PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 42, NO. 3, PP. 311-335 (1998)

NEW ALGORITHM FOR BEHAVIOURAL TEST GENERATION[1]

Balázs BENYÓ

Department of Measurement and Information Systems
Technical University of Budapest
H-1521 Budapest, Hungary
Fax: +36 1 4632 057, E-mail: benyo@mit.bme.hu, galett.bme.hu .

Received: Dec. 10, 1995; Revised: Aug. 8, 1998

Abstract

Significant efforts of the test design community have addressed the development of high level test generation algorithms in the last decade. The main problem originates in the insufficiently low gate level fault coverage of test sets generated at the behavioural or functional levels due to oversimplifications which result from the application of highly abstract and technology-independent fault models.

In this paper a novel behavioural level test generation algorithm is presented effectively utilizing information on the circuit structure, which is extracted from the high level synthesis process.

Experimental results show that the gate level fault coverage of the test sets generated by the new algorithm is similar to those assured by the gate level test generation algorithms.

Keywords: automatic test generation, behavioural level digital circuit synthesis, path testing, VHDL.

1. Behavioural Level Automatic Test Generation

The radical growth in the size and complexity of VLSI systems results in significant problems for automatic test generation (ATG). Gate level test generation for complex digital circuits – containing hundreds of thousands of gates – is impossible due to the huge computational complexity.

ATG for these complex systems requires the use of high level circuit descriptions – such as those at a behavioural or functional level – in the test generation process. The recognition of this problem was the first factor triggering the development of the behavioural level ATG algorithms.

The other factor radically accelerating research was the development of automated synthesis systems generating the circuit layout directly from a behavioural specification. Both in the bottom-up or in the top-down

[1]This research was sponsored by the Hungarian National Scientific Foundation, grant W 015411. The research project has additional support from the research grant of the European Community, FUTEG project No. 9694.

synthesis approaches the circuit is described at varying levels of abstraction. At the different phases of the design process designers have to generate tests to validate their actual design models.

1.1. Application Fields of High Level Test Generation

The goal of the test generation at the behavioural level depends on the intended application of the test vectors. *Behavioural* test vectors (e.g. test vectors generated from the behavioural description of the digital circuit) are basically used for two purposes:

- *validation* and
- *final test* of the circuit.

In the case of validation the test set is used for testing the functional equivalence of different descriptions of the same circuit, thus validating the transformations between bvarious levels of abstraction. The most common application area of validation – as mentioned earlier – means the high level automated synthesis systems [2].

The other potential field of application of behavioural txest vectors is their use for *final testing* of manufactured circuits. This is the traditional application field of the test vectors when the tester intends to discover manufacturing faults by test sets.

The main problem in this second field is that the *physical fault coverage* of the behavioural test set is generally *low*. The fault coverage of a behavioural test set – even though it hardly depends on the tested circuit – is typically around 50% of the low-level faults [4].

1.2. Quality Criteria for the Test Set

The evaluation of the quality of the test vector set may be based either on

- the traditional low level quality criteria, e.g. the *gate level fault coverage* and the *length* of the test set or
- other criteria defined at *more abstract* digital circuit modelling levels.

A typical example for such a special criterion is the so-called *path coverage*. In the case of control flow graph (CFG) based test generation each test vector traverses a particular path in it. The path coverage defines the ratio of the traversed paths and all paths in the CFG.

The type of the actually used criterion depends on the intended application of the test vectors. Validation takes place typically at the higher levels of abstraction. Accordingly, criteria defined at high levels of abstraction are applied in the case of validation [5] [10].

The applied test quality criterion for manufacturing tests cannot be more coarse than the gate level fault coverage as this is the highest level still providing a proper model for faults originating in the technology.

1.2.1. Gate Level Fault Coverage

The measurement of gate level fault coverage is based on the *stuck-at fault model*. This is the most widely accepted fault model of the test design community since the early sixties [7]. Even though the stuck-at model is one of the simplest fault models it has some basic advantages:

- *Realistic* – proper model of the physical failures for the majority of static defect mechanisms.
- *Easy to handle* – it is simple and appropriate for simulation and modeling.
- It is *widely used* – allowing a comparative analysis of effectiveness of the different test generation algorithms.

The correlation between the gate level and the physical fault coverages is usually so high that they can be considered as approximately identical ones from the practical point of view. The advantage of the use of the gate level fault coverage as test quality measure in the face of physical fault coverage is that its estimation is essentially easier as it requires only the knowledge of a gate level logic model instead of the much more detailed transistor or layout level model. Correspondingly, gate level fault coverage is commonly used as the evaluation criterion of manufacturing test sets.

1.3. Classification of Behavioural Level Test Generation Algorithms

The very first functional level test generation algorithms were developed at the beginning of the eighties [8] [9] [11] [12] [13] [14]. Several behavioural level test generation methods have been developed since that time.

Low level ATG algorithms were classified at the end of the eighties according to the *fault model* applied [1]. Now, a similar classification of the high level ATG algorithms will be given. There are three major classes of high-level ATG algorithms:

- *Behavioural fault model* based algorithms define an own fault model for test generation. For instance, these fault models describe physical faults as incorrect executions of a statement in the behavioural description of the circuit. The test generation process is divided into two phases: *model perturbation and propagation of the effect of the fault to the output.* The advantages of these fault models are their

easy implementation and the flexibility during simulation. The basic weakness of these fault models is the incorrect correspondence to the physical faults, thus the fault model is not *realistic* [3].

- *Implicit* fault model based algorithms aim at the use of a very general fault model instead of a particular one. This implicit fault model assumes the occurrence of any permanent fault in the system with the exception of those which increase the number of the states in the system. Test generation is performed exhaustively based on a mathematical model of the circuit such as the CFG or the state transition diagram [5] [15].
  These algorithms efficiently test the control sequence of the digital circuit but do not cover the faults of the other parts of the circuits within practical run-time limits. In the cases of the control dominated circuits, the fault coverage of the test set is high – even if measured in terms of low level faults – but in circuits containing arithmetic logic, data storage, or manipulation parts the fault coverage is low. Another drawback of this approach is a potential over-testing, generating tests for hypothetical faults not appearing even in the low level fault model.
- *Ad-hoc* algorithms do not assure a general solution to the test generation problem, as these deliver only ad hoc solutions in the case of existence of some kind of special conditions. For example several setups of the hierarchical test generation or the random test generation belong to this category [4] [6].

1.3.1. Algorithms Using Behavioural Fault Model

The Behavioural Test Generator developed at the Virginia Technical University is one of the characteristic representatives of the ATG algorithms using behavioural fault model.

A subset of VHDL is allowed for modelling the behavior of the circuits in this approach. Both control and data fault models are developed which perturb the operation of the language constructs. All of the possible faults of the circuit are injected into the behavioural model once. The test vectors are estimated by searching for an input vector propagating the effect of the actually injected fault to a primary output of the circuit.

The major classes of faults injected are as follows:

- *Micro-operation fault* – instead of the execution of a basic VHDL statement another one is executed, e.g. OR statement instead of AND statement.
- *Assignment control fault* – the assignment of some new value to a variable or to a signal is not executed.
- *Dead-clause fault* – In the case of a conditional branch statement like a CASE statement the selected branch is not executed.

