Structural Verification in Minimal Time

M. Holzer, B. Knerr, and M. Rupp
Vienna University of Technology
Institute for Communications and RF Engineering
Gusshausstr. 25/389, 1040 Wien, Austria
{mholzer, bknerr, mrupp}@nt.tuwien.ac.at

Abstract— **During the design process of a complex system on chip most time is spent on the verification task. Structural verification is one of the primary strategies for testing. We present a method, where the structural testing effort is minimised. This is based on an algorithm, which identifies a set of linearly independent paths of a control flow graph together with a shortest path search. An example is given, where the effort for structural testing can be reduced by more than 40%.**

I. Introduction

With the high complexity of integrated circuits, especially in the wireless communication domain, verification has become one of the most complicated and time consuming tasks in the design process. Already up to 70% of development time is spent on verification and testing [1], [2]. Verification is done on many different design levels starting on algorithmic level down to architecture level and finally ends after production [3], [4]. CoWare [5] gives an example, where a simulation of four seconds in real-time takes five minutes simulation time at the algorithmic level. The simulation time on gate level alone would take 1.25 years. The high complexity imposed by a modern System-on-Chip increases the number of test cases and therefore simulation time.

The major approaches for verification are structural (white box testing) and functional verification (black box testing). Functional testing compares test program behavior against its specification, whereas structural verification analysis the internal structure of a program for errors [6]. This is often expressed with different kinds of metrics like test coverage metrics, fraction of code exercised by verification, statement coverage, branch coverage, or path coverage.

The structural testing criteria [7] requires a set of test cases, where all decisions (if, for and while statements) of a control flow graph (CFG) are tested independently. Usually, there exists more than one solution for such a set of test cases. We present an algorithm, which allows for deriving this set of test cases, while a minimal verification time effort is accomplished.

The rest of the paper is organised as follows. Section II gives an overview of the related work in the field of structural testing. Further in Section III some definitions are given for the graph representation. The minimisation problem and an algorithm for the optimisation of the verification time is presented in Section IV. In Section V the algorithm is applied to an example. Finally some conclusions are drawn in Section VI.

This work has been funded by the Christian Doppler Laboratory for Design Methodology of Signal Processing Algorithms.

II. Related Work

McCabe [7] defined the cyclomatic complexity of a software function and emphasised the importance of structural verification for efficient white box testing. The designer selects a so called primary path, which is used to derive other paths. The primary path and the derived paths build up a set of test cases. This approach constructs test cases tailored to the functionality of a software function, whereas our approach minimises run time of test cases. Agrawal [8] presented a method for computing all bases of an CFG. In this work a reduction of test cases is achieved by selecting the leaves of the post- or predominator tree for the test case. The aim is to provide solutions for situations, which do not need a full test coverage. Structural testing based on minimum kernels is presented by Dubrova [9]. In this work, similar to the work Agrawal, also a combination of post- and predominator trees is used to identify test cases. Poole's algorithm [10] is used for deriving an arbitrary set of linear independent paths and Makaruk et al. [11] derives paths from descriptions in the unified modelling language (UML). Not every possible path of a control flow is feasible, because of interdependent conditions, so that removing so called non feasible paths [12], [13] allows for reducing the verification effort.

III. Graph Prerequisites

The structural verification of a function is based on the analysis of graph structures. The following list enumerates the basic definitions for graphs that are needed in the remaining part.

Definition 1: Graph
A **graph** $G(\mathcal{V}, \mathcal{E})$ is defined as an ordered pair of a set $\mathcal{V} = \{v_1, v_2, ..., v_{|\mathcal{V}|}\}$ of vertices and a set $\mathcal{E} = \{e_1, e_2, ..., e_{|\mathcal{E}|}\}$ of edges.

Definition 2: Edge
The set of edges $\mathcal{E}$ is defined as a 2-tuple of vertices $\mathcal{E} = \{(v, w) \,|\, v, w \in \mathcal{V}\}$

Definition 3: begin/end
The operation **beg** returns the source vertex, and the operation **end** returns the sink vertex of an edge e as follows: $\forall e = (v, w) \in \mathcal{E} : \mathrm{beg}(e) = v, \mathrm{end}(e) = w$.

Definition 4: indegree
The operation **indegree**(v) returns the number of incoming edges to the vertex $v \in \mathcal{V}$.

Definition 5: outdegree
The operation **outdegree**(v) returns the number of outgoing edges from the vertex $v \in \mathcal{V}$.

