APRICOT: a Framework for Teaching Digital Systems Verification

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

Department of Computer Engineering, Tallinn University of Technology
E-mail: { jaan | maksim | anchep | uljana | raiub }@pld.ttu.ee

***Abstract* - The paper presents a new framework for digital systems verification. The framework has been developed in Tallinn University of Technology and it is called APRICOT. It supports a wide range of verification tasks including assertion checking, code coverage analysis, simulation, test generation and property checking and it is also easy to set up and use. Therefore it is highly suitable for supporting higher education and research in verification. The novelty of APRICOT lies in a system representation model called High-Level Decision Diagrams (HLDD). APRICOT has also interfaces to commonly used design formats such as VHDL, SystemC, PSL and EDIF.**

1. Introduction and Motivation

With the increase in size and complexity of modern integrated circuits, it has become imperative to address critical verification issues in the design cycle. The process of verifying correctness of designs consumes between 60% and 80% of design effort [1]. For every designer the number of verification engineers may vary from 2 to 4 depending on the design complexity. Moreover, validation is so complex that, even though it consumes most of the computational resources and time, it is still the weakest link in the design process. Ensuring functional correctness is the most difficult part of designing a hardware system [2].

All this shows there is a great need for education and training in the field of hardware verification. However, in order to provide for an adequate knowledge in this domain a wide range of verification tasks should be covered including, both, dynamic verification (validation, code coverage analysis, assertion-checking) and static verification (equivalence checking, model-checking, test generation). Learning these different aspects should be accompanied with laboratory courses and corresponding software tools. It may require obtaining and installing different verification CAD systems from different vendors to cover such kind of wide range of tasks.

Current paper presents a new framework of verification tools, called APRICOT (Assertions, PRopertIes,COde coverage and Test generation). It includes different tasks, such as assertion checking, code coverage analysis, simulation, test generation and property checking. APRICOT is easy to set up and use. Therefore it is highly suitable for supporting higher education and research in verification. The novelty of APRICOT lies in a system representation model called High-Level Decision Diagrams (HLDD). The framework has interfaces to commonly used design formats such as VHDL, SystemC, PSL and EDIF. Figure 1 presents the general structure of the APRICOT framework.

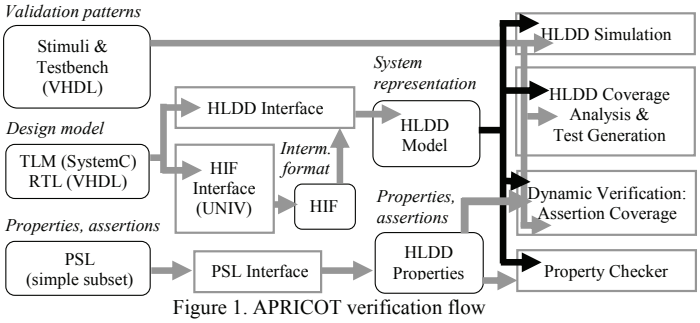

Figure 1. APRICOT verification flow

The paper is organized as follows. Section 2 introduces design modelling using HLDDs. Section 3 presents HLDD based simulation. In Section 4 code coverage analysis tool is described. Section 5 discusses assertion based verification. Section 6 presents test pattern generation on HLDD models. Finally conclusions are drawn.

2. High-Level Decision Diagrams

Decision Diagrams (DD) have been used in verification for about two decades. Reduced Ordered Binary Decision Diagrams (BDD) [3] as canonical forms of Boolean functions have their application in equivalence checking and in symbolic model checking. Additionally, a higher abstraction level DD representation, called Assignment Decision Diagrams (ADD) [4], have been successfully applied to, both, register-transfer level (RTL) verification and test.

In this paper we consider a different decision diagram representation, High-Level Decision Diagrams (HLDD) that, unlike ADDs can be viewed as a generalization of BDD. HLDDs can be used for representing different abstraction levels from RTL to behavioral. HLDDs have proven to be an efficient model for simulation and diagnosis since they provide for a fast evaluation by graph traversal and for easy identification of cause-effect relationships [5].

2.1. HLDD data structure

High-Level Decision Diagrams (HLDD) are graph representations of discrete functions. 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. A high-level decision diagram G_y can be used for representing functions $y = f(x)$.

