High Level Test Pattern Generation for VHDL Circuits

Balázs Sallay, András Petri, Károly Tilly, András Pataricza
Department of Measurement and Instrument Engineering
Technical University of Budapest
H–1521 Budapest, Műegyetem rkp. 9, Hungary
Phone +36–1–463 2057
E-mail: sallay@mmt.bme.hu

Abstract

This paper presents a new approach that uses functional level circuit descriptions as a basis for automatic test pattern generation (ATPG). The VHDL model of the circuit is transformed into a constraint network, and the ATPG problem is solved as a constraint satisfaction problem. Techniques and heuristic methods for the acceleration of the search are also examined.
Keywords: *ATPG, high level, VHDL, constraint, CSP, heuristics.*

1 Introduction

The main goal of an automatic test pattern generation (ATPG) algorithm is to find such input values for circuits that detect given physical faults.

The growing complexity of digital circuits have imposed greater and greater demand on test pattern generation algorithms for the past twenty years. Though promising solutions have been elaborated to speed up the test generation process, no algorithm can cope with its excessive computational complexity, as Fujiwara has shown that the ATPG problem is NP-complete [1]. One possible answer, put forward by the design and test communities, to this problem was the invention and application of *design for testability* (DFT) principles. Full testability is, however, rarely affordable, and the development of ATPG algorithms is still an important issue.

Gate level algorithms, based on a systematic search [2], are very sensitive to the number of gates in the circuit. Although the speed of the test generation environment grows rapidly as well, this increase is less than the exponential increase in the demand of time, which is caused by the growing number of gates, would make it necessary.

The basic principle of the presented approach is to raise the abstraction level of the input of the ATPG procedure, in order to reduce the structural complexity of the circuit model. Instead of a collection of independent gates and signals, functional objects and signals of abstract data types are considered, thus the number of network elements are decreased. Obviously, an object described at a higher level will take more time to justify, yet a significant gain in the overall time requirement is expectable. This increase of performance is effected by the fact that high level elements can exploit the regularity of the components, which information is otherwise lost during top-down transformations of structured CAD methodologies. Another important advantage of high level test generation is that functional circuit models are sooner available and portable between technologies.

These advantages imply a somewhat reduced fault coverage: the more abstract the circuit model is, the coarser the fault model. However, since the vast majority of gate level stuck-at faults can be modelled at higher levels as well, this solution is a good trade-off between the gain in computation time and the loss in accuracy.

The following sections discuss the abstraction level suitable for ATPG as well as the corresponding fault model and fault coverage issues. Subsequently, a powerful constraint-based circuit representation method is introduced. Section 6 deals with the solution of the resulting constraint satisfaction problem, with special attention to heuristic measures that aid decisions. Finally, the efficiency of the algorithm and the accelerating methods is evaluated.

2 The role of high level ATPG in technology

High-level ATPG provides the following main advantages:

- It is possible to aim at the independence of test patterns of the employed technology. This also involves that the ATPG process may be launched sooner, prior to the gate level synthesis.
- Typically, the use of compact models results in a significant gain in computational time.

We consider the second objective as our primary research goal. The problems of early phase test development and technology-independence are not addressed in this paper for reasons discussed in Section 4. According to this, we view the abstraction level immediately following the high level synthesis step as the most suitable level for launching the ATPG process (Figure 1).