Definition 6: path
A **path** $\mathcal{P}$ from a vertex v_0 to a vertex v_n in a directed graph is a sequence of vertices v_0, v_1, v_2, ..., v_n that satisfies: $\forall i,\ i = 0 \dots n-1\, \exists\, (v_i, v_{i+1}) \in \mathcal{E}$. The vertex v_0 is the initial vertex and v_n is the terminal vertex of the path. Equivalently a **path** from a vertex v_0 to vertex v_n can be described by a sequence of edges $e_1, e_2, \dots, e_n$. A **simple path** $\mathcal{P}_s$ additionally fulfills the condition: $\forall v_i, v_j \in \mathcal{P}, i \neq j : v_i \neq v_j$.

Definition 7: Control Flow Graph
A **control flow graph (CFG)** is a directed graph $G(\mathcal{V}, \mathcal{E}, root, exit)$ with the set $\mathcal{V}$ of vertices and the set $\mathcal{E}$ of edges. The vertices $root$ and $exit$ are special vertices because $root$ does not have any incoming and $exit$ does not have any outgoing edge ($\mathrm{indegree}(root) = \mathrm{outdegree}(exit) = 0$).
In general an algorithm within a function, which is written in form of sequential code, can be decomposed into a CFG. In a CFG representation the interconnected vertices are called basic blocks (BB). Each basic block contains a sequence of data operations ended by a control flow statement as its last instruction. This means that the operations inside a BB are completely data oriented and can be represented as data flow graph.

IV. Minimisation of Verification Time

Each path of a CFG can be associated to a vector, where each element of the vector is associated to a distinct edge of the CFG and its value represents the number of times that the edge is traversed.

Example 1: Consider the graph in Fig. 1 with seven vertices (basic blocks) and eight edges. The vertex $BB1$ is the $root$ and the vertex $BB7$ is the $exit$ vertex of the CFG.

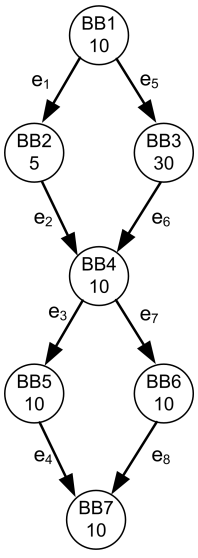

Fig. 1. Simple example of a CFG. The basic blocks are annotated with the corresponding cycle count (CC) that is needed to execute the internal DFG.

The path consisting of the edges e_1, e_2, e_3, and e_4 can be represented by the vector $(1,1,1,1,0,0,0,0)^T$. All the possible paths of the shown CFG are given in their vector notation, $\vec{p}_1 = (1,1,1,1,0,0,0,0)^T$, $\vec{p}_2 = (0,0,0,0,1,1,1,1)^T$, $\vec{p}_3 = (1,1,0,0,0,0,1,1)^T$, $\vec{p}_4 = (0,0,1,1,1,1,0,0)^T$.

Paths are combined by adding or subtracting the corresponding vector representations. For example the path $\vec{p}_4$ can be linearly expressed by the $\vec{p}_4 = \vec{p}_1 + \vec{p}_2 - \vec{p}_3$.

Not every permutation of vector entries corresponds to a valid path through the CFG, which means that the set of vectors of valid paths for the CFG is a subset of the set of all possible vectors with the dimension $|\mathcal{E}|$. Since linear combinations of vectors from this subset exist, that do not create valid paths through the CFG, this subset does not form a subspace.

In the following we use the term *basis of paths* or short *basis* as a maximal set of linearly independent paths of the span of the possible paths. Linear independence means each path vector in the basis set cannot be formed as a combination of other paths of the basis. Therefore, any path through the control flow graph can be formed as a combination of paths in the basis. The set $\{\vec{p}_1, \vec{p}_2\}$ is not a basis, because there is no possibility to construct the path vector $\vec{p}_4$. Whereas the set $\{\vec{p}_1, \vec{p}_2, \vec{p}_3\}$ defines a basis. Such a basis is not unique, thus a CFG can have more than one basis.