Such an algorithm is implemented in a constraint based ATG environment.

1.3.2. Implicit Fault Model Based Test Generation by the Example of Path Testing

The class of the implicit fault model based test generation algorithms will be illustrated by the *path testing* algorithm, originally developed for software validation. For software validation the CFG of the tested program is extracted at first and subsequently the operation of the program is validated by executing the program statements along several paths of the CFG and by comparing the effect of the executed operations on the output variables with the specification.

The behavioural level model describes the function of the circuit under test by means of sequential statements in a similar program-like way so the CFG of this description can be extracted as well. The validation of the hardware is similar to the software validation: the operation of the circuit is tested by simulation applying the input vectors causing the traversal of the different paths in the CFG. Accordingly, the goal of this ATG approach is the estimation of an input vector sequence traversing all the paths in the CFG of the behavioural circuit description.

The algorithm is typically implemented in a constraint based form. The conditions of the execution of a particular path are translated into constraints. The solution of the several constraint sets defines the input vectors traversing the current path. The test set consists of these input vectors. The path testing algorithm will be introduced in detail in *Section 4.3*.

2. Automated Digital Circuit Synthesis

Modern computer aided design environments for digital circuits synthesis accelerate the design process by automating several phases of the design flow. Nowadays some manufacturers offer automated synthesis systems designing the digital circuit layout directly from the behavioural specification of the circuit without any kind of human interaction. The most common features of the recently developed automated synthesis systems are as follows:

- *Top-down design flow*
  The design process is divided into several independent phases. An increasingly detailed description of the digital circuit is developed through stepwise model refinement during the synthesis process. The abstraction level of the developed descriptions decreases by each step of the model refinement process.
- *Library-based synthesis*
  The synthesis system contains a component library including the description of the most frequently used circuits. The design system composes the designed circuit of multiple instances of these predefined circuits. The design system binds the current design to the library

components and implements them as many times as they need. Subsequently, the final part of the synthesis is connecting the registers according to the bindings.

The typical structure of a top-down digital circuit design process is illustrated in *Fig. 1*. The input of the design process is the *behavioural specification* of the circuit. During the design flow in each design phase a new, more detailed circuit description is generated at the lower level of abstraction. In the figure three abstraction levels are outlined:

- Register level
- Gate level
- Layout level

In the case of fault-free design process the generated specifications are functionally equivalent descriptions of the same digital circuit.

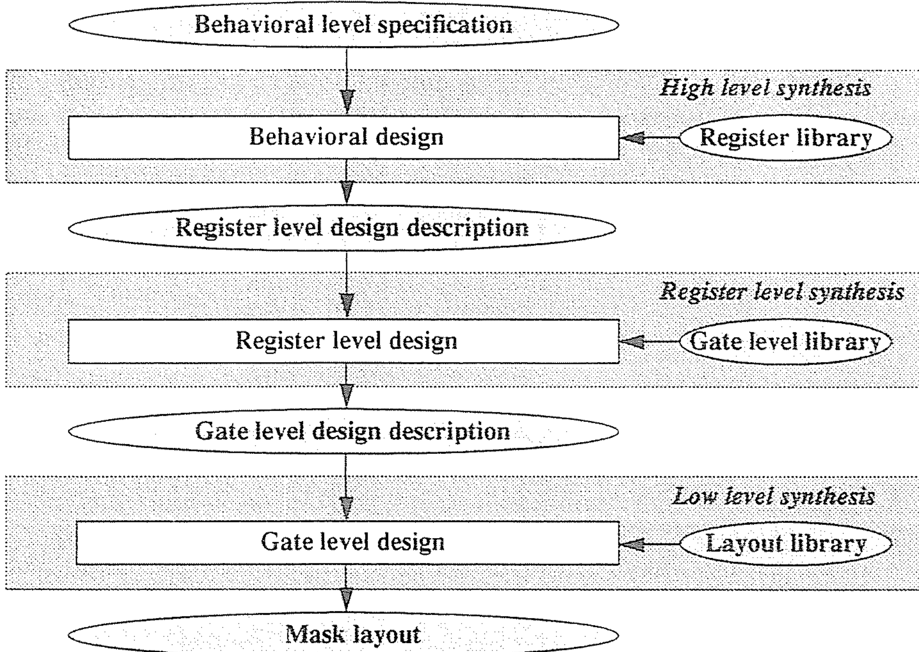

Fig. 1. Basic scheme of the top-down design process

2.1. Synthesis Libraries

The design libraries contain the description of the most commonly used circuit components at the different levels of abstraction. The description of these circuits is special in the aspects that they describe the circuits *in*

terms of the input bit-width. This description technique is necessary in order to decrease the size of the library.

At the higher levels of abstraction this description technique of the circuits does not cause any problem. The recently developed hardware description languages include this modelling feature like the VHDL contains the possibility of the definition of the generic statement.

The structural description of a digital circuit parametrized by the bit-width of the circuit is much more difficult at the lower levels of abstraction. Only a small part of the digital circuits can be described in this way, called *bit-sliced designs.*

The bit-sliced circuits are built up by the chain of equal sub-circuit components (which can be called a slice). Each component processes one single bit of the input and generates one or more output bits belonging to the current input bit. The components are connected by signals but only the links between two subsequent components are allowed. These connections make the bit-sliced design similar to the chain.

Because of this easy application of these circuits in the automated synthesis systems the importance of these circuits has radically increased in the last few years.

2.2. Behavioural Level Synthesis

The very first phase of the digital circuit synthesis is the *behavioural (high) level synthesis*, generating a register-transfer level model of the actual circuit based on its behavioural specification.

The cardinal difference between these two descriptions is that while in a behavioural description *data path* and *control* statements can appear mixed, at the register transfer level they are already separated. The main function of the behavioural design phase is the decomposition of the current circuit into controller and data part (*Fig. 2*). These two parts of the circuit are handled separately in each phase of the design process after behavioural synthesis. Even the manufactured digital circuits can be divided into these two main parts:

- Controller
- Data part consisting of the arithmetic logic and data storage

The arithmetical and data storage elements cover the evaluation of the data part of the RT description. The control part of the circuit corresponds to the controller in the RT description.

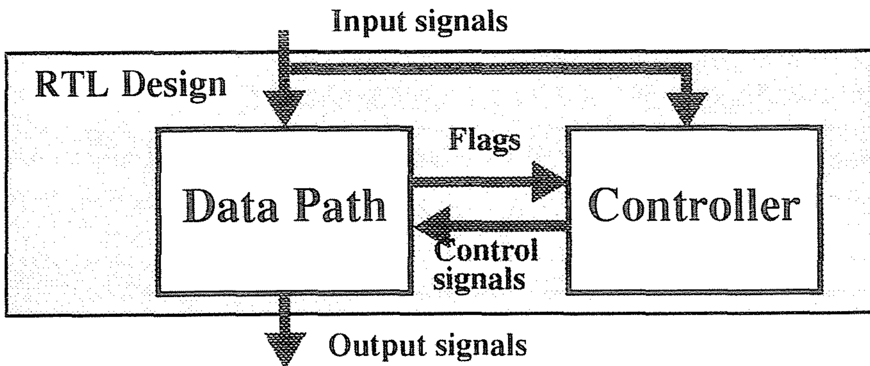

Fig. 2. Structure of the register transfer level design

3. New Approach of the Behavioural Test Generation

The gate level fault coverage of the test sets generated by traditional behavioural level ATG algorithms is insufficient for a thoroughgoing test of circuits, especially in the case of arithmetics intensive circuits. The novel algorithm proposed in this chapter generates test sets of a high fault coverage even for this class of circuits.