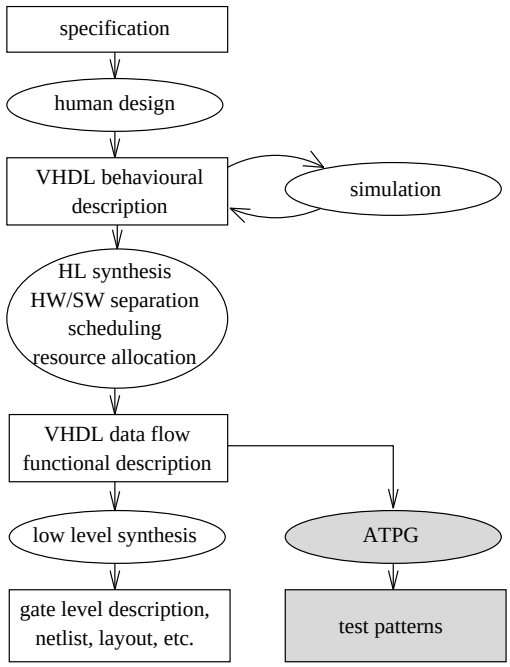

Figure 1: A typical CAD system design flow

We wish to integrate our ATPG tool under development into an existing design environment in order to evaluate its performance on circuits taken from real designs. The AMICAL system [4] has been chosen as the first CAD tool to be supported.

3 Circuit model

The applicable constructs of VHDL are more and more confined as the design process advances toward the lower levels. The functional architectural level, chosen as the entry point to ATPG, has a conforming description style as well. At this level two basic parts, composing the circuit itself, are separated:

- the *control part*, which is usually a finite state machine, typically described only by its function, and
- the *data part*, which describes how data move and are manipulated in terms of functional components and their interconnections. In a typical CAD system there is a set of predefined components constituting the data paths (such as multiplexers, registers, constant registers, etc.), and an open library of functional units performing manipulative operations.

The use of *multiplexer based* or *bus based* data part architectures does not imply an essential theoretical difference from the point of view of modelling:

- Multiplexers determine their output values using a distinguished selector signal.
- The *resolution function* of buses, associated to the bus signal in its *subtype declaration*, is truly value-driven and commutative.

By the applied constraint data structure, however, both kinds of selection are easily represented, since resolution functions can also be treated as functional elements.

In a cooperation between the Tallinn and Budapest technical universities, in the framework of the EC-sponsored FUTEG (Functional Test Generation) international project, our data-flow dominated and their control-oriented approach are being integrated. The approach based on the concept of *alternative graphs*, proposed in [9], offers the capability of dealing with circuits of deeper control sequences.

Our research focuses on generating tests for faults in the data part. At present, our approach is applicable to combinational models or to those of a very moderate sequentiality by the use of well-known extension methods, similar to the iterative array model based ones.

4 Fault model

The validity of the fault model is one of the most important issues considered at the selection of the

ATPG model. At present, several promising approaches exist which start ATPG from behavioural descriptions [7] [8]. The fault model is a strong point and also a weakness at the same time in these approaches. Their fault set, defined in terms related to VHDL, is independent of the subsequent steps (architectural, gate-level, and layout synthesis) of the synthesis procedure. On the other hand, as *scheduling* and *resource allocation* are not yet performed, the extent of hardware reuse is unknown at this phase. Since neither VHDL operations are associated with functional units, nor VHDL *assignment statements* are associated with data paths, reliable assumptions lack the correlation between physical faults and errors in VHDL statements.

We have therefore chosen the functional architectural level as the entry point to ATPG. VHDL signals here have direct hardware correspondents; hence, physical faults affecting interconnections are manifested as storage problems of these signals. The traditional *stuck-at* and *short* fault model is applicable here, too; moreover, our model makes the handling of multiple bit-faults on a single high-level signal possible.

To functional units (FU) functional faults are associated, e.g. the execution of a wrong operation. However, in an ATPG tool integrated into a design system based on FU libraries, a more accurate fault representation is possible: with the use of either

- *faulty unit libraries*, where library elements are obtained by the back-annotation of lower level faults, or
- *test situations*, where test patterns for FUs are given together with the FUs themselves.

This fault model defines a superset of the possible physical faults. The set of faults that may indeed appear in the circuit can only be determined when decisions and technological details of the subsequent low level synthesis step are known. Integer encoding, for example, affects the set of possible stuck-at faults, as well as shorts can be selected only after the wire placement is known. This means that ATPG is still not a stand-alone procedure; however, generic algorithms that need less time for a given fault can be developed.

5 Constraint based representation

Although only input patterns are of primary interest during the execution of a test pattern generation procedure, test vectors can be found only after the justification of the internal signals.

