assignments for all the junctions which affect these values.

The Fault Model

We apply the simple single-stuck-at-fault model, where only one fault is assumed to occur in the network at a time, and the fault can be described by fixing the value of the given junction at 0 (s-a-0 fault) or 1 (s-a-1 fault). We use the same notations for the values in the circuit as Roth in his D algorithm: LOW (logical low level, 0), HIGH (logical high level, 1), X (don't care), U (unknown value), D (stuck at zero fault), -D (stuck at one fault). Although this model is a strong simplification of real life situations, it was shown that the significant majority of multiple faults and even more complicated faults (e.g. bridging faults) can be covered by single-stuck-at

tests MICZO (1986), and it has the advantage that it can be handled very simply and efficiently.

Network Representation: a Closer Look

So far we have dealt with network representation and control algorithms in general. Now we choose a more distinct domain and determine the elements of network representation from this scope.

Though for the sake of simplicity we have chosen combinatorial circuits and gate level test generation to gather experiences about the applied methods, at this point we have to make some important remarks.

The concepts stated above make no theoretical limitations on the complexity of individual constraints, so it is possible to extend them to functional level testing. Since algorithmic parts are distributed among the run-time system and the individual constraint nodes, complex tasks can be efficiently handled, because the solvable problem is decomposed in a natural manner into subproblems.

Data-flow graphs also offer the possibility of handling multi level system models thus offering new possibilities of hierarchical model decomposition.

The extension of these methods to sequential circuits also seems possible, in this case although some additional solutions are required (e.g. for the determination of those states which can be reached from the starting state supposing that a given fault is present).

Last but not least in constraint based test generation every gate (or functional element) contains individual algorithmic parts (methods) that is why they can be separately run on multiprocessor architectures, so this method supports parallel implementations.

Object Classes

We represent network components by three basic object classes: gates, nodes and ports.

Gates are the primary processing units of the network. There is a superclass called gate with several subclasses representing the basic gate types (e.g. AND, NAND, OR, NOR, INVERT, XOR). Gates represent the constraint nodes in the network, so they contain sufficient data structures and algorithms for performing any tasks dedicated to a gate.

The gate class instance variable structure consists of the following elements:

- *The output node and the list of input nodes.* The gates can set the values of the connected nodes using this information. Although the output node is also handled bidirectionally, it must be differentiated from the input nodes because of its different role.
- *Number of input pins.* This way gates with the same type but different number of input pins can be handled with the same class but different parametrisation.
- *Level of the gate from the output.* This information is used by the D path search procedure of the run-time system for determining the shortest path from a given fault to a primary output point.
- *List of supporting nodes.* This is a list where those nodes are stored, whose values were known when the gate was started.
- *List of consequent nodes.* This is a list of those nodes, whose values were set by the given gate. These two lists are used for recording data dependencies in the network.
- *Contradiction list of supporting nodes* and contradiction list of consequent nodes. These two lists contain the same information as the previous two, although only built and used during contradiction resolution.
- *The state of the gate* is used by the run-time system to support decisions which gate and how to activate.
- *The significant flag.* Only those gates will be activated, whose significant flag is set. It is used for minimising the number of gates involved in the test generation process.
- *Weight*: It is used when a gate must be selected from the hesitate agenda.

Nodes are the analogies of nodes in a digital circuit. They connect several inputs with an output by 'wires' and hold specific values. Any pin of a gate (but naturally only one output) can be connected to any node.

The instance variable structure of the node object class consists of the following elements:

- *The value and state of the node.* A node can have two kinds of state and two value slots according to the different states: one during normal execution and another during contradiction resolution. (These two operating modes will be explained later when introducing the operation of the run-time system.)
- *The list of connected gates.* Using this list the node is able to determine for the run-time system which gates to run. It is ordered by the increasing level of the connected gates (the gate nearest to an output comes first).
- *The identifier of the driver gate* which has set the value of that node. The role of this variable is again to record data dependencies.