Implicit fault model based algorithms generate high fault coverage tests for control intensive circuits as described in *Section 1.3.* The new ATG algorithm incorporates a modified version of the path testing algorithm to generate test vectors for the control part of the circuit.

However, a good test quality for the arithmetic logic and data storage elements necessitates as first step the definition of a *realistic* fault model. Since previous experiments from the literature have clearly shown that a realistic description of the physical faults at the highest, behavioural level is impossible, a version of the gate level *stuck-at fault model* is used for test generation in the arithmetical, data storage and manipulation parts. In library based systems the components in the data part are substituted by macros from the library. The basic idea of the novel ATG algorithm is to generate elementary tests corresponding to the components in the library and using a back-annotation technique to embed them into the behavioural level test generation process.

The advantage of this solution is that the fault model applied can exactly describe the effects of technology-related defects in the data part, while omitting an over-detailed modelling of the physical faults in the control part of the circuit.

3.1. Fault Model Generation

The fault model can be generated for any library based synthesis system. As first step a thoroughgoing test set is generated from the gate level description of the components in the design library using the gate level stuck-at fault model. The gate level descriptions of the elementary library components of a modern design system are typically scaleable, i.e. they are parametrized by the bit-width of the function to be implemented.The generated test set of the library components has to be described in terms of the bit width of the function as well in order to save the scaleability for the ATG. In this case the gate-level ATG has to be executed for several instances of the registers and subsequently a parametrized form of the test vector set is to be estimated. This phase of the test generation involves human interaction but it should be performed only once for each synthesis system.

These scaleable abstract test vector sets consist of the symbolic list of input values sensitizing the component for stuck-at faults. This set of 'sensitive' values on the input ports of the library components can be considered as an equivalent of their register level fault model.

After this definition of the register level fault model, the register level faults have to be 'back-annotated' to the behavioural level by performing the inverse transformation of the design process. In more detail, the variables in the behavioural description are to be identified which are transformed to input ports of the components during the design process.

After assigning the 'sensitive' values to the variables ATG reduces to a well-known behavioural level justification problem. The fault model generation method corresponding to a top-down design system is shown in *Fig. 3.*

The development of the behavioural fault model in the Multicomponent Synthesis System is described in detail in the next chapter.

4. Direct Component Testing

A new test pattern generation algorithm, called *Direct Component Testing* (DCT), has been developed based on the idea introduced in the chapter above. In this algorithm, the circuit is modelled at the *behavioural level* (i.e. the test generation process deals with behavioural level objects) but the well known *stuck-at fault model* is used for modelling the physical failures of the components of the digital circuit.

The low complexity of DCT originates in generating test vectors based on the behavioural model of the circuit. The high fault coverage of the generated test set is guaranteed by using the stuck-at fault model.

Test pattern generation is possible by DCT for circuits designed by a library-based digital circuit synthesis system, i.e. standard components are

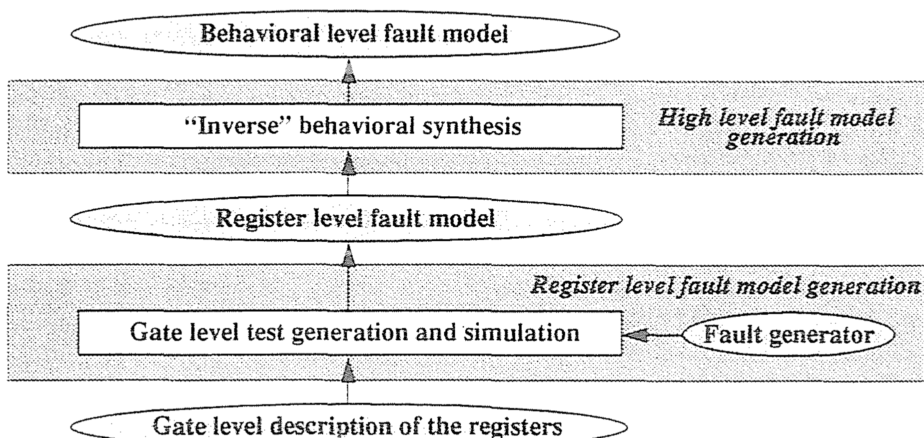

Fig. 3. Fault model generation at the behavioural level

used in the circuit and the data path description of the register transfer level design is available.

The test generation algorithm according to DCT can be divided into two phases:

- *Synthesis system dependent installation phase*
- *Target circuit dependent test generation phase*

The main difference between these two phases lies in the frequency of execution. The synthesis system dependent installation phase is executed just once, in the installation of the test generation system or in the case of changing the design library of the synthesis system. Actions belonging to the target circuit dependent phase are executed when generating each test vector.

The DCT algorithm consists of the following actions:

- *Synthesis system dependent installation phase:*

1. Generate general test sets for the components of the design library of the synthesis system.

- *Target circuit dependent test generation phase:*

2. Take the behavioural description of the circuit under test and enumerate the feasible paths of CFG of the behavioural model.
3. Recognize the components used and their interconnection in the implementation of the circuit based on the Register Transfer Level (RTL) description.

4. Extract the mapping between the carriers in the behavioural description and the components in the RTL description.
5. Generate input vectors for each feasible path warranting the component test vectors on the input ports of the components in the path.

All the listed phases will be described in detail in later sections.

There were other practical considerations during the implementation which have to be mentioned. As almost all the modern integrated synthesis systems use scalable design library components by the input-bit width of the components, the new test generation algorithm has been developed supposing that the test generator incorporates such a synthesis system even if it is not a necessary requirement for the application of the DCT algorithm.

4.1. Direct Component Testing Algorithm Description

In this section, a detailed description of the digital circuit testing algorithm, called Direct Component Testing, will be given. During the description a constraint based implementation environment is assumed.

The input data used in the DCT algorithm are as follows:

- Behavioural level description: B
- Gate level description: G
- Bit width dependent description of the test sets of the components in the design library: TestSetDescription(N_i, x_i) (for component called N_i : x_i is the bit width of the component).

The algorithm is divided into several parts:

Algorithm A: Generation of Path Predicate Constraint Set for Feasible Paths

Note that this algorithm is very similar to the Path testing algorithm.

Given input data is the behavioural level description of the digital circuit noted by B.

STEP 1. Extract the control flow graph (G_B) of the behavioural description B. $G_B = (V_B, E_B)$, $G_B = cfg(B)$.