Definition 1: A HLDD representing a discrete function $y=f(x)$ is a directed non-cyclic labeled 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 the variable domains, and Γ is a function on E. The function $Z(m_i)$ returns a pair (x_i,X_i), where x_i is the variable letter, which is labeling node m_i and X_i is the domain of x_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 $e=(m_1,m_2) \in E^2$, where E^2 is the set of all the possible ordered pairs in set E. 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 X_i, where $e=(m_i,m_j)$ and $Z(m_i)=(x_i,X_i)$. It is required that $Pm_i=\{\Gamma(e) \mid e=(m_i,m_j) \in E\}$ is a partition of the set X_i. HLDD has only one starting node (*root node*), for which there are no preceding nodes. The nodes, for which successor nodes are missing, are referred to as *terminal nodes* $M^T \in M$.

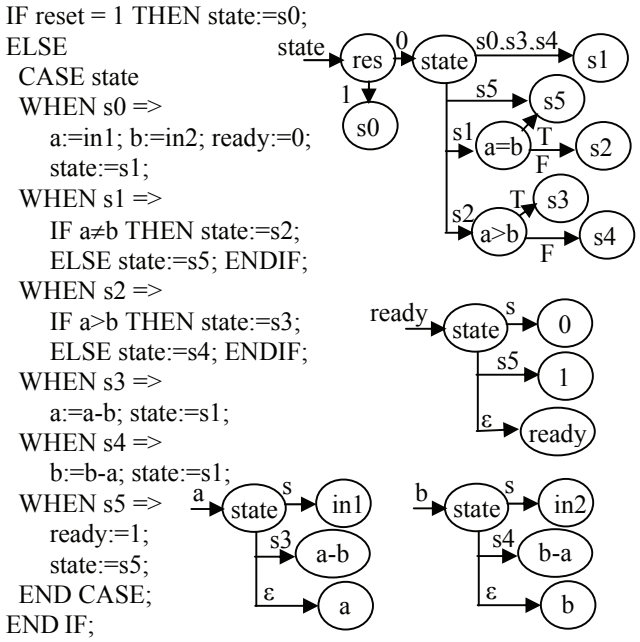

Figure 2. a) RTL VHDL and b) the corresponding HLDD

2.2. HLDDs for digital systems

HLDD models can be used for representing digital systems. In such models, the non-terminal nodes correspond to conditions or to control signals, and the terminal nodes represent data operations (functional units). Register transfers and constant assignments are treated as special cases of operations. When representing systems by decision diagram models, in general case, a network of HLDDs rather than a single HLDD is required. 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.

Fig. 2b presents the HLDD system for the RTL VHDL code shown in Fig. 2a implementing the Greatest Common Divisor (GCD) algorithm. In the Figure, T and F stand for true and false, respectively. The ε character denotes default edges.

2.3. Advantages of HLDD modeling

As an example, consider a subnetwork of a digital system and its HLDD are depicted in Figure 3. Here, R_1 and R_2 are registers (R_2 is also output), M_1, M_2 and M_3 are multiplexers, + and * denote adder and multiplier, *IN* is input bus, y_1, y_2, y_3 and y_4 serve as input control variables, and *a,b,c,d,e* denote internal buses. In the HLDD, the control variables y_1, y_2, y_3 and y_4 are labeling internal decision nodes of the HLDD with their values shown at edges. The terminal nodes are labeled by constant *#0* (reset of R_2), by word variables R_1 and R_2 (data transfers to R_2), and by expressions related to data manipulation operations of the network. By bold lines and colored nodes, a full activated path in the HLDD is shown from $Z(m^0)=y_4$ to $Z(m^T)=R_1*R_2$, which corresponds to the pattern $y_4=2$, $y_3=3$, and $y_2=0$. By colored boxes, the activated part of the network at this pattern is denoted.