The structured testing criteria [7] requires the verification of all paths of a basis in order to test all decisions of the CFG independently. Hence, the number of vectors of the basis defines the number of needed test cases to achieve this structural testing criteria. McCabe's measure [7] *cyclomatic complexity* $V(G)$, which has its origin from the *cyclomatic number* defined in graph theory [14], is equal to the number of paths in the basis (dimension of the basis). It can be computed with the expression

$$V(G) = |\mathcal{E}| - |\mathcal{V}| + 2. \tag{1}$$

According to (1) a basis for the CFG of Example 1 (Fig. 1) has three entries (eight edges and seven vertices). In this case four different bases $\mathcal{B}_i$ can be identified

$$\begin{aligned} &\mathcal{B}_1 = \{\vec{p}_1, \vec{p}_2, \vec{p}_3\}, \mathcal{B}_2 = \{\vec{p}_1, \vec{p}_2, \vec{p}_4\}, \\ &\mathcal{B}_3 = \{\vec{p}_1, \vec{p}_3, \vec{p}_4\}, \mathcal{B}_4 = \{\vec{p}_2, \vec{p}_3, \vec{p}_4\}. \end{aligned} \tag{2}$$

The cycle count (CC) annotation of the basic blocks in Fig. 1 allows to determine the overall verification time of a distinct basis. For example the execution time of $\mathcal{B}_3$ is higher (CC=185, BB_3 is used twice) than the one of $\mathcal{B}_1$ (CC=160, BB_3 is used only once).

Therefore, the aim for reducing verification time is to find a basis $\mathcal{B}_i$ out of the set of all M possible bases $\mathbf{B} = (\mathcal{B}_1, \dots, \mathcal{B}_M)$ that has minimal overall execution time

$$T_{min} = \min_{\mathcal{B}_i \in \mathbf{B}} \{T_v(\mathcal{B}_i)\}. \tag{3}$$

The time effort T_v for testing all paths of the basis is the sum of the execution time for all paths of the basis

$$T_v(\mathcal{B}_i) = \sum_{\vec{p}_j \in \mathcal{B}_i} t(\vec{p}_j). \quad (4)$$

For each basic block BB an execution time $\mathrm{CC}(BB)$ is assumed. This can be accomplished by measuring the execution time by simulation or by static estimation of the execution time [13]. The execution time of one path $\vec{p}_j$ can be expressed with

$$t(\vec{p}_j) = \vec{p}_j \vec{C}_{beg} + \mathrm{CC}(exit). \quad (5)$$

Here the vector $\vec{C}_{beg}$ represents the execution times of the basic blocks of the CFG,

$$\vec{C}_{beg} = \begin{pmatrix} \mathrm{CC}(\mathrm{beg}(e_1)) \\ \mathrm{CC}(\mathrm{beg}(e_2)) \\ \mathrm{CC}(\mathrm{beg}(e_3)) \\ \vdots \\ \mathrm{CC}(\mathrm{beg}(e_{|\mathcal{E}|})) \end{pmatrix}. \quad (6)$$

The entries are built by assigning each entry of the vector the execution time of the corresponding basic block.

An algorithm for the generation of a basis has been presented by Poole [10] and is shown in Listing 1 for convenience. This algorithm is based on a depth-first search through the CFG. Each time a vertex is visited for the first time, one of the outgoing edges is marked arbitrarily as default edge (line 6). After that the recursion of the function follows the default edge and afterwards it follows the other outgoing edges of the current vertex (line 8-9). If the vertex has been already visited, only the default path is taken (line 12). This causes that the default edges build the main parts of the basis vectors. If the exit vertex is reached than the currently followed path is a path of the basis (line 2).

```
1  FindBasis(vertex)
2    if (vertex == EXIT) then store path
3    else if (vertex not VISITED)
4    {
5      mark vertex as VISITED
6      label default edge
7      FindBasis(end(defEdge(vertex)))
8      for all other outgoing edges
9        FindBasis(end(edge))
10   }
11   else
12     FindBasis(dest(defEdge(vertex)))
```

Listing 1. Poole's algorithm for the identification of a basis of a CFG.

By storing the paths to the exit vertex, so that each time an already visited vertex is found, the already found default path to the exit can be used. This yields a complexity of $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$. In this algorithm the resulting basis depends on the arbitrarily chosen edges for the default paths. In Fig. 2 the four possible default edge selections of Example 1 (marked with thicker edges) are depicted, resulting in the four bases given in (2).