STEP 2. Enumerate the CFG paths by traversal. The complete set of paths is denoted by $P_\Sigma = \{P_1, P_2, \ldots, P_s\}$, where P_i are paths in G_B. According to the definition of the path P_i is a vector consisting of vertices:

$$P = [\text{START}, v_1, v_2, \ldots, v_i, \ldots, v_n],$$

and

$$v_i \in V_B,\ \forall i < n\ \exists e_{i(i+1)} \in E_B.$$

STEP 3. Generate constraint sets called *path predicate constraint set* for each CFG path. $PP_i = (X_i, C_i)$ denotes path predicate constraint set belonging to path P_i. X_i is the set of variables and C_i is the set of constraints. Variables in X_i are carriers (ports, signals, variables) used in behavioural description B or variables derived from these behavioural carriers. (In the case of sequential circuits, variables can take different values in different time frames, thus, multiplied instances of carriers are used in the constraint sets.)

STEP 4. Generate the set of these path predicate constraint sets: $PP_\Sigma := \{PP_i\}$.

STEP 5. Generate the set of feasible paths and the corresponding path predicate constraint sets. Constraint solver can be used for this. Denote P_f the set of feasible paths, PP_f the set of corresponding path predicates.
$P_i \in P_\Sigma$, $P_i \in P_f$ if and only if there exists at least one solution of PP_i.
$PP_i \in PP_\Sigma$, $PP_i \in PP_f$ if and only if $P_i \in P_f$.

Algorithm B: Constraint Set Generation Describing Components Test Sets

Given input data are as follows:

- Behavioural level description of the digital circuit denoted by B:
- Gate-level description denoted by G
- Bit width dependent description of the test sets of the design library components
  TestSetDescription(N_i, x_i) denoting the description of the test set of component called N_i. x_i is the bit width of the component.

STEP 1. Extract the library components used in the gate level description G. Let us denote the set of library components by $L = \{L_1, L_2, \ldots, L_e\}$. L_i is defined by the name of the component and the actual bit width: $L_i = (N_i; bw_i)$.

STEP 2. Identify the signals connected to the input ports of the components in gate level description G. The description of the components is completed by the name of connected signals:
$L_i = (N_i; bw_i) \Rightarrow L'_i = (N_i; bw_i; \{S_i\})$.

STEP 3. Generate constraint set – noted by $(X_i, C_i)^R$ – describing the test set of the used components.
$L'_i = (N_i; bw_i; S_i) \Rightarrow (X_i, C_i)$ where $X_i := \{S_i\}$,
$C_i :=$TestSetDescription(N_i, bw_i)
Thus $(X_i; C_i)^R := (\{S_i\};$TestSetDescription$(N_i, bw_i))$.

STEP 4. Identify the carriers corresponding to the gate level signals in the behavioural description comparing the behavioural level description B and gate level description G. The result will be couples of behavioural carriers (R_i) and register level signals (S_i): $R_i = corresponding(S_i)$. (The couples are described in the form of a function.)

STEP 5. Substitute the variables in the component testing constraint sets according to the function defined above:
$(X_i, C_i)^R := (\{S_i\}, \text{TestSetDescription}(N_i, bw_i)) \Rightarrow$
$(X_i, C_i)^B := (\{corresponding(S_i)\}, \text{TestSetDescription}(N_i, bw_i))$

The version of the algorithm described above is the most general in the sense that this variation of the algorithm can be applied in each design system. However, the majority of synthesis systems allow modifications radically simplifying the previously described general algorithm.

The usage of RTL description is not necessary since most of the synthesis systems are rule-based design systems driven by cost functions (e.g. the design process is not heuristic). The RTL level description of the digital circuit is used for the identification of the library components and for the definition of the bit-width of the used components. High confidence estimation can be made regarding this information considering the design rules and the behavioural specification of the designed circuit in the majority of synthesis systems.

In the declarative part of the behavioural description, the bit-width of the used carriers is defined. Generally these features remain unchanged during the high-level design, i.e. the bit-width of the components implementing the given carrier can be estimated based on the behavioural specification of the circuit.

The used components can be predicted easily in the case of arithmetical instructions occurring in the behavioural description as the mapping of these instructions is generally well-defined by design rules. Since the algorithm deals basically with the data path this estimation gives sufficient result for the prediction of the used components. (Note that the estimation of the structure of the controller part is a much more difficult task. Fortunately, this problem is out of the scope of this algorithm as the controller part is tested implicitly by the execution of each CFG path.)

Algorithm C: Constraint Sets Composition

Given input data are as follows:

- Component testing constraint sets (in terms of behavioural variables): $(X_i, C_i)^B$ is the constraint set for the component number i, $C_i = \{C_i^1, C_i^2, \ldots, C_i^t\}$, where $|C_i| = t$.
- Path predicate constraint sets: $PP_f = \{PP_1, PP_2, \ldots, PP_s\}$, where $|PP_f| = s$.

Suppose that there were m components in the circuit.

STEP 1. For loop ($j = 1$; $j <= s$; $j++$) do (for each path predicate constraint set)
STEP 2. For loop ($k = 1$; $k <= m$; $k++$) do (for each component)
STEP 3. For loop ($l = 1$; $l <= t; l++$) do (for each constraint in the test description set)
STEP 4. Generate final constraint set (denoted by PP_j^{kl}):
$PP_j^{kl} := \{PP_j, C_m^l\}$
STEP 5. Solve the constraint set by constraint solver. The values of the variables symbolizing the input ports of the circuit define the test vectors.
STEP 6. END loop / END loop / END loop

In test generation of a certain digital circuit the Algorithm A and Algorithm B are executed first. These steps can be executed simultaneously. The Algorithm C is executed after the end of the first two steps.

4.1.1. Completeness and Correctness of DCT

As it has been declared above, the primary goal of the DCT algorithm is to generate test vectors for all feasible CFG paths and the secondary aim is to force the previously generated test vectors to the input port of the used components.

The first question to be posed is whether or not the algorithm finds a test vector for a feasible path. The answer to the question is unfortunately negative. The path predicate constraint set describes the necessary conditions of the traversal of a particular CFG path. Since the DCT algorithm adds further constraints to the path predicate constraint set, it is possible that the constraint set can be satisfied before the modification and cannot be satisfied after.

Fortunately, correction of this problem is simple. The path predicate constraint set has to be solved by the constraint solver without any modification in addition to those described above. Note that this step is exactly the same as the task of the classic path testing algorithm. In this case, the DCT algorithm subsumes the path testing algorithm, e.g. the test vectors generated by the DCT will include the test vectors generated by path testing algorithm. In this way, the testing of each feasible path is ensured.

The reason why this step is not included in the original algorithm description is simple. In practice, all the component test vector description constraint sets are completed by an additional general constraint which can be satisfied by any value of used variables. Since this constraint does not restrict the solution of the original path predicate constraint set, the solution of this constraint set is the same as the solution of the path predicate

constraint set, i.e. the output is equal to the output of the path testing algorithm.