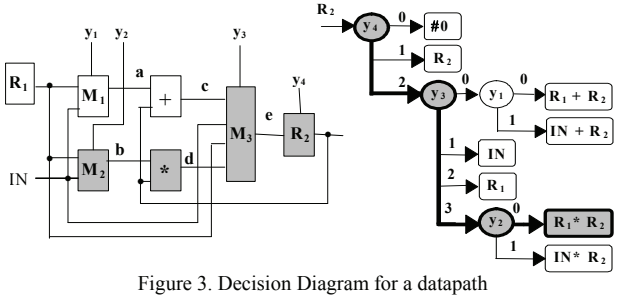

Figure 3. Decision Diagram for a datapath

The main advantage and motivation of using high-level DDs compared to the netlists of primitive functions are the increased efficiency of simulation and diagnostic modeling because of direct and compact presentation of cause-effect relationships. For example, instead of simulating the control word $y_1, y_2, y_3, y_4 = 0032$ by computing the functions $a = R_1$, $b = R_1$, $c = a + R_1$, $d = b * R_1$, e = d, ja $R_2 = e$, we need only to trace the nodes y_4, y_3 and y_2 on the HLDD and compute a single operation $R_2 = R_1 * R_2$. In case of detecting an error in R_2 the possible causes can be defined immediately along the simulated path through y_4, y_3 ja y_2 without any diagnostic analysis inside thecorresponding RTL network.

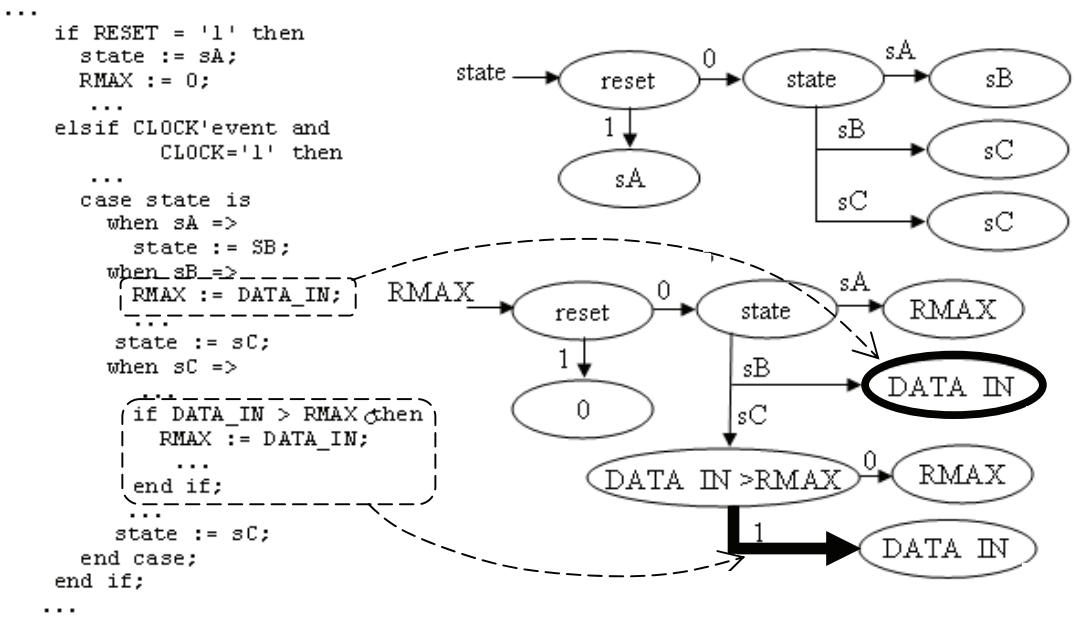

Fig. 4. HLDDs for VHDL variables *state* and *RMAX*

As a result of such a quick reasoning the debugging of a system can be considerably speeded up. The detailed analysis inside the RTL network is needed only if all the values of y_4, y_3 ja y_2 are correct. In such a way, a very efficient hierarchical debugging procedure can be carried out with HLDDs first, by a quick trace of faulty nodes in HLDDs, and then after locating the erroneous RT-level region, by exactly locating the cause of error in this region.