Ports are the primary input/output points of the network. They are similar to nodes, although they have some extra functions. They can be driven from outside and must signal that some values have reached them from inside, so they make the run-time system able to decide when the test generation process ends.

The port instance variable structure consists of the same elements described by nodes, except that it also holds the direction of the port (input or output).

To make the operation of the run-time system as simple as possible, different object classes use the same kinds of methods (although a given method can have different function in a different class).

We have to mention that some of the methods used here are similar to or identical with methods already known in the field of testing. In the description of these methods we gave the appropriate terms in *Italic*.
The following methods are used:

- *Make*: object instance generation;
- *Init*: initialises the instance variable structure;
- *Level Process*: activates the procedure for determining the level of the given object: The level of output ports is 1. The level of a node connected to an output port is also 1. The level of a gate is the level of its output node plus 1. The level of any internal node is the minimal level of the gates whose inputs it is connected to.
- *D Process*: D path propagation in the network from the place of the fault to a primary output (port), this is known as *D-drive* in the literature. Values of nodes set by this method cannot be altered in a possible contradiction resolution process;
- *Mark Path*: If a gate gets a Mark Path message, it sets its own significant flag, and if its significant flag was previously not set, it sends a Mark Path message to all its input nodes. If a node gets a Mark Path message, it forwards it to the gate whose output is connected to this node. The Mark Path procedure is initiated by the run-time system during D path selection, and ends either on gates that are significant, or on input ports.
- *Process*: the normal operation of the object based upon its internal constraints. The internal operation of a gate is described by constraint tables using nine valued logic, although nodes can only get values which are elements of the stated fault model. This method is activated when a complete D path is found and it is able to propagate values to both forward and backward directions. This action includes the *implication* and *line justification* steps known from the D algorithm.

- *Set Weight*: This message is sent by the run-time system, when the weights in the hesitate agenda must be refreshed. The weight is computed on the basis of known and unknown values around the gate.

The following methods are only used during contradiction resolution:

- *Start Resolution:* Meaningful only by gates. When accepting this message, any gate determines and sets the first consistent value set for the unknown nodes around it.
- *Continue Resolution:* Meaningful only by gates. When a gate gets this message, it determines the next possible consistent value set. These two methods can also signal failure to the run time system, which means that no more consistent value sets exist. This time the backtracking procedure is initiated.
- *Retract:* If a gate gets a Retract message, it sends a Retract message to all the nodes, whose values were set by that gate. If a node gets a Retract message, sets its own value to Unknown, sends a Retract message to all the gates, which it supports (i.e. to all gates connected to it, except the driver gate), and deletes its own data dependencies (i.e. deletes itself from the support lists of supported gates and deletes its own driver gate registration).

Basic Network Operation

The run-time system has the task to schedule the network (i.e. to select gates to activate) and to switch between the different operation modes characterised later. The run-time system uses the following data structures for network scheduling: the normal queue, the hesitate agenda and a stack for optimal D path selection called the D stack. The execution of the run-time system supported by the nodes and gates is as follows:

N1 Set the levels of gates using the Level Process method. This step is only performed once for a given network, since in our fault model the structure of the network will not be changed by any fault.

N2 Initialise all nodes in the network in the following way: set every node to U (unknown) except the fault node, which is set to D or −D. (In the following we neglect the systematic generation of faults, although it can be very easily solved e.g. by using a linked list of nodes.) Mark the fault node 'fault'. Set the backtrack stack and backtrack list to NIL and the engagement level to 0 (the purpose of it see by the contradiction resolution algorithm).

N3 Generate a D path from the fault node to the nearest output port in the following manner.

N3.1 Set actual node to fault node, set the D stack to NIL, set the operating mode to D path selection (*D drive mode*).