Completion of the secondary goal of the test generation is obvious. In each case when a constraint set is passed to the constraint solver it contains the path predicate constraint set describing the necessary requirement of the path traversal and a single additional constraint describing the occurrence of one of the previously generated test vectors on the input port of the corresponding component. Since both constraint sets to be merged are minimal in the sense that they do not contain unnecessary constraints, the resulting union of the constraint sets will be minimal, too, i.e. a solution will be found if the described conditions can be satisfied.

Here we have to note that, by using this feature of DCT, the algorithm can be extended. It is not necessary to use constraint sets strictly describing test sets of components for adding it to the path predicate constraint sets. If there are any further values which is preferred to occur on the input ports of components and the description of these values is possible by constraints, then this constraint set can be used instead of the component test description constraint sets. The DCT algorithm in this case will generate input vectors of the circuit thus forcing the preferred values on the input port of the corresponding component.

Many test generation algorithms use restrictions regarding the logical values occurring on the internal nodes of the tested circuit. If these restrictions can be described by constraints, then DCT algorithm can potentially be used to implement these test generation algorithms. In this way, for instance, interesting comparisons can be made between different algorithms.

4.2. Implementation of the DCT Algorithm in Constraint Environment

In recent years, the theory of the constraint satisfaction problem has radically improved and some efficient algorithms and systems have been developed [17] [18] . The constraint environment is found to be suitablez for the description of test generation problems [19] . The implementation of the DCT algorithm is also done in a constraint environment. Moreover, the common advantages of the constraint environment meant there was a special reason for this selection. In the DCT algorithm, entities from different levels of abstraction (e.g. bit variables and integer variables) have to be uniformly managed. The constraint environment proved itself to be flexible in this sense.

The phases of the DCT algorithm described above can be implemented in the constraint environment by the following steps:

- *Synthesis system dependent phase*
  - General component test set generation for the library elements

- *Target circuit dependent phase*
  - Constraint set development for path predicates
  - Constraint set development for component testing
  - Constraint set composition and solving

The test generation of the actual circuit is preceded by the synthesis system dependent installation. During this process, the scalable test set of the library components is generated. This process is described in detail in [22] .

The description of the development of the several constraint sets and solving algorithms will be presented in subsequent sections.

Prior to detailed description of different phases of the algorithm, the data flow diagram of the DCT algorithm is presented in *Fig. 4*, giving an overview of the algorithm. The elliptical boxes represent the data entities and the rectangular boxes the processes. The arrows show the connections between the processes and the data entities.

4.3. Constraint Set Development for Path Predicates

Test generation for behavioural VHDL models is a problem similar to the software test generation regarding the fact that the behavioural description of the circuit consists of sequential instructions similar to any programming language.

Testing the software by traversing all the branches in the program is a well-known idea in software testing. The *path testing* algorithm is based on this idea. In the path testing algorithm the Control Flow Graph (CFG) of the program is first extracted, the paths in the graph are enumerated, and then one input vector of the program is generated, one for each path. This input vector will cause the traversal of the corresponding path in the CFG during the circuit operation [20].

The path testing algorithm has already been applied to hardware testing and has been proven an efficient algorithm for the generation of design verification tests as mentioned in the literature survey [5] [21].

In the DCT algorithm the path testing algorithm is applied to develop the predicates of the CFG paths. These predicates are described in the form of constraints. The predicates of the unfeasible paths are eliminated by recognizing the contradictions in the constraint sets by the constraint solver.

A simple path testing example is shown in *Fig. 5*. At the top of the figure, the CFG of the behavioural description is shown. The description is at the bottom left part of the figure. There are two CFG paths in the model. On the right, the predicates of the traversal of the paths are presented. The arrows show the correspondence between the behavioural model and the constraints.

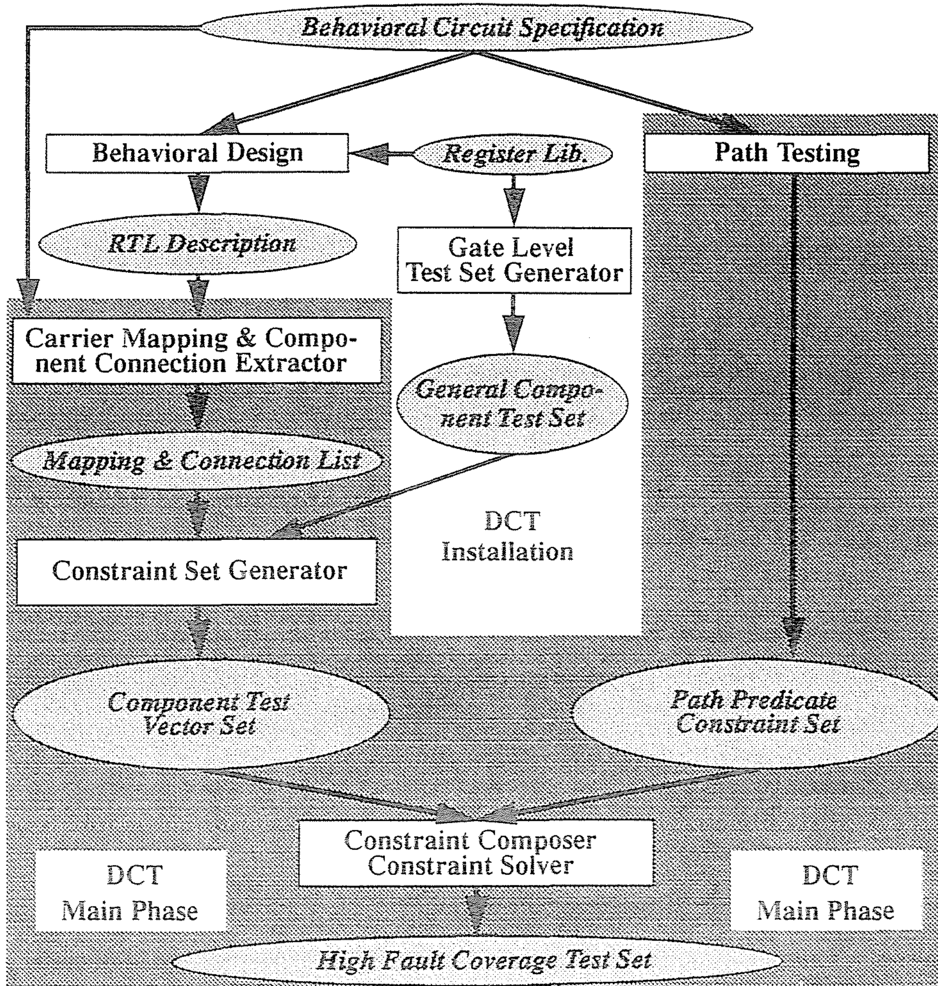

Fig. 4. Data Flow Diagram of the DCT algorithm

4.4. Constraint Set Composition and Solving

In this phase, the separately generated constraint sets are merged and passed to the constraint solver. This final phase of the test generation is found to be very sensitive with regard to the efficiency of the DCT algorithm. Several constraint set generation methods are tested during the development of the algorithm.

Which algorithm is most efficient depends on

- the structure of the circuit and
- the used constraint solving algorithm.

