Code Coverage Analysis for Concurrent Programming Languages Using High-Level Decision Diagrams

Maksim Jenihhin, Jaan Raik, Anton Chepurov, Uljana Reinsalu, Raimund Ubar

Code Coverage Analysis for Concurrent Programming Languages Using High-Level Decision Diagrams

Maksim Jenihhin, Jaan Raik, Anton Chepurov, Uljana Reinsalu, Raimund Ubar
Department of Computer Engineering, Tallinn University of Technology
E-mail: { maksim|jaan|anchep|uljana|raiub }@pld.ttu.ee

Abstract

The paper presents using high-level decision diagram (HLDD) as a suitable graph model for code coverage analysis in concurrent programming languages. The authors show that HLDD models are scalable and compact models for realistic problems can be automatically obtained. At the same time they allow covering paths of non-blocking assignments, which conventional code coverage metrics designed for procedural programming languages are unable to handle. The paper proposes optimal minimization rules to be used in HLDD model synthesis for code coverage analysis.

1. Introduction

In order to verify the correctness of a design, different test cases are generated. Due to the fact that it is impractical to verify exhaustively all possible inputs and states of a design, the confidence level regarding the quality of the design must be quantified to control the verification effort. The fundamental question is: How do I know if I have verified or simulated enough? Verification coverage is a measure of confidence and it is expressed as a percentage of items verified out of all possible items. Different definitions of items give rise to different coverage measures or coverage metrics.

Over the years, a large variety of code coverage metrics have been proposed, including statement coverage, block coverage, path coverage, branch coverage, expression coverage, transition coverage, sequence coverage, toggle coverage, condition coverage etc [1],[2]. Structural coverage has a long history in software testing and only with the emergence of hardware description languages has it been applied to hardware verification and test.

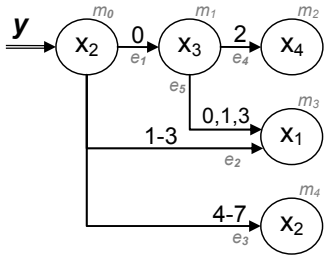
Note, however that all the above-mentioned coverage metrics are designed to support sequential procedural languages. However, concurrency inherent in e.g. hardware description languages and concurrent programming languages is not supported. In this paper we present using high-level decision diagram (HLDD) as a graph model for code coverage analysis in concurrent programming languages. We show that HLDD models are scalable and compact models for realistic problems can be automatically obtained. At the same time they allow covering paths of non-blocking assignments, which conventional code coverage metrics designed for procedural programming languages are unable to handle.

The paper proposes optimal minimization rules to be used in HLDD model synthesis for code coverage analysis. Experiments using concurrent hardware description language VHDL on ITC99 benchmarks show that up to 14 % increase in coverage accuracy can be achieved by the proposed methodology compared to traditional code coverage metrics.

2. High-Level Decision Diagrams

2.1 HLDD model definition

A *High-Level Decision Diagram* (HLDD) is a graph representation of a discrete function. A discrete function $y = f(x)$, where $y = (y_1, \ldots, y_n)$ and $x = (x_1, \ldots, x_m)$ are vectors is defined on $X = X_1 \times \ldots \times X_m$ with values $y \in Y = Y_1 \times \ldots \times Y_n$, and both, the domain X and the range Y are finite sets of values. The values of variables may be Boolean, Boolean vectors, integers. Fig. 1 presents an example of a graphical interpretation of a HLDD.



G_y=(M,E,Z,Γ),

M={m_0, m_1, m_2, m_3, m_4};

E={e_1, e_2, e_3, e_4, e_5}, e_1=(m_0, m_1), e_2=(m_0, m_3), e_3=(m_0, m_4), e_4=(m_1, m_2), e_5=(m_1, m_3);

Z(m_0)=Z(m_4)=x_2, Z(m_1)=x_3, Z(m_2)=x_4, Z(m_3)=x_1;

Γ(e_1)={0}, Γ(e_2)={1,2,3}, Γ(e_3)={4,5,6,7}, Γ(e_4)={2}, Γ(e_5)={0,1,3}.

Fig.1. A high-level decision diagram representing a function *y= f(x1,x2,x3,x4)*

Definition 1**:** A high-level decision diagram is a directed non-cyclic labelled graph that can be defined as a quadruple $G=(M,E,Z,\Gamma)$, where M is a finite set of vertices (referred to as *nodes*), E is a finite set of *edges*, Z is a function which defines the *variables labeling the nodes*, and Γ is a function on E.