N3.2 Get the list of neighbouring gates of actual node and push it to the D stack (One element on the D stack is the same as the *D frontier* in the D algorithm). Put the first element of the neighbours' list into the normal queue. Activate this gate for D Process (*D drive*). This gate will set the value of its output node.

N3.3 If an output port is reached, continue with step N3.4, else set actual node to the output node and repeat step N3.2.

Steps N3.4, N3.5 and N3.6 are for resolving the problem of multiple reconvergent D paths.

N3.4 Trace the D stack top down and activate all elements of the D path (these are the first elements of the lists in the D stack) with Mark Path method.

N3.5 Trace the D stack bottom up and queue all significant gates that are neighbours of the output node of a significant D path gate.

N3.6 While the queue is not empty, remove the first element of it, put it into the normal queue and start it with D Process. (It implies that nodes whose values are set will automatically queue significant gates attached to them). If the queue is empty, set the operating mode to normal operation and continue with N4.

N4 Activate the first gate of the normal queue, or if it is empty, start the best gate of the hesitate agenda with Process method. Running the elements of the normal queue means implication and/or line justification, but no decision is made. If there are more alternatives, the actual element is transferred to the hesitate agenda.

We think that the handling of the hesitate agenda needs a bit further explanation. Agendas are used for supporting best-first search in several successful AI applications. An agenda consists of tasks to be started. Any task has a list of justifications (i.e. reasons why we want that task to start) and a weight, which can be computed based upon the current list of justifications. The best task is always the task with the best weight and this one is started.

In our case the tasks are gates, the justifications are the list of supporting nodes and the weight is computed by the Set Weight method of gates. An additional problem we have not mentioned yet is the maintenance of the weights in the agenda. It would be too time consuming to recalculate weights any time when a new gate is entered to the hesitate agenda. It is only necessary, when the normal queue is empty and a hesitating gate must be started.

When a gate is started, it can produce three kinds of results:

1. The gate generates values for all the nodes around it having unknown values and registers itself as the driver gate in the nodes' appropriate slot. If a node gets a value from an output (i.e. in forward propagation), it puts all its neighbouring gates to the normal queue and registers itself on the list of supporting nodes of all of them. If a node gets a value from an input (i.e. in backward propagation) it does the same, but only for significant gates. This way the consideration of those gates, which are unnecessary for testing a given fault can be avoided.

2. The gate 'hesitates' (it is only allowed, if the gate was taken from the normal queue). This means that it replaces itself to the hesitate agenda. It can happen in cases when too few nodes have known values around the gate, so because of the low reliability of the new values the gate could generate, it rather postpones the decision hoping that the number of known nodes around it will increase. We note that a hesitating gate can become normal again if (before being activated from the hesitate agenda) a neighbouring node gets a value from another gate and this node puts the hesitating gate to the normal queue.

3. The gate detects a contradiction. It can happen when the nodes around the gate having known values do not match the constraints of the given gate. (E.g. there is LOW at one of the inputs of an AND gate and the output is HIGH.) In this case the run-time system starts the contradiction resolution session as described from C1. If the contradiction resolution fails, systematically choose a new D path according to step N6.

N5 If the normal queue and hesitate agenda are both empty, the test generation process successfully ended. Else continue with step N4.

N6 Use the registered data dependencies to neglect those gates of the D path, which have no effect on the contradiction. It is done as follows:

N6.1 Recursively trace backward starting from the contradiction gate using the support lists of gates and driver gates of nodes. The trace ends when all available gates in the D path using the registered dependencies are reached and marked. Based upon a heuristics we consider guilty that marked gate which lays nearest to the top of the D stack (i.e. nearest to an output). This may be sometimes a bit pessimistic, although it ensures that paths which surely will not terminate the contradiction are neglected, though no paths are dropped which can be important.

N6.2 Delete all elements of the D stack above the guilty gate, get the next gate of the list at the current stack top and mark the output node of it as actual (see step N3.1.), initialise the network (except the elements of the D path currently registered in the D stack), and continue with step N3.2.