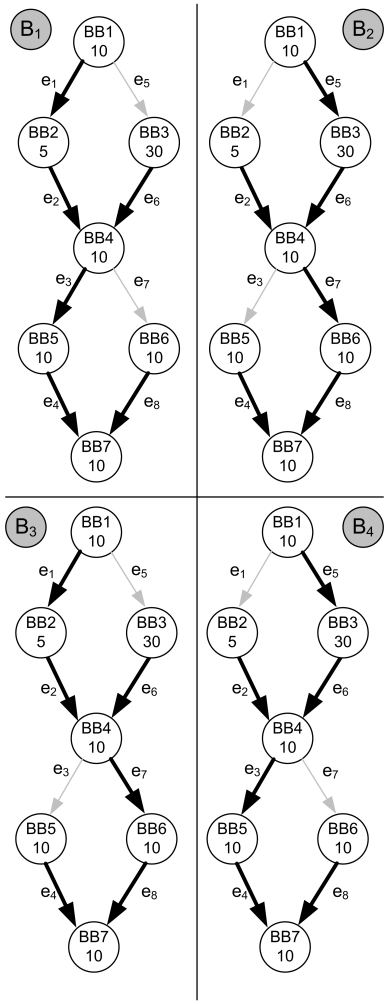

Fig. 2. Control flow graph with four different selections of the default edges indicated by bold edges.

The minimisation of (3) is achieved by first calculating the shortest path (each vertex is weighted with its run time) from each vertex of the CFG to the exit vertex of the CFG. In the case of a CFG the shortest path search has only a complexity of $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$, since there is exactly one *start* and one *exit* node.

With this list each vertex' edge is annotated that points to the next vertex that is nearest to the exit vertex (the first vertex of the shortest path successor list). Afterwards Poole's algorithm is performed, where, instead of the arbitrary default edge selection, the edge leading to the shortest path will be chosen. In Listing 1 line 6 has to be exchanged to label the shortest path edge as default edge.

Theorem 1: Let the graph G be a control flow graph. A basis $\mathcal{B}_1$ which yields a minimum verification time as described in (3) is achieved by:

- Firstly, the application of the shortest path algorithm which marks the edge of each vertex that points via the shortest path to the $exit$ node.
- Secondly, the application of Poole's algorithm (Listing 1)

on the graph G, where the default edge is chosen to be the marked shortest path edge.

Proof: We assume that a basis $\mathcal{B}_2$ exists, which is different from $\mathcal{B}_1$, with an overall execution time that is smaller than the one of $\mathcal{B}_1$. This implies that at least one of the chosen default edges is different compared to the default edges of the basis $\mathcal{B}_1$. This means that a default edge has been chosen that points to a basic block with a higher execution time. Therefore, the overall execution of the basis $\mathcal{B}_2$ will be higher than the one of $\mathcal{B}_1$ which contradicts the assumption. ■

V. Example

The presented minimisation algorithm will be demonstrated on one function of an *MPEG* algorithm out of the embedded systems benchmark library MediaBench [15]. The function under test (*predcase2*) has 47 vertices and 74 edges that results in a cyclomatic complexity of 27, which determines the number of paths that establish the basis. The CFG of this example has 68 possible paths with their corresponding vectors $\vec{p}_1, ..., \vec{p}_{68}$. The worst case scenario is achieved by finding a maximum of the overall verification time. In order to identify this worst case scenario regarding testing time a longest path search has been applied to the function and the edges, which points to longest path to the exit vertex have been marked and are used to determine the overall testing time.

$\mathcal{B}_{\min}$	$t(\vec{p}_j)$	$\mathcal{B}_{\max}$	$t(\vec{p}_j)$
$\vec{p}_1$	6	$\vec{p}_2$	14
$\vec{p}_5$	14	$\vec{p}_3$	22
$\vec{p}_7$	15	$\vec{p}_4$	23
$\vec{p}_{10}$	16	$\vec{p}_6$	24
$\vec{p}_{11}$	15	$\vec{p}_8$	23
$\vec{p}_{18}$	16	$\vec{p}_9$	24
$\vec{p}_{20}$	15	$\vec{p}_{12}$	23
$\vec{p}_{21}$	16	$\vec{p}_{13}$	24
$\vec{p}_{27}$	15	$\vec{p}_{15}$	23
$\vec{p}_{28}$	16	$\vec{p}_{19}$	24
$\vec{p}_{29}$	14	$\vec{p}_{22}$	22
$\vec{p}_{40}$	15	$\vec{p}_{23}$	23
$\vec{p}_{41}$	14	$\vec{p}_{30}$	22
$\vec{p}_{43}$	15	$\vec{p}_{31}$	23
$\vec{p}_{44}$	16	$\vec{p}_{32}$	24
$\vec{p}_{46}$	15	$\vec{p}_{33}$	23
$\vec{p}_{50}$	16	$\vec{p}_{34}$	24
$\vec{p}_{51}$	15	$\vec{p}_{35}$	23
$\vec{p}_{52}$	16	$\vec{p}_{36}$	24
$\vec{p}_{56}$	15	$\vec{p}_{37}$	23
$\vec{p}_{57}$	16	$\vec{p}_{38}$	24
$\vec{p}_{59}$	14	$\vec{p}_{42}$	22
$\vec{p}_{60}$	15	$\vec{p}_{53}$	23
$\vec{p}_{61}$	7	$\vec{p}_{54}$	7
$\vec{p}_{62}$	13	$\vec{p}_{62}$	13
$\vec{p}_{63}$	14	$\vec{p}_{63}$	14
$\vec{p}_{64}$	15	$\vec{p}_{64}$	15
	389		573