The function $Z(m_i)$ returns the variable x_k, which is labeling node m_i. Each node of a HLDD is labeled by a variable. In special cases, nodes can be labeled by constants or algebraic expressions. An edge $e \in E$ of a HLDD is an ordered pair $e=(m_{pc},m_{sc}) \in E^2$, where E^2 is the set of all the possible ordered pairs in set E. Graphical interpretation of e is an edge leading from node m_{pc} to node m_{sc}.

It is said that m_{pc} is a *predecessor node* of m_{sc}, and m_{sc} is a *successor node* of the node m_{pc}, respectively. Γ is a function on E representing the activating conditions of the edges for the simulating procedures. The value of $\Gamma(e)$ is a subset of the domain X_k of the variable x_k, where $e=(m_i,m_j)$ and $Z(m_i)=x_k$. It is required that $Pm_i=\{\ \Gamma(e) \mid e=(m_i,m_j) \in E\}$ is a partition of the set X_k.

Fig. 1 presents a HLDD for a discrete function $y=f(x_1,x_2,x_3,x_4)$. HLDD has only one starting node (*root node*) m_0, for which there are no preceding nodes. The nodes that have no successor nodes are referred to as *terminal nodes* $M^{term} \in M$ (nodes m_2, m_3 and m_4).

Design representation by high-level decision diagrams, in general case, is a system of HLDDs rather than a single HLDD. In fact, for each program variable a separate HLDD is generated. During the simulation in HLDD systems, the values of some variables labeling the nodes of a HLDD are calculated by other HLDDs of the system.

3. Concurrent programming using non-blocking signal assignments

In this paper we consider code coverage measurement for non-blocking signal assignments on the example of a concurrent hardware description language VHDL. VHDL uses a discrete event system to model time and deal with concurrency, and so is very flexible. The discrete event model is very general, but as a result, somewhat difficult to analyze [7].

VHDL uses, both, blocking and non-blocking assignments. All assignments to signals (with '<=') are non blocking (i.e. they happen some (delta) time in the future), and all assignments to variables (with ':=')are blocking (i.e. they happen immediately).

Consider the following VHDL example *ex1* provided in Fig. 2, which includes only non-blocking assignments to signals. The signals have the following naming notations {*V*- an output variable; *cS* - a conditional statement; *D*- a decision; *T*- a terminal node; *C*- a condition; *W*- a value}. The keywords emphasized by bold determine if a line has a statement, a branch or conditions.

In the following we will explain how efficient code coverage measurement for the non-blocking assignments can be performed using HLDD models. This notion is totally missing in traditional code coverage metrics and tools.

Stm	Dcn	VHDL code
		...
1		**if** (cS1_C1 **and** cS1_C2)
	1	**then**
2		V1 **<=** V1_T1;
	2	**else**
3		V1 **<=** V1_T2;
		end if;
4		**case** cS2_C is
	3	**when** cS2_C_W1 =>
5		V2 **<=** V2_T1;
	4	**when** cS2_C_W2 =>
6		V2 **<=** V2_T2;
	5	**when** cS2_C_W3 =>
7		V1 **<=** V1_T2;
8		**if**(cS3_C1 **and** ((**not** cS3_C2) **or** cS3_C3))
	6	**then**
9		V2 **<=** V2_T2;
	7	**else**
10		V2 **<=** V2_T3;
		end if;
		end case;
		...

Fig. 2. A segment of the VHDL code of *latw09_ex1* design

4. Optimization level of HLDD model

In the paper we distinguish three types of HLDD representation according to their compactness. We present reduction rules for HLDD. These rules are similar to the reduction for BDDs presented in [6] and can be generalized as follows:

- *HLDD reduction rule1:* Eliminate all the redundant nodes whose all edges point to an equivalent sub-graph.
- *HLDD reduction rule2:* Share all the equivalent sub-graphs.

The three optimization levels in the increasing order of compactness are:

- *Full tree HLDD* contains all control flow branches of the design.
- *Reduced HLDD* is obtained by application of the HLDD reduction rule 1 to the full tree representation.

Minimized HLDD is obtained by application of both HLDD reduction rules 1 and 2 to the full tree representation.

Figures 3, 4 and 5 present a reduced, minimized, full-tree and HLDD model representations for the *ex1* example shown in Section 3, respectively.