Algorithm 1: RTL/behavioral simulation on HLDDs

SimulateHLDD()
  For each diagram G in the model
    $m_{Current} = m_0$
    Let $x_{Current}$ be the variable labeling $m_{Current}$
    While $m_{Current}$ is not a terminal node
      If $x_{Current}$ is clocked or its DD is ranked after G then
        $Value$ = previous time-step value of $x_{Current}$
      Else
        $Value$ = present time-step value of $x_{Current}$
      End if
      For $\{\Gamma \mid Value \in \Gamma(e_{active}), e_{active} = (m_{Current}, m_{Next})\}$
        $m_{Current} = m_{Next}$
      End if
    End while
    Assign $x_{Current}$ to the DD variable x_G
  End for
End SimulateHLDD

3. HLDD-BASED SIMULATION

The basis for assertion coverage analysis in this paper is a simulator engine relying on HLDD models. In our earlier works [5], we have implemented an algorithm supporting, both, Register-Transfer Level (RTL) and behavioral design abstraction levels. This algorithm is briefly explained below and it will be used for simulating the system model. In the RTL style, the algorithm takes the previous time step value of variable x_j labeling a node m_i if x_j represents a clocked variable in the corresponding HDL. Otherwise, the present value of x_j will be used. In the case of behavioral HDL coding style HLDDs are generated and ranked in a specific order to ensure causality. For variables x_j labeling HLDD nodes the previous time step value is used if the HLDD diagram calculating x_j is ranked after current decision diagram. Otherwise, the present time step value will be used.

Algorithm 1 presents the HLDD based simulation engine for RTL, behavioral and mixed HDL description styles. (Refer to Section 2.1 for HLDD data structure definition).

4. CODE COVERAGE ANALYSIS

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.

Various coverage metrics exist such as code coverage, parameter coverage and functional coverage. In this paper, only code coverage would be used, which provides insight into how thoroughly the code of a design is exercised by a

suite of simulations. The main disadvantage of code coverage metrics lies in the fact that they only measure the quality of the test case in stimulating the implementation and do not necessarily prove its correctness with respect to the specification. On the other hand, code coverage analysis is a well-defined, well-scalable procedure and, thus, applicable to large designs. The goal of current work is to propose a method for speeding up the analysis by implementing new models for simulating coverage items.

Following Miller and Maloney [3], 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, etc. [2][4]. The *statement coverage* metric measures the percentage of code instructions exercised with respect to total instruc-tions contained in the code by the program stimuli. *Toggle coverage* shows the percentage of bits toggling in the nodes in the design, i.e., how many bits change their state from 0 to 1 or vice versa. In the case of *branch coverage*, we measure the ratio of branches in the control flow graph of the code that are traversed under the set of stimuli. *Path coverage* measures the percentage of paths in the control flow graph is exercised by the stimuli. A potential goal of software testing is to have 100 % path coverage that implies branch and statement coverage. However, full path coverage is a very stringent requirement as the num-ber of paths in a program may be exponentially related to program size.

In this paper, we present a method and a tool for fast analysis of classical code coverage metrics, such as statement, branch and toggle coverage. We introduce High-Level Decision Diagrams (HLDD) model for efficient code coverage analysis and show how those classical coverage metrics map to HLDD constructs. Consider the example in Fig. 4. The statement coverage maps directly to the ratio of nodes mCurrent traversed during the HLDD simulation presented in Algorithm 1. For example, see Fig. 4 for HLDD representations of state and data register variables in a VHDL design. Covering all nodes in the HLDD model 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. This is due to the fact that in HLDDs diagrams are generated to each data variable separately. Such partition on variables includes an additional context to statement coverage. The HDL example in Fig. 3 has 12 statements while the corresponding HLDD model contains 14 nodes.

Similar to the statement coverage, branch coverage has also very clear representation in HLDD simulation. The ratio of every edge eactive activated in the simulation process of Algorithm 1 constitutes to HLDD branch coverage.

HLDD toggle coverage is calculated similarly to traditional HDL toggle coverage. However, in this paper a more stringent approach has been selected, where, both, rising and falling front toggling are counted separately. Furthermore, toggling is measured in DD nodes, not in HDL variables. For example, the branch coverage item corresponding to DATA_IN > RMAX = true in the VHDL code of the b04 design maps to the edge denoted by a bold arrow in the HLDD in Figure 4. The statement RMAX := DATA_IN is represented by the terminal node surrounded by bold circle in the corresponding HLDD.