The requirements imposed on a justified signal setting can be expressed by the concept of *constraints*. Signals are treated as variables of discrete domains in our approach, while a constraint is a relation between them, which describes what consistent value combinations they may take. Since the prerequisites of a successful justification are determined by the architectural elements, constraints are used to represent the components of the circuit. This way a constraint network similar to the circuit topology is obtained, and additional constraints are added which guarantee that the assumed fault be sensitised and observed.

A vector in the variable space, satisfying the constraints, is the solution of the *constraint satisfaction problem* (CSP); test patterns are then the values of variables corresponding to the input ports of the circuit. Thus ATPG problems and CSPs are equivalent.

The constraint notation and representation is introduced on account of the symmetric nature of constraints. In contrast to circuit components, which have dedicated *in* and *out* ports and their output values are deduced of their inputs, a constraint does not have a predefined direction of data flow. This conforms very well to the justification problem where an algorithm assigns values to nodes in any order and attempts to propagate the effect of an assignment in the network both in forward and backward direction.

Representation of constraint variables

The role of variables in the ATPG is simply the storage of values assigned to them by the constraints. They maintain a stack of previous values in order to support backtrack mechanisms, and they have a special data structure supporting two features not common in CSP contexts: *partial access of constraints to variables* and *interval logic*. Partial access is required because VHDL allows the *indexing* and the *slicing* of signals. Interval logic is a data representation technique that supports the assignment of allowed ranges or of *masked array values* (i.e. an array value which has *don't care* elements) to variables, instead of specific values or domains. This logic often proves useful when values can be classified into groups of different behaviours and the separation of data-domain partitions can be postponed.

Representation of constraints

A constraint is generated whenever the VHDL compiler encounters a *component instantiation statement*. This constraint ensures the consistency of the values on the ports of the component.

Some of the conventional gate level ATPG algorithms [2] can also be considered as constraint based algorithms. Though a gate, e.g. OR gate, implements a function, its behaviour can also be described by means of a *cube set* ({0, 0, 0}, {X, 1, 1}, {1, X, 1}), which is in fact undirected and can be regarded as a constraint specified by the enumeration of the satisfying tuples (also called truth-table representation). They feature two considerable advantages: the access to a tuple requires only simple indexing instead of extensive computations, and the number of remaining tuples is always known.

Unfortunately, this representation cannot be extended to the high abstraction level, because the size of such a truth table would be extreme. Hence a symbolic or algorithmic representation methodology is required, which enables the constraint to perform the following tasks:

- *decision*: generation and assignment of a new tuple.
- *implication*: propagation of the effect of a change (a domain restriction) on a variable to the other variables. Implication is obvious when all the input signals become fully defined, as the output is a function of input signals. However, backward implication necessitates the use of more sophisticated methods.
- *support for other decisions*: calculation of different heuristic measures, such as the number of unassigned tuples, transparency, etc.

A high level ATPG tool can be implemented by either of the following two concepts:

- use of a *constraint library*. This concept is applicable whenever only a fixed set of components is instantiated in the input VHDL description, which is usually the case at the output of a high level synthesis tool. Tuple generation and implication methods can be provided in advance by the ATPG implementor.
- *compiled constraints*. This concept requires that the VHDL compiler be capable of processing the architecture bodies of the functional units as well as of generating their corresponding constraints.

The compilation methodology for functional units of a true data-flow nature has been solved in our approach by the creation of an elementary constraint network that consists of primitive nodes. However, the transformation of functional units of complex procedural nature cannot be handled in a satisfactory way with the current arsenal of data-flow based constraints. Consequently, in the first version of our software package the constraint library implementation is used for the modelling of functional units.

Fault representation

To make the constraint satisfaction problem fully equivalent to the ATPG problem for the given circuit, two further constraints are added to the network. The first one is responsible for the fault representation, while the second one ensures the observability of the fault effect. As faults are represented the same way as other information on the circuit, the fault model can be extended arbitrarily if the new fault can be described by custom constraints.

The modelling of stuck-at and short faults on signals is quite straightforward, because they have a direct correspondent in the generated constraint network. Thus signal faults are transformed into simple fault monitoring constraints. Functional faults are also easily represented by the addition of a multiplexer and a component that implements the wrong functionality.