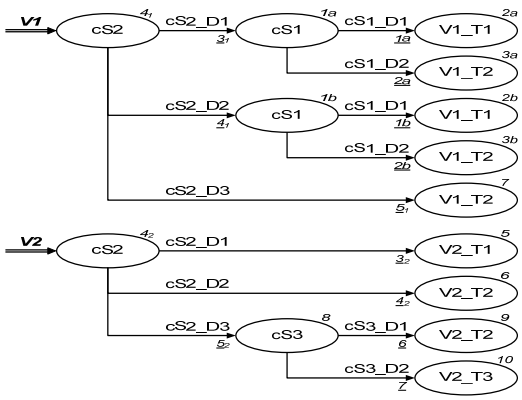


Fig. 3. Reduced HLDD for *latw09_ex1*

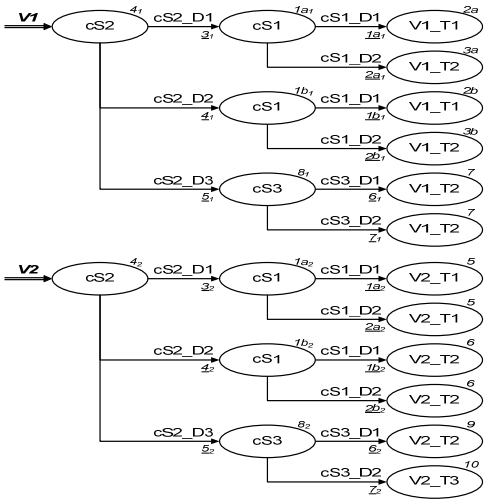


Fig. 4. Full tree HLDD for *latw09_ex1*

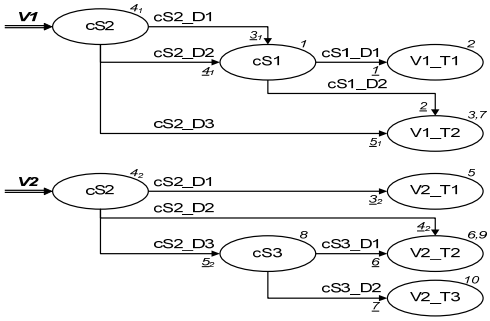


Fig. 5. Minimized HLDD for *latw09_ex1*

4. HLDD-based coverage analysis

4.1 Statement coverage mapping

The statement coverage metric has a straightforward mapping to HLDD-based coverage (an idea was proposed in [3],[4], the details are demonstrated in this paper). It maps directly to the ratio of nodes $m_{Current}$ traversed during the HLDD simulation presented in *Algorithm 1* (Subsection 2.2) to the total number of the HLDD nodes in the DUV's representation. The appropriate type of HLDD representation for the analysis of, both, statement and branch coverage metrics is the *reduced* one. The variations in the analysis caused by different HLDD representation types are discussed in Subsection 5.1.

Please consider the VHDL description of the *ex1* example provided in Fig. 3. The numbers from the first column (*Stm*) correspond to the lines with 10 statements (both conditional and assignment ones). The 14 HLDD nodes of the two graphs in Fig. 4 correspond to these statements. Covering all nodes in a HLDD model (i.e. full *HLDD node coverage*) corresponds to covering all statements in the respective HDL. However, the opposite is not true. HLDD node coverage is slightly more stringent that HDL statement coverage. Please consider as an example VHDL statements *1*, *2* and *3* and the respective nodes in Fig. 3 *1a*, *1b*, *2a*, *2b*, and *3a*, *3b*. This impact in terms of the stringency is also discussed in Subsection 5.1. At the same time some of the HDL statements have duplicated representation by the HLDD nodes (with subscript indexes) due to the fact that in HLDD-based design representation the diagrams are normally generated for each data variable separately. As an example please consider VHDL statement *4* and HLDD nodes 4_1, 4_2 in Fig.3.

4.2 Branch coverage mapping

Similar to the statement coverage, branch coverage also has very clear representation in HLDD model (an idea was proposed in [3],[4], the details are demonstrated in this paper). It is the ratio of every edge e_{active} activated in the simulation process presented by *Algorithm 1* (Subsection 2.2) to the total number of edges in the corresponding HLDD representation of the code.

The numbers (underlined) from the second column (*Dcn*) in Fig. 2 correspond to the lines with 7 branches (i.e. decisions). The 12 HLDD nodes of the two graphs in Fig. 3 correspond to these decisions. Covering all edges in a HLDD model (i.e. full *HLDD edge coverage*) corresponds to covering all branches in the respective HDL. However, similar to the previously discussed statement coverage mapping, here the opposite is also not true and HLDD edge coverage is slightly more stringent that HDL branch coverage.

The VHDL branches (Fig. 2) *1* and *2* are represented in Fig. 3 by respective edges *1a*, *1b* and *2a*, *2b*. The duplicated edges are also emphasized by subscript indexes, (e.g. 3_1,3_2 ; 4_1,4_2 ; 5_1,5_2).