The code coverage analysis tool integrated into apricot allows the user to evaluate its design and test bench against different code coverage metrics. The model and coverage items are represented by a graph model of HLDDs.

5. Assertion-based Verification

Assertions have been found to be beneficial for solving a wide range of tasks in systems design ranging from modelling, verification and even manufacturing test [9]. In this paper we consider assertion-based verification which has been recognized as an efficient approach to cope with many difficulties [1] in the state-of-the-art digital systems functional verification. Verification assertions can be used in both dynamic and static verification (see section 6!).

Property Specification Language (PSL) is a recently accepted IEEE standard language [10] that is commonly used to express the assertions. In this paper, we present an approach to checking PSL assertions using HLDDs. Here, the assertions are translated to HLDD graphs and integrated into fast HLDD-based simulation. The structure of HLDD design representation with a temporal extension proposed in [11] allows straightforward and lossless translation of PSL properties. The efficiency of the HLDD-based approach is demonstrated by the experimental results, where the proposed method is compared against a commercial simulator with PSL assertions support QuestaSim from Mentor Graphics.

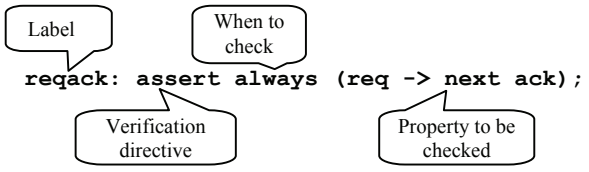

Figure 5. PSL property "reqack"

An example PSL property *reqack* structure is shown in Figure 5. Its possible timing diagram is also illustrated by Figure 6a. It states that `ack` must become high next after `req` being high. A system behaviour that activates *reqack* property however obviously violating it is demonstrated in Figure 6b. Figure 6c shows the case when the property was not activated.

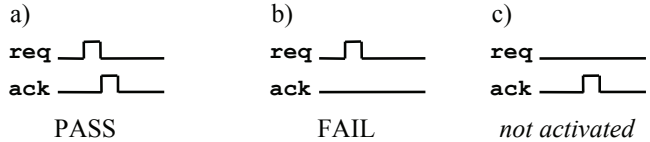

Figure6. Timing diagrams for the property "reqack"

Let us consider an example PSL property for GCD implementation given in Figure 1.

```
P1: assert always !ready and (a=b)->next_e[1:3]ready
```

Assertion P1 states that whenever 'ready' is low and 'a' is equal to 'b' then during the next three cycles ready must become true. The resulting HLDD graph describing this property is shown in Figure 7.

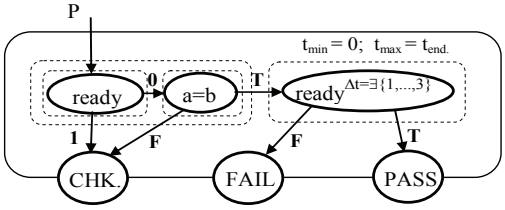

Figure7. HLDD for property P1

TABLE 1. EXECUTION TIME COMPARISON

Design	Stimuli Length (clocks)	APRICOT			Commercial
		Simulation Time (seconds)	Checking Time (seconds)	Total Time (seconds)	Total Time (seconds)
gcd	10,000	0.02	0.04	0.06	0.67
	100,000	0.20	0.40	0.60	1.71
	1,000,000	2.07	4.87	6.94	13.52
b00	10,000	0.03	0.03	0.06	0.79
	100,000	0.30	0.30	0.60	1.83
	1,000,000	3.43	2.95	6.38	13.84
b04	10,000	0.05	0.03	0.08	0.84
	100,000	0.54	0.28	0.82	2.21
	1,000,000	5.47	3.61	9.08	19.23
b09	10,000	0.02	0.04	0.06	0.72
	100,000	0.22	0.39	0.61	1.74
	1,000,000	2.21	4.55	6.76	12.4

Table 1 shows the experimental results of assertion checking execution times comparison between HLDD simulator and a popular commercial tool. The experimental benchmarks are GCD implementation given in Figure 1 and 3 designs from ITC'99 benchmarks family. A set of 5 realistic assertions has been created for each benchmark. The assertions selected for GCD are the following:

```
p1: assert always( ((not ready) and (a = b)) -> next_e[1 to 3](ready) );
p2: assert always (reset -> next next((not ready) until (a = b)));
p3: assert never ((a /= b) and ready);
p4: assert never ((a /= b) and (not ready));
p5: assert always( reset -> next_a[2 to 5](not ready) );
```

Similar to the code coverage analysis tool described in Section 4 the assertion-checker displays statistics in a human-readable format using ASCII graphics. Thus the tool is well suited for teaching assertion based verification.