6 Solution of the CSP

A solution of a CSP signifies a value setting of the variables which satisfies all the constraints. To obtain this, a search algorithm that traverses the variable space is activated. This algorithm must be systematic, i.e. it must find a solution if the problem is solvable.

The most primitive ("brute force") method to find a solution would be an exhaustive search by the enumeration of value tuples, then the selection of those tuples that comply with every constraint. Somewhat more sophisticated is the algorithm that lists only tuples of variables corresponding to circuit inputs and calculates the rest. These algorithms without a special search control consume a huge amount of time, because a lot more tuples are examined than would be necessary. A much better version of the second algorithm is when only a subset of input variables is assigned first and this set is gradually extended if no inconsistency is experienced. By that we actually get the high level equivalent of the gate level PODEM algorithm. The cost of improvement is the emerging need of a decision control mechanism.

Algorithm candidates too have to possess the ability of partial and extendible variable assignment, inconsistency detection, and decision nullification. The execution of such search procedures can be illustrated by the well-known *decision tree* (DT) notation. The nodes of a DT, ranging from the root to the leaves of the tree, represent more and

more determined states of the overall system. Edges denote decisions that transfer the system down one level, into a more specific state, by restricting or assigning values to variables. Leaves of the DT may either be inconsistent states where backtracking step is to follow, or the goal (entirely determined and consistent) node.

These algorithms consist of two basic steps in the traversal of the DT: a *forward* and a *backward* step. During a forward step a decision is made and some variables become determined. If the state of the system remains still consistent (more precisely, no inherent inconsistency is discovered), a forward step follows again. If an inconsistency is detected, a previous state, assumed to be consistent, is restored, and a next decision alternative is tried. This is called a backward step.

The simplest algorithm implementing these steps is *chronological backtracking*, widely used e.g. in traditional gate level ATPG algorithms. In case of a contradiction some decisions are retracted in reverse chronological order until the contradiction is resolved. Figure 2 describes the algorithm in details, during which consistent values are assigned consecutively to variables in an order $\{X_1, X_2, ..., X_n\}$ determined by the applied selection method. After a consistent state i is reached, a further assignment to X_{i+1} is tried. If no consistent value can be assigned to X_{i+1}, then it is a dead-end state, and a backtracking step must be performed. During this step, a variable X_j is selected where $j < i$ and X_j has still multiple candidate assignments. Its value is altered to the next consistent choice.

```
Forward(x1, x2, ...xi)
   if there are no more variables to assign   then SUCCESS
   Let Di+1 be the set of values of Xi+1
        which are consistent with {x1, x2, ... ,xi}
   while Di+1 is not empty do
      Let xi+1 be the first element of Di+1
      Remove xi+1 from Di+1
      Forward(x1, x2, ... ,xi,Xi+1)
   Backtrack(i)
end Forward

Backtrack(i)
   if i=1 then FAILURE
   Unset Xi
end Backtrack
```

Figure 2: Chronological backtracking

This fairly simple algorithm surely finds a solution if it exists. The search trace on the corresponding decision tree can be mapped onto an easy-to-implement stack. The efficiency of the chronological backtracking can be in several cases extremely low due to the discrepancy between the chronological order of backtracks and the problem structure. Suppose that a decision in the middle of the DT is the cause of a contradiction that is discovered only at the leaves of the tree (Figure 4a). In this case chronological backtracking performs an exhaustive check of all underlying leaves before rejecting this decision alternative. This is entirely unnecessary, since subsequent decisions can have no impact on the outcome of that subsearch, which is surely failure and which would cause the withdrawal of the wrong decision.

Our proposed algorithm applies two significant improvements compared to this one. It incorporates an *advanced implication step* after each decision in the forward phase, and skips considerable parts of the tree without the risk of losing solutions in the backward phase by *intelligent backjumping*.