5. Experiments

This subsection presents experimental results for four ITC99 benchmarks [8] that evaluate the proposed HLDD-based structural coverage analysis methodology. Table 1 presents the characteristics of the different HLDD representations introduced in Section 3.

Table 1. Characteristics of different HLDD

Design	Number of nodes			Number of edges		
	min	*red.*	*f.tree*	*min*	*red.*	*f.tree*
b01	30	57	267	52	54	267
b02	16	26	48	24	24	46
b06	47	116	440	83	111	435
b09	44	69	125	62	64	120

Table 2 shows comparison results of the proposed methodology based on different HLDD representations and coverage analysis achieved by a commercial state-of-the-art HDL simulation tool from a major CAD vendor using the same sets of stimuli.

As it can be seen, the reduced HLDDs allow equal or more stringent coverage evaluation in comparison to the commercial coverage analysis software. For three designs (*b01*, *b06* and *b09*) more stringent analysis is achieved using HLDDs. The HLDD model allows increasing the coverage accuracy up to 13 % more exact statement measurement and 14 % branch measurement (*b09* design). In our previous work [3] it was shown that HLDD-based coverage analysis has significantly lower (tens of times) computation (i.e. measurement) time overhead compared to the same commercial simulator.

Table 2. Comparison of code coverage analysis results

Design	Stimuli, (vectors)	Statement coverage,(%)			Branch coverage, (%)		
		red.	*min.*	*VHDL*	*red.*	*min.*	*VHDL*
b01	14	86.0	100	93.8	74.2	84.6	88.9
	23	**96.5**	100	**100**	**90.3**	100	**100**
b02	10	92.3	100	96.3	91.7	91.7	93.8
	14	100	100	100	100	100	100
b06	11	80.2	100	85.5	79.3	89.2	87.5
	52	**98.3**	100	**100**	**98.2**	100	**100**
b09	23	**87.0**	100	**100**	**85.9**	87.1	**100**
	33	100	100	100	100	100	100

6. Conclusions

The paper presents using high-level decision diagram (HLDD) as a suitable graph model for code coverage analysis in concurrent programming languages. The authors show that HLDD models are scalable and compact models for realistic problems can be automatically obtained. At the same time they allow covering paths of non-blocking assignments, which conventional code coverage metrics designed for procedural programming languages are unable to handle. The paper proposes optimal minimization rules to be used in HLDD model synthesis for code coverage analysis suitable for concurrent languages.

It is important to emphasize that all coverage metrics (i.e. statement, branch, condition or a combination of them) in the proposed methodology are analyzed by a single HLDD simulation tool which relies on HLDD design representation model. Different levels of coverages are distinguished by simply generating a different level of HLDD (i.e. minimized, reduced, or hierarchical with expanded conditional nodes). Experimental results demonstrate feasibility and efficiency of the proposed methodology.

Acknowledgments The work has been supported in part by EC FP 7 REGPOT project CREDES, Estonian Competence Centre CEBE, Enterprise Estonia funded ELIKO Centre, Estonian SF grants 7068 and 7483, and Estonian Information Technology Foundation (EITSA).

References

[1] Andrew Piziali, "Functional Verification Coverage Measurement and Analysis", *Springer*, 2008

[2] S. Tasiran, K. Keutzer, Coverage metrics for functional validation of hardware designs. *Design & Test of Computers*, IEEE, Volume 18, Issue 4, Jul-Aug. 2001, pp. 36-45

[3] J. Raik, U. Reinsalu, R. Ubar, M. Jenihhin, P. Ellervee, "Code Coverage Analysis using High-Level Decision Diagrams", *DDECS 2008*, April, 2008, pp. 201-206

[4] K. Minakova, U. Reinsalu, A. Chepurov, J. Raik, M. Jenihhin, R.Ubar, P. Ellervee, "High-Level Decision Diagram Manipulations for Code Coverage Analysis", *BEC 2008,* Tallinn, Estonia, October 2008, pp. 207 - 210

[5] R. Ubar, J. Raik, A. Morawiec, "Back-tracing and Event-driven Techniques in High-level Simulation with Decision Diagrams", *ISCAS 2000*, Vol. 1, pp. 208-211.

[6] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Computers, Vol. C-35, No. 8*, August 1986, pp. 677-691

[7] David A. Penry and David I. August. Optimizations for a Simulator Construction System Supporting Reusable Components *Proceedings of the 40th Design Automation Conference (DAC)*, June 2003.

[8] http://www.cerc.utexas.edu/itc99-benchmarks/bench.html