The following algorithm is found to be efficient in all circuit structures even if, in the case of some special circuit structures, there are more efficient methods.

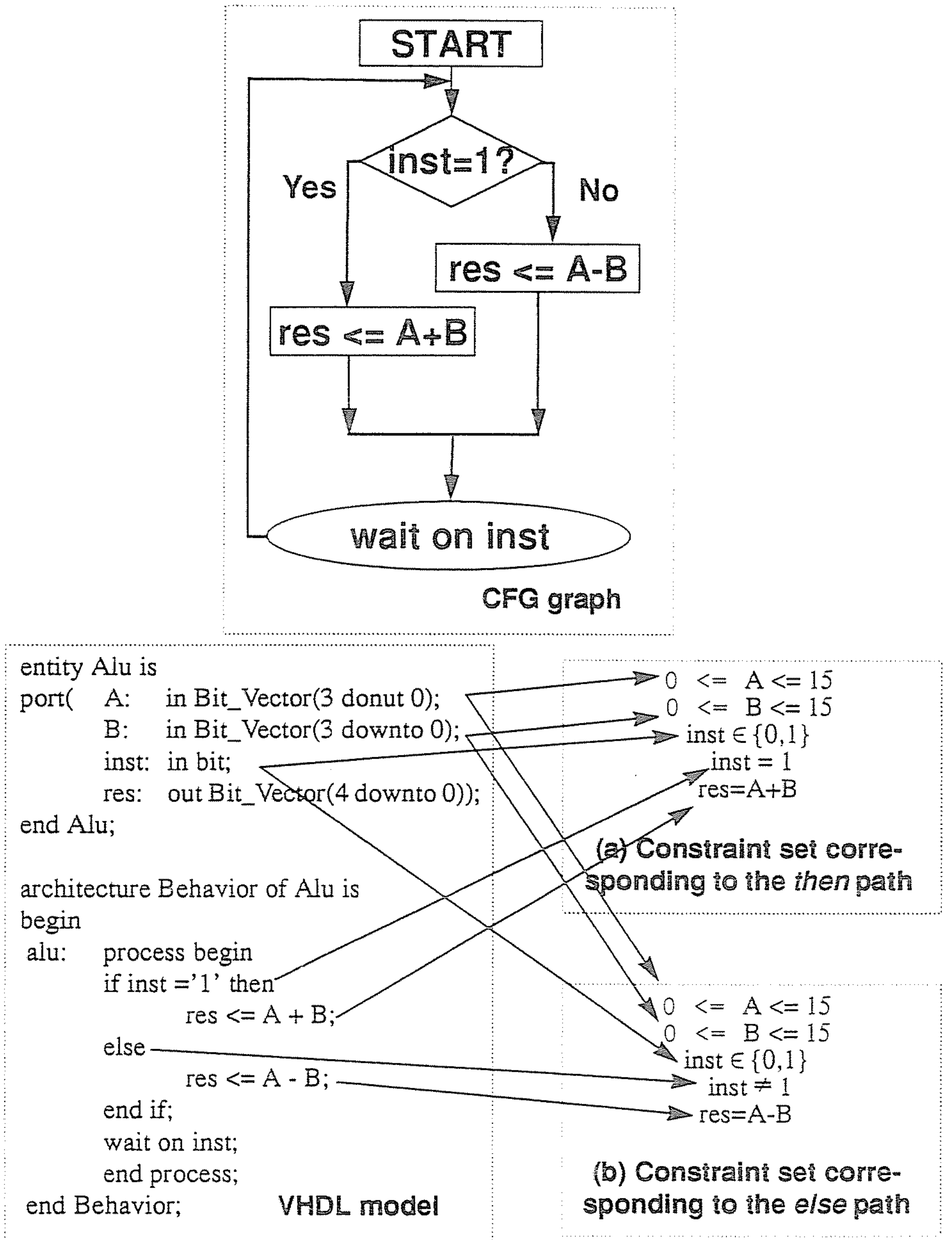

Fig. 5. Constraint sets generated by the path testing algorithm

The goal of DCT algorithm is to generate those input vectors which force the component test vectors generated in the initial phase of DCT installation to the input ports of arithmetical components.

The constraint generation algorithm implements directly the above idea in the constraint environment:

Firstly, it takes the constraint set describing one path predicate, then takes one component and one test vector from the corresponding component test set. The final constraint set is composed by the *path predicate constraint set* and a further *constraint forcing the chosen test vector to the input port of the arithmetical unit.*

The algorithm is illustrated by an example. In *Fig. 6* the constraint composition of the example introduced in *Fig. 5* is shown. The selected path is the 'then' path in the CFG and the actual port is 'A'. At the top of the figure there are two constraint sets describing the path predicate of the 'then' path and the test vectors for the components connected to the port 'A'.

From the path predicate constraint set as many constraint sets are derived as many test vectors exist in the actual test set. In each derived constraint set an extra constraint is inserted forcing one test vector to the ports of the adder.

In the derived constraint set there can be a contradiction which means that the CFG path with the given condition is not feasible. The constraint solver discovers this situation. As the test generation for this circuit under the given conditions is not possible, the constraint set will be eliminated.

If there is no contradiction, the constraint solver generates a solution for the constraint set. The value of the constraint variables representing the input ports of the circuit defines the searched test vector.

4.5. Testability of the Bit-Sliced Design

The synthesis library components – as already mentioned – are bit-sliced circuits. However, this class of circuits suffers from some testability problems. The core of this problem is that all sub-circuit components and connections in the bit-sliced design are uniform. The cascade inputs of the very first component in a chain of cascaded bit-sliced components are unused and they are inactivated by connecting them to a constant voltage (ground or power supply) depending on the actual logical function of the input. Because of this input bit of the circuit is uncontrollable, this – still redundant – part of the circuit remains naturally untestable. Fortunately, such untestable parts form less than 10% of the entire circuit in practice.