It can happen that the list containing the guilty gate contains only one element. In this case while the D stack is not empty and the top of the D stack consists of just one element, pop it. If a list with more than one element appears on the top, get the next gate of it and use it to generate a new D path. If the D stack gets empty, no test was found and the process ends.

Contradiction Resolution

Contradictions must be handled very carefully, because they can totally destroy the overall efficiency of the test generation process. It is obvious that if there can be cycles (reconvergent fanouts) in the network, no simple method is known to assure global consistency of values.

That is why our system tries to avoid contradictions whenever possible.

This contradiction avoidance stems from two properties. The internal implementation of gates — since being distributed among different object classes — can be sophisticated enough to assure high quality generation of different value sets. The second property is the use of the hesitate agenda, which makes it possible to significantly decrease the number of contradictions. Experimental measurements show that for certain Fujiwara networks our system finds tests without any contradictions for more than 80% of the faults, but this ratio is by these networks is generally not lower than 45%.

The contradiction resolution is based upon the structure of the network, i.e. the dependencies naturally defined by the connections between individual elements.

During contradiction resolution no such nodes get values, whose value was previously unknown. Similarly no such gates are activated, which have not been started yet. The algorithm is started from the primary contradiction (i.e. the gate which signals the contradiction) and a special kind of dependency directed backtracking, the so-called constraint slackening is performed when a new, secondary contradiction arises. The backtracking is based on data dependencies registered during contradiction resolution in the same way as given by the normal operation, although in this case dependencies are stored in the contradiction slots. Gates are able to autonomously and systematically generate consistent value sets. Instead of the normal queue and the hesitate agenda a distinct contradiction queue, a backtrack list for storing retracted gates and a backtrack stack containing backtrack lists are used.

Since the size of this article is limited we will give a more detailed description of the contradiction resolution procedure in another article.

Correctness

We have to discuss the correctness of this algorithm. Step N3 assures that the fault is spread to an output. The problem of reconvergent fanouts is solved in steps N3.4–6. If the D path is blocked somewhere by an inadequate D & – D input combination, this fact is signalled by the gate as a contradiction and can be handled accordingly. When a gate is started, it sets all the nodes around it which have unknown values. It will result in the queuing of other relevant gates. This means that if all the queues are empty and no contradiction occurred, then no needed value is missing and all values are consistent that is a test is found.

Furthermore, it is worth telling some words about the correctness of the applied contradiction resolution algorithm. It is essentially an exhaustive breadth-first search algorithm, which is based upon structural data dependencies and considers the most relevant values first and neglects parts which have no effect on the resolution. It is also important to mention that — since we keep the assumption that a gate must set every unknown values around it — no circular data dependencies exist, which make value retraction much simpler.

The convergence of the contradiction resolution algorithm (i.e. that a solved primary contradiction will never occur as a primary contradiction again) is assured by the following.

After resolving a primary contradiction no values will remain retracted compared to the state when the resolution was started, but new values will be added by the primary contradiction gate (it can be done, since this was the purpose of the resolution). That means that the number of known values through resolving primary contradictions will monotonically increase.

The case of resolving a secondary contradiction is more complicated, because this time values can also be retracted. That is why, if a secondary contradiction is resolved, the conditions under which it was done are registered in the backtrack stack and nodes belonging to a given secondary contradiction are marked with the appropriate engagement level. This assures that if another secondary contradiction occurs, it cannot result in the retraction of the values set in previous secondary contradiction resolution sessions. Such values can only be altered under the same conditions (i.e. by the same gate, which once initiated the secondary contradiction). This solution — although may seem a bit complicated — leads to an automatic problem decomposition, where the more secondary contradictions occur, the resolution of them will be primarily based upon more and more limited value sets.