TABLE I

Minimal and maximal timing for testing of the *predcase2* function.

Table I reports on the number of basis paths found in the control flow graph and its needed cycle count. The set $\mathcal{B}_{\min}$ establishes the basis, which achieves a minimum verification time ($T_{\min} = 389$) and the set $\mathcal{B}_{\max}$ builds the maximal verification time ($T_{\max} = 573$) for this functions. Note that the set of paths, which builds a minimal and the maximal verification time has three paths in common ($\vec{p}_{62}$, $\vec{p}_{63}$, and $\vec{p}_{64}$).

The difference between maximum and minimum overall verification time for this function exhibits ($\frac{T_{\max}}{T_{\min}} = 1.47$) 47%, which highlights the tremendous importance of a carefully chosen test bench.

VI. Conclusions

The need for reduced verification both in regression test and in post production testing of SoC has increased with the complexity of the manufactured devices. Especially structured testing is a standard technique for the verification of an SoC. We present a method for the selection of a basis that consumes least time for verification by applying Poole's algorithm together with a shortest path algorithm. The difference between a best case and worst case scenario is shown on the example of a function of an *MPEG* algorithm, which demonstrates the potential of a time minimised selection of test paths.

References

[1] A. Hoffmann, T. Kogel, and H. Meyr, "A Framework for Fast Hardware-Software Co-simulation," in *Design, Automation and Test in Europe DATE'01*, Munich, 2001, pp. 760–765.

[2] B. Bailey, "The Waking of the Sleeping Giant – Verification," April 2002.

[3] P. Belanović, B. Knerr, M. Holzer, and M. Rupp, "A fully automated environment for verification on virtual prototypes," *EURASIP Journal on Applied Signal Processing*, vol. 2006, 2006.

[4] P. Belanović, B. Knerr, M. Holzer, G. Sauzon, and M. Rupp, "A consistent design methodology for wireless embedded systems," *EURASIP Journal on Applied Signal Processing*, vol. 2005, no. 16, pp. 2598–2612, 2005.

[5] CoWare, Inc., "SoC platform-based design using ConvergenSC/SystemC," July 2002, http://www.coware.com.

[6] B. Beizer, *Software Testing Techniques for Functional Testing of Software and Systems*, Wiley, New York, 1990.

[7] McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308–320, 1976.

[8] H. Agrawal, "Dominators, super blocks, and program coverage," in *Annual Symposium on Principles of Programming Languages*, 1994, pp. 25–34.

[9] Elena Dubrova, "Structural Testing Based on Minimum Kernels," in *Design and Test in Europe*, March 2005, pp. 1168–1171.

[10] J. Poole, "A Method to Determine a Basis Set of Paths to Perform Program Testing," U.S. Department of Commerce/National Institute of Standards and Technology, November 1995.

[11] Hanna Makaruk, Robert Owczarek, and Nikita Sakhanenko, "Systematic method for path-complete white box testing," 2005.

[12] C. Pauli, M. L. Nivet, and J. F. Santucci, "Use of constraint solving in order to generate test vectors for behaviroal validation," in *High-Level Design Validation and Test Workshop*, November 2000, pp. 15–20.

[13] M. Holzer and M. Rupp, "Static Estimation of the Execution Time for Hardware Accelerators in System-on-Chips," in *Symposium on System-on-Chip*, Tampere, November 2005, pp. 62–65.

[14] C. Berge, *Graphs and Hypergraphs*, North-Holland Publishing Company, 1973.

[15] C. Lee, M. Potkonjak, and William H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communication systems," in *International Symposium on Microarchitecture*, 1997, pp. 330–335.