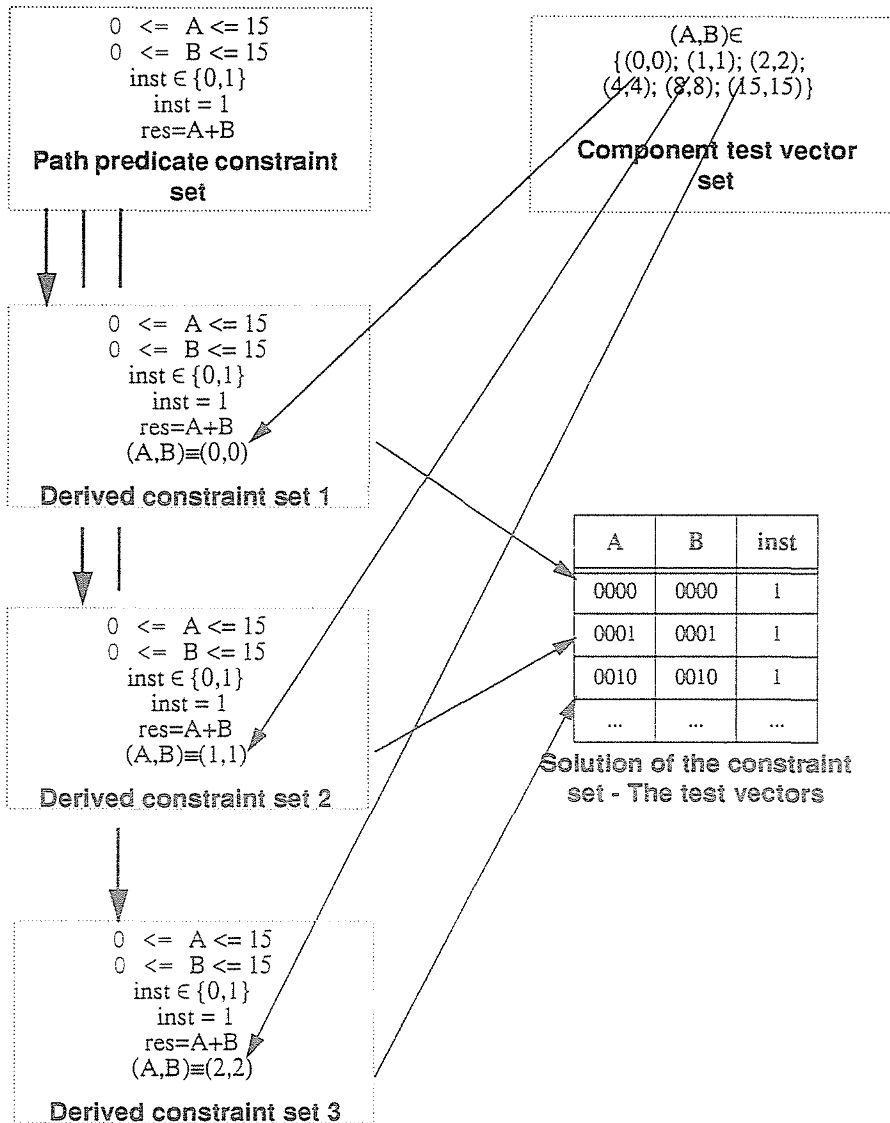

Fig. 6. Constraint set composition and test vector generation by the DCT algorithm

5. Results

The efficiency of DCT algorithm is evaluated by test pattern generation experiments while the analytical evaluation of the computational complexity

of the algorithm is a rather difficult and time-consuming process.

The correct experimental evaluation of the efficiency requires the comparison of the new algorithm to a gate-level test generation algorithm. In our case, the well-known PODEM algorithm [16] is selected as reference. This algorithm is considered one of the most efficient gate level test generation algorithms.

In the efficiency analysis both the behavioural and gate-level circuit models of the same circuit are used. The test generation experiments had to be preceded by a benchmark circuit selection process. Since no standard benchmark set fulfilling the above requirement is publicly available a special, self-developed benchmark circuit set is used for our experiments.

As a part of the efficiency evaluation process the experiments' aim is to determine the dependency of the *computation complexity* of DCT algorithm from the *size of the circuit.*

For this purpose two basic classes of benchmark circuits are developed representing different types of arithmetic intensive digital circuits. Each circuit belonging to the same class has the same structure and same function but the input bit width of the circuits is different. These circuits served as a test bed for estimating the dependency of the computation complexity of the test generation and the size of the circuit.

The ALU 4, ALU 8, and ALU 16 are arithmetical logical units with four instructions: *add, subtract, reset, compare.* These benchmarks represent the circuits with components connected parallel. In these circuits several arithmetical units are connected parallel having the same input and output port. The arithmetical units used in these circuits are a comparator, a subtractor and an adder.

The SERIAL 4 and SERIAL 8 circuits have four inputs called: A,B,C, D. The output value of the circuits is equal to AC+AD+BC+BD. These benchmarks represent the circuits with arithmetical units which are connected sequentially. Two adders and a multiplier are used in these circuits.

Basic parameters of benchmark circuits can be seen in *Table 1.* In this table the number of input vectors, number of design library components used in the RTL level design and number of gates in the gate level design are listed. Remember that the number of input vectors is equal to the number of test vectors in the case of exhaustive testing.

5.1. Results of Test Pattern Generation Experiments

The results of the test generation are presented in *Table 2* and *Table 3.* The test generation parameters of DCT and PODEM are shown in a common table in order to facilitate comparison. The goal of the test generation by PODEM algorithm was, in each case, the generation of a complete test set. Thus, the fault coverage belonging to PODEM algorithm (except for

Table 1. Benchmark circuits – series

	ALU 4	**ALU 8**	**ALU 16**	**SERIAL 4**	**SERIAL 8**
No. of input vectors	2^{10}	2^{18}	2^{34}	2^{16}	2^{32}
No. of library components	3	3	3	3	3
No. of gates	212	392	752	421	1373

cases when the predefined time limit of the generation is exceeded) is the *proportion of potentially testable faults*. It has to be outlined (especially in the case of real circuits) that the proportion of non-testable faults is often very high. The reason for this is the usage of bit-sliced components where many input ports of the components are not controllable, i.e. the component is not completely testable.

Below the listed parameters of the test generation process are shown:

- *Test generation time*: The CPU time of the test vector generation. (For the experiments SUN SPARC station 20 with 64M main memory and SunOS 5.5 is used.)
- *Number of test vectors*: The number of generated test vectors by the current algorithm.
- *Fault coverage*: Gate-level fault coverage of the generated test set, i.e. the proportion of the covered stuck-at faults and the total number of stuck-at faults (including the non-testable faults).

Comparison of the *fault coverage* of the DCT and PODEM algorithms shows a very favorable picture. *The fault coverage of the test sets generated by DCT approached and sometimes even reached the fault coverage of the PODEM-generated test sets.* This feature was independent of the proportion of non-testable faults. (It is demonstrated in SERIAL 4 where the proportion of non-testable faults was quite high (about 20%) and in the case of ALU_16 where almost all the faults were testable.) This is an important feature which allows the application of DCT algorithm in manufacturing testing.

The increase of the test generation time can be estimated based on the experiments by the ALU and SERIAL circuits. The DCT algorithm showed an exponential nature as expected but the growth of the test generation was significantly lower than the increase of the test generation time of the PODEM algorithm. This advantage is seen in the case of the synthesis example circuits when the test generation time of the DCT algorithm was much lower than the test generation time of PODEM.

The DCT algorithm is found to be more efficient in parallel target circuit structure, i.e. when the circuit components are connected parallel.

Table 2. Test generation for ALU 4, ALU 8, and ALU 16 by DCT and PODEM[a]

Benchmark circuit	**ALU 4**	**ALU 8**	**ALU 16**
Number of test vectors	160	352	736
Test generation time	127.07	277.88	679.5
Fault coverage (%)	90.81%	95.11%	97.48%
PODEM algorithm			
Number of test vectors	54	102	197
Test generation time	9.52	82.38	more than 20507[b]
Fault coverage (%)	91.45%	95.45%	97.65%

a. The time is measured in seconds.
b. The test generation is interrupted as it exceeded the given time limit.