6. TEST PATTERN GENERATION AND FORMAL METHODS

The formal methods implemented in the APRICOT framework include high-level Automated Test Pattern Generation (ATPG) and formal property checking. The latter is reduced to using the first one as a model-checking engine. Both, the ATPG engine and the property checker are explained below.

In order to perform high-level test pattern generation, an algorithm and a tool called Decider has been implemented [12]. The algorithm runs in two phases. During the first phase, constraints required to activate test paths in the system are extracted using HLDD models. At the second stage, the constraints are solved relying on traditional solvers. The test generation constraints considered in current paper can be divided into two categories: path activation constraints and transformation constraints. Path activation constraints correspond to the logic conditions in the control flow graph that have to be satisfied in order to perform propagation and value justification through the circuit.

Transformation constraints, in turn, reflect the value changes along the paths from the inputs of the high-level module under test to the primary inputs of the whole circuit. These constraints are needed in order to derive the local test patterns for the module under test. Both types of constraints can be represented by common data structures and manipulated by common procedures for creation, update, modeling and simulation.

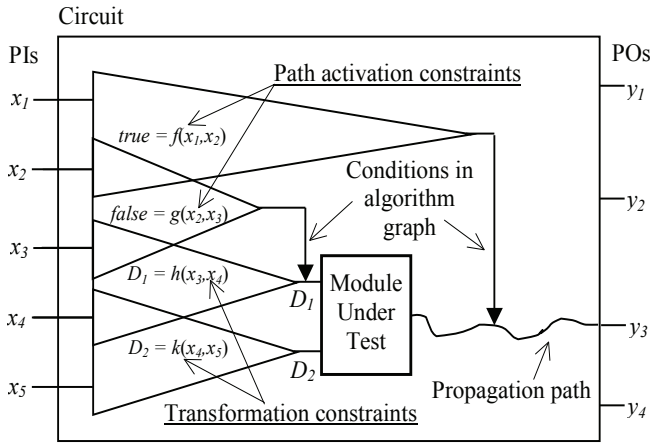

Fig. 8. HLDD-based path activation with constraint extraction

Figure 8 explains the role of these constraints in test generation for a circuit module. In the Figure there are two path activation constraints: true = f(x1,x2) and false = g(x2,x3). The first one is necessary to propagate the value from the output of the module to the primary output y3 of the circuit. The latter is required for justification of the first input (D1) of the module under test. Both these constraints are extracted from the conditional nodes traversed in the HLDD of the system during high-level path activation. In addition, the Figure presents two transformation constraints. These constraints represent the function for computing the value of the corresponding module input depending based on the values of primary inputs of the circuit.

In order to support property checking in APRICOT, this particular task has been reduced to ATPG relying on the above-described Decider engine [12]. Here, in addition to the system model, properties to be checked will also be transformed into HLDDs. The two HLDD models will then be

composed into a single system and a modified ATPG will be used to generate counter examples (i.e. tests) for the properties. The ATPG is able to prove properties when it exhausts its search space. The limitation of the approach is that only bounded property checking can be performed. See Fig. 9 for the general setup of property checking with APRICOT.

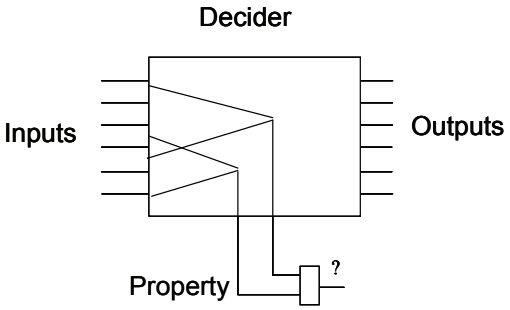

Fig. 9. Reducing model-