Implication

Once a decision is made in the course of forward extension steps, it is worth estimating the effects implied by that decision in order to reduce the search space by the exclusion of inconsistent states.

```
Implication()
   Let I, the queue of elements about to imply, be empty
   Place the variables of the constraint
        that have just made the decision on I
   while I is not empty do
      Remove e, the first element of I, from I
      e.Imply()
end Implication

constraint::Imply()
   Find which variables have been restrained
   Propagate confinement to other variables
   Put newly restrained variables to I
end constraint::Imply

variable::Imply()
   Put neighbouring constraints to I
end variable::Imply
```

Figure 3: Implication

If this implication step were absolutely perfect, the exploration of all the impacts of a decision would entirely eliminate the need of backward steps, since a complete implication would not leave latent contradictions in the system state. Perfect implication is, unfortunately, only a theoretical option, since this itself would be as time-consuming as the ATPG process is. Nevertheless, the effect of a decision must be discovered to as great an extent as reasonable.

Thus an implicative step is incorporated in the forward step of the search. As a reasonable trade-off between implication perfectness and efficient implementation, one-step implication is used (Figure

3). The shown method is a breadth-first propagation of value restrictions, which could just as well be implemented in a depth-first manner. The crucial point of the algorithm is the propagative step in the middle of the constraint implication method, which is implemented in an efficient way in our constraint library based adaptation.

Backjumping

Whenever a contradiction is discovered, it must be eliminated by restoring a previous, contradiction-free state. In chronological backtracking the state preceding the last decision is restored first, where either a new decision alternative is selected or the decision history is traced further back until the consistent state is found. As this state may be as well located in the middle of the DT, jumping immediately back there would be a great acceleration without losing solutions (Figure 4b).

The target of a jump-back, on the other hand, is not easy to find. Still, this can be done in a CSP originating of an ATPG problem, since the nature of ATPG is such that structural data dependencies help find candidates for the wrong decision, of which the last one can be immediately cancelled. Consider, for example, the case when a constraint realises that no value tuples satisfy its condition (*inconsistency detection*). Hence, a backward step needs to follow. Now there is no use in taking back the last decision, made somewhere else in the network, because the only way to eliminate the contradiction is to widen the range of the neighbouring variables. The wrong decision candidates are, therefore, those that may have had an impact on these variables.

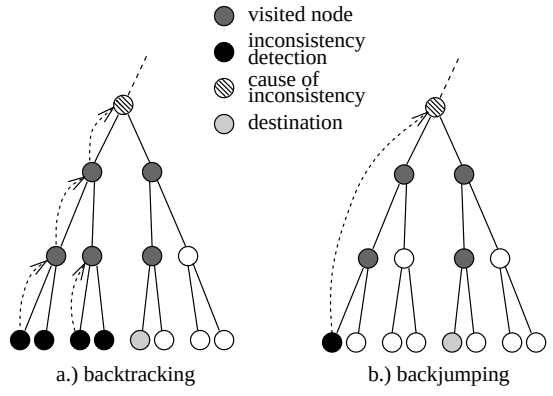

Figure 4: Comparison of backtracking and backjumping

Backjumping needs no more computation in the state administration than backtracking. By maintaining simple DT level stamps with decisions data dependencies can be easily explored.

Labelling of network nodes

An additional idea beside the ones above is to keep the solution space as narrow as possible by the application of circuit topology based reductions on the domain of the variables. Variables can be classified in the following way in the initial phase of ATPG for a given fault (Figure 5):

- those that must be affected by the fault (*active* nodes), i.e. those that carry different values in the good and the faulty circuit;
- those that may be affected by the fault (*potentially active* nodes);
- those that cannot be affected by the fault (*inactive* nodes), i.e. those that carry identical values in the good and the faulty circuit;
- those of irrelevant value (*don't care* nodes).

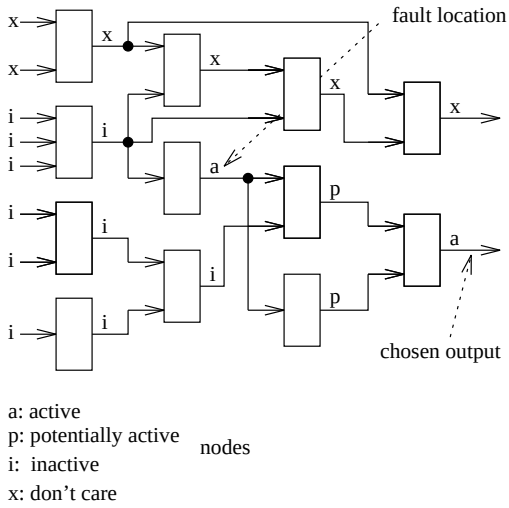

Figure 5: Labelling of nodes

Since constraints with *don't care* variables do not make decisions at all, the depth of the DT is decreased with several levels, thus the time consumption is radically reduced (see Section 8). Inactive and active variables also involve important data-domain confinements.

These search acceleration concepts drastically reduce the number of traversed nodes in the DT, that is, the time required by the algorithm; yet the depth-first nature of the search method remains unaltered. A sensitivity to early decisions is very characteristic of the recursive search algorithms. The order of the decisions and the extent of value restrictions should therefore be carefully determined.

7 Heuristic decision support measures

As exact information on the effect of decisions is unavailable and would be extremely expensive to obtain at the time they are made, heuristics are used in order to predict the effect of the decision alternatives, thus helping the selection of the potentially best one. Unlike in the case of the accelerating methods enlisted in the previous section, the appropriateness of the individual kinds of heuristics cannot be exactly proved, only validated by means of benchmarking.

Many kinds of heuristics have been examined by the authors. Some of them depend only on the topology of the network (*static heuristics*), while others are recalculated after certain changes in the network state (*dynamic heuristics*). A sophisticated algorithm uses a properly weighed combination of several measures. At the current phase of our research we focus on each heuristic method separately, but we intend to inspect their combined effect after an experimental validation. At the gate level, however, it was shown in [10] that by the extreme weighing of different measures it is possible to implement each of the well-known gate level ATPG algorithms such as PODEM, composite justification [5], etc. as special cases of a meta-algorithm. In the following, certain cost function components will be discussed.

Structural decisions

One classification of decisions is based on the nature of their impact. When the result of a decision is simply the confinement of the domain of certain variables, this decision is called a *data-domain decision*. Decisions may, however, involve that subsequent operations, mostly implication, become easier to perform, because the part of the circuit with the given values can be replaced with a simpler structural equivalent. We call these decisions *structural decisions*.

Example:

Consider a multiplexer constraint operating on *Integer* variables and on a selector variable (Figure 6). If the selector is set to a specific value, then subsequent implications performed by the constraint are simple assignments from the selected input to the output, or vice versa. This is a structural decision. When, however, the selected input (together with the output) receives a specific value, a simple data-domain decision is made.

It is desirable to make structural decisions first, and then perform data-domain justification only in a significantly simplified circuit, because the exponential explosion of time consumption is due mostly to the data-domain complexity of the model. Therefore, a number which measures the implicative improvement in the overall circuit is associated to each decision candidate.

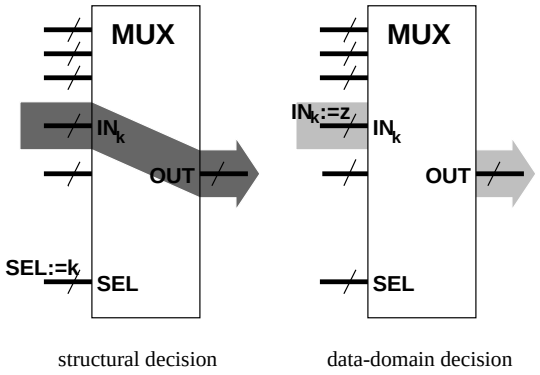

Figure 6: Structural and data-domain decisions

A practical data-independent cost function can be derived from the classical *observability* and *controllability* measures proposed in [3]. In order to make the effect of a fault observable, a fault propagation path must be established between the fault site and the chosen output. The classical idea, first mentioned in the late sixties, is to select the shortest fault propagation paths (i.e. that have the highest observability), as the establishment of these paths involves the least interference with other components.

The length of fault propagation paths can be measured by a simple topological distance between the output and the fault site. However, it is more profitable to use – in a way similar to the recursive cost functions shown in [3] – a measure of difficultness where the characteristics of the components involved in the fault path are also considered. The establishment of a fault path requires value assignments and value range confinements which make the involved components transparent, and the number of these extra assignments determines the amount of interference produced.

A very important factor is the width of the fault path, i.e. the number of "parallel" abstract data lines with different faulty and fault-free values. This must be kept as low as possible close to 1 in order to prevent the state space from growing. This cross-cut number can be maintained by the introduction of a supervising constraint, similar to the *"D-frontier"* concept of gate level algorithms. By cost punishments the search algorithm can be forced to try values that block unnecessary fault paths, e.g.

by assigning 0 to the other input of a multiplier component.

Our future plan is a hierarchical *structural/ data-domain separation* of decisions by the elaboration of a two-phase mechanism. Since high level ATPG is integrated into an overall design process, external information incoming from the synthesis steps can be utilised by the ATPG tool. The most useful is the information which defines the highest level data paths of the circuit. The extraction of this information is quite straightforward if the CAD system applies a modelling mechanism similar to the *hardware-software co-design* approach, proposed in [6]. In these systems the highest level of abstraction is an untyped data flow model where only the direction and the timing of the data flow is considered without detailing data manipulation. In the first phase global fault-free and faulty data paths can be defined, and a test superset can be determined. This superset is tightened later by data-domain dependencies, so the second phase of test generation is performed, i.e. when the ATPG deals with exact data types and performs complete data-domain justification.

Constraint selection

It has been shown that the risk of causing interferences among decisions should be minimized. Besides implication, another idea is to let the data be propagated in a wavefront-like manner in order to avoid creating multiple independent decision centres in the network. Thus only those constraints are allowed to make decision which have variables of an already confined range. Out of those constraints that are still considered after this restriction, the selection of the decision-maker is aided by the *lowest tuple number* heuristics. According to this, those constraints make better decisions that can choose from less tuples, because they partition the problem space better and are less likely to lead to contradiction. This requires the capability of counting or estimating the number of the untried tuples.

Data-domain decisions

At one point, the CSP solver algorithm gets into the justification phase when all the values must be determined. The best choice is not necessarily the assignment of fully determined values. For example, consider the case when an assignment of value 1 proves to be a wrong decision. After its cancellation, there is little hope in trying value 2, because it is very likely that the result will be the same contradiction. The problem here is running into details too early. It is perhaps better to merely restrain the value range of the given variable, and to postpone further decisions concerning that variable. A cost function, observing the history of successful values, can be set up to control this method.

8 Measurement results

In order to evaluate the presented algorithms and heuristics, a high level TPG tool is being developed. It consists of several modules and is implemented in an object oriented manner in C++ (Figure 7). This tool is the second version of our TPG software package. Preliminary measurements have been made in a HLTPG environment, implemented, oddly, in VHDL, too. The highest programming level subset of VHDL was used for that. An algorithm written in a HDL was not, of course, very powerful, but spared us a lot of compiler development and other programming work, and served very well for the evaluation of the applied heuristic methods. We summarize now only the effect of those accelerating methods and heuristics which were elaborated at the time this preliminary version was used.

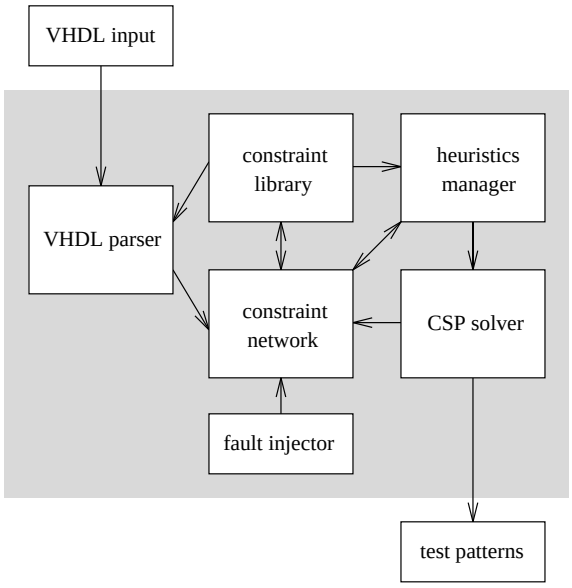

Figure 7: Module hierarchy of the HLTPG tool

The algorithm was executed on many benchmark circuits of two abstraction levels, mainly of complex reconvergent structure, with measured constructs switched on and off. High level circuits were put together of abstract type signals and components performing arithmetic operations; low level descriptions contained bit-level signals and gates. Particularly interesting is the ATPG for undetectable faults: when a fault is not detectable, exhaustive search must be performed with the traversal of the entire decision tree.

Let us first examine the impact of accelerating ideas that bring improvement compared to simple backtracking (Section 6). To obtain their pure impact, heuristics were switched off for these measurements (Table 1). Numbers in the table signify the number of backward steps, which is proportional to the overall time consumption; *h6* denotes a high level circuit with 6 functional units, while *det.* and *undet.* mean detectable and undetectable faults, respectively. It can be seen how impressive the result of the labelling algorithm is; implication entailed less improvement at the high level than expected, however, we believe this is merely caused by the moderate size of the circuits.

	none	node labelling	implication	both
h6, det.	16	14	16	14
h6, undet.	132884	2087	132884	2087
g25, det.	797029	20452	1029	283
h11, det.	32624	1373	32624	1373

Table 1: Impact of accelerating methods

One heuristic cost function has been examined: the one helping the selection of the decision-maker constraint by the consideration of the number of available tuples. The result of its application is indeed radical, as Table 2 shows: the improvement was especially enormous at undetectable faults.

9 Future work

There are some important directions in which our high level ATPG concept can be improved with further research. The constraint library based version should always follow the CAD technology: those components must be used which are supported by current high level synthesis tools. However, the library extension via automatic functional unit compilation will be a qualitative change.

	with	without
h6, det.	5	14
h6, undet.	295	2087
g25, det.	18	283
g25, undet.	24	1221716
h11, det.	112	1373
h11, undet.	257	3281

Table 2: Impact of tuple counting heuristics

At present, the focus of our research is confined to combinational circuits, or to those of a very moderate sequentiality for which iterative extensions can be applied. The integration with a control-dominated approach [9] is planned in the near future.

References

[1] H. Fujiwara: Logic Testing and Design for Testability. MIT Press, Cambridge, Mass., 1985

[2] M. Abramovici, M. Breuer, A. Friedman: Digital Systems Testing and Testable Design. Computer Science Press 1990

[3] L. H. Goldstein: Controllability/Observability Analysis of Digital Circuits. IEEE Trans. on Circuits and Systems, Vol. CAS-26, No.9, pp. 685-693, September, 1979.

[4] A. A. Jerraya et al: AMICAL – Interactive Architectural Synthesis Based on VHDL. INPG/ TIMA System Level Synthesis Group, Grenoble, 1994

[5] J. Sziray: Test Calculation for Logic Networks by Composite Justification. Digital Processes, Vol. 5, No. 1-2, pp. 3-15, 1979

[6] Gy. Csertán, A. Pataricza, E. Selényi: Dependability Analysis in HW-SW Co-design. IEEE IPDS'95 International Computer Performance and Dependability Symposium, April 1995, Erlangen

[7] D. S. Barclay, J. R. Armstrong: A Heuristic Chip-Level Test Generation Algorithm, 23rd Design Automation Conference, pp. 257-262, June 1986

[8] E. Gramatova, T. Cibakova, J. Bezakova: Test Pattern Generation Algorithms on Functional/ Behavioral Level, Technical Report FUTEG-4/1995

[9] R. Ubar: Test Generation for Digital Systems Based on Alternative Graphs, Dependable Computing - Proc. of EDCC-1, pp. 151-164, Springer-Verlag, 1994

[10] A. Pataricza, K. Tilly, E. Selényi, M. Dal Cin: A Constraint Based Approach to System Level Diagnosis. Internal Report 4/94, IMMD III, Friedrich-Alexander Universität, Erlangen.