This can be seen by comparing the experiments with SERIAL and ALU circuits. The ALU circuits have a parallel structure. The fault coverage of the test sets of DCT algorithm in this case almost reached the fault coverage of the PODEM generated test sets. In sequentially structured SERIAL circuits, the fault coverage approached the fault coverage of the PODEM generated test set but did not reach it.

Table 3. Test generation for SERIAL 4 and SERIAL 8 by DCT and PODEM[a]

Benchmark circuit	**SERIAL 4**	**SERIAL 8**
Number of test vectors	64	142
Test generation time	46.33	94.61
Fault coverage (%)	64.68%	74.81%
PODEM algorithm		
Number of test vectors	63	86
Test generation time	6819	more than 21410[b]
Fault coverage (%)	80.54%	83.78%

a. The time is measured in seconds.
b. The test generation is interrupted as it exceeded the given time limit.

This was an expected result as in sequential circuit structure many different components are activated during the traversal of a certain CFG path.

The operation of components results in several new constraints regarding the processed data which is reflected in complex path predicate constraint sets. Because of this, the chance of the infeasibility of the path predicate constraint set completed by the additional constraint forcing a particular logical value to the input port of a component is considerably higher than in the case of parallel target circuit structure where the path predicate constraint set is in general much simpler.

In practice, the structure of the circuit is not homogeneous, i.e. both parallel and sequential connections of components occur in the same circuits. Because of this, the efficiency of DCT algorithm in general is expected to fall between the results seen above.

5.2. Conclusion

A new high-level constraint based test generation algorithm has been developed which can be an alternative way of generating test vectors not only for design verification testing but also for manufacturing testing.

Through Direct Component Testing, i.e. combining CFG path predicate constraint sets with constraint sets describing the effective test set of the components, a new possibility of using structural information for the behavioural level test generation is introduced. The described algorithm is an efficient method of information compaction which regards the structure of the designed circuit and extracted from the synthesis roles of the design system.

The introduced description formalism for test set representation of bit-sliced digital circuit – i.e. test set description in terms of the bit width of the digital circuit – is a new idea in the field of digital circuit testing.

Experimental results show a sufficiently high degree of gate-level fault coverage of the test sets generated by the new algorithm. The test generation time of the new algorithm is found to be significantly lower than the test generation time of the PODEM algorithm. The DCT algorithm is proven to be more efficient for circuits with components connected parallel based on the experimental result. Path predicate constraint sets for circuits with sequentially connected components result in strict conditions with regard to the path traversal. These strict conditions potentially frustrate the forcemeat of candidate test vectors to the component input ports, e.g. the improper controllability of sequentially connected circuit structures potentially inhibits the generation of a high fault coverage test set.

The developed special benchmarks parametrized by the bit-width of the circuits served as a perfect test bed for estimating the dependency of the computation complexity of the test generation and the size of the circuit.

References

[1] ABRAMOVICI, M. – BREUER, M. A. – FRIEDMAN, A. D.: Digital System Testing and Testable Design, Computer Science Press, New York, 1990, ISBN 0-7167-8179-4.
[2] RANGA VEMURI, – KUMAR, N. – VUTUKURU, R. – RAO, P. S. – PRAVEEN SINHA, NING REN, – PADDY MAMTORA, – RAM MANDAYAM, – RAM VEMURI, – JAJANTA ROY: An Integrated Multicomponent Synthesis Environment for MCMs, *IEEE Computer*, April 1993, Vol. 26, No. 4, pp. 62–74.
[3] SCHOEN, J. M.: Performance and Fault Modelling with VHDL, Englewood Cliffs, N.J.: Prentice Hall, 1992.
[4] BENYÓ, B. – VEMURI, R. – PATARICZA, A.: Algorithmic Versus Random Functional Test Generation, submitted paper to *Journal of Electrical and Electronic Testing.*
[5] VEMURI, R. – KALYANARAMAN, R.: Generation of Design Verification Test from Behavioural VHDL Programs Using Path Enumeration and Constraint Programming, *IEEE Transactions on VLSI Systems*, 1994.
[6] ARMSTRONG, J. R.: Hierarchical Test Generation: Where We Are, and Where We Should Be Going?, *IEEE*, pp. 434–439. 1993, ISBN 0-8186-4350-1/93.
[7] SHELDON B. AKERS: Test Generation Techniques, *IEEE Computer*, March 1980.
[8] LEVENDEL, Y. H. – MENON, P. R.: Test Generation Algorithms for Computer Hardware Description Languages, *IEEE Transactions on Computers*, Vol. C–31, No. 7, July 1982.
[9] BREUER, M. A. – FRIEDMAN, A. D.: Functional Level Primitives in Test Generation, *IEEE Trans. on Computers*, Vol. C–29, No. 3, pp. 223–225, March 1980.
[10] RAVISHANKAR KALYANARAMAN: Behavioural Test Generation in VHDL/WAVES Environment, M. Sc. Thesis, Dept. of Engineering and Computer Sciences, Univ. of Cincinnati, 1993.
[11] CHIANG, A. C. L. – MCCASKILL, R: Two New Approaches Simplify Testing of Microprocessors, *Electronics*, Vol. 49, No. 2, pp. 100–105, January 1976.
[12] MCCASKILL, – BOZORGUI-NESBAT, S.: Design for Autonomous Test, *IEEE Trans. on Computers*, Vol. C–30, No. 11, pp. 866–875, November 1981.
[13] THATTE, S. M. – ABRAHAM, J. A.: Test Generation for Microprocessors, *IEEE Trans. on Computers*, Vol. C–29, No. 6, pp. 429–441, June 1980.
[14] BRAHME, D. – ABRAHAM, A.: Functional Testing of Microprocessors, *IEEE Trans. on Computers*, Vol. C–33, No. 6, pp. 475–485, June 1984.
[15] HUMMER, H. D. – VEIT, H. – TOPFER, H.: Functional Test of Hardware Described from VHDL, *Proceedings of CHDL-91*, pp. 465–477, April 1991.
[16] USC Test Group: Test Generation System (TGS) User's Manual – Version 1.0, University of Southern California.
[17] JAFFAR, J. – MICHALOV, S.: Methodology and Implementation of CLP(R) System, *Proc of 4th ICLP*, MIT Press, May 1987, pp. 196–218.
[18] JAFFAR, J. – MICHALOV, S. – STUCKEY, P. J. – JAP, R. H. C.: The CLP(R) Language and System, *IBM Research Rep.* RC 16292 (#72336), Nov. 1990.
[19] TILLY, K.: A Comparative Study of Automatic Test Pattern Generation and Constraint Satisfaction Methods, *Technical Report Ser. Electrical Engineering*, Technical University of Budapest, June 20, 1994.
[20] INCE, D.: Software Testing, in J. McDermin (ed.): Software Engineer's Reference Book, Butterworth-Heinemann Ltd., 1991.
[21] LIN, T. S. – SU, S. Y. H.: VLSI Functional Test Pattern Generation – A Design and Implementation, *IEEE International Test Conference*, pp. 922–929, November 1985.
[22] BENYÓ, B.: Test Pattern Generation Based on High Level Hardware Description, CSc (PhD) thesis, Budapest, 1997.