When a secondary contradiction has been resolved, results at the same gate can never cause a secondary contradiction again. This gate can only be involved in the backtracking procedure of resolving other secondary contradictions, but itself will not signal contradictions, since in this case a contradiction is merely an inappropriate value set which the gate will not accept. So the only thing which can happen later on is that the gate gets exhausted and a new backtracking step is initiated.

In an average case the contradiction resolution algorithm assures acceptable results, although in the worst case it runs with exponential time. Our experiences show that it is in most cases more efficient to use a simpler — although nonexhaustive — algorithm, which tries out all possible consistent combinations around the primary contradiction and when a secondary contradiction occurs, every value in the previous resolution step is set again to unknown and the procedure is started again with the next value set of the primary contradiction gate. If the primary contradiction gate is exhausted, a new D path is selected. This method in practical cases offers rather good results. For faults untestable with this simple algorithm, the described constraint slackening procedure can be used.

Conclusions

Constraint-based test generation is a new method which offers some major advantages. It can be used for gate level as for functional testing and it can also be applied for building multi-level, hierarchical test generation and diagnostic systems. If we represent test generation problems with the described methods, it becomes also possible to easily and naturally distribute the task on multi-processor architectures.

Although we have proposed algorithms for the run-time system (i.e. for controlling the test generation process), and we think these algorithms are well fitted to the applied network representation methods, we would like to point out that the normal operation algorithm has several common points with known test generation methods, so in theory any well known test generation algorithms could be adapted instead of it. On the contrary the stated contradiction resolution and backtracking method seems to us more unusual.

The test generation environment presented in this article is the result of our first experiments for checking some of our ideas for being viable. That means that on certain points the applied solutions are far from being optimal. The most significant point of these is the internal representation of gates, which currently contains too many heuristics, so we must find methods to make such descriptions more formalised and uniform.

The stated contradiction resolution algorithm can also be further refined by using algebraic graph rewriting methods to reduce the sizes of cycles in the network. The reason why we did not implement it yet is the increased complexity of such a system.

In spite of these drawbacks our first prototype has proved to be surprisingly efficient. For example for the Fujiwara c432 network the average time of generating a given test vector (i.e. the total time of test generation for all faults divided by the number of faults) is 0.6 seconds on a 12 MHz IBM/PC–AT 286 by a coverage rate of 98%.

Based upon current experiences we consider constraint-based test generation to be viable and we are working on the enhancements of this technique in multiple directions.

First of all we are searching for solutions to resolve the weaknesses of the combinatorial test generator, and we also intend to investigate the possibilities for using it for sequential circuits, multi-level functional testing and multiprocessor parallel implementations.

References

BIEGL, C. (1988): Design and Implementation of an Execution Environment for Knowledge Based Systems. Ph.D. Thesis, Electrical Engineering, Vanderbilt University, Nashville, TN.

DAVIS, D. (1985): Diagnostic Reasoning Based on Structure and Behavior. AI, Vol 32. No. 1–3, pp. 347–410.

FUJIWARA, H. (1985): FAN: A Fanout-oriented Test Pattern Generation Algorithm. IEEE Proc. of ISCAS '85. pp. 671–674.

KIRKLAND, T. (1988): Algorithms for Automatic Test Pattern Generation. *IEEE Design & Test*, Vol 5, No. 3, pp. 43–55.

LELER, W. (1988): Constraint Programming Languages. Addison Wesley Publishing Co.

MICZO, A. (1986): Digital Testing and Logic Simulation. Harper & Row, NY.

MONTANARI, U. – ROSSI. F. (1991): Constraint Relaxation May Be Perfect. AI, Vol. 48. pp. 143–170.

ROTH, J. P. (1966): Diagnosis of Automata Failures: A Calculus and a Method. *IBM J. Research and Development*, Vol. 10, pp. 278–291.

STEELE, J. Jr. (1980): A Computer Programming Language Based on Constraints, Technical Report No. AI–TR–555, AI Lab, MIT.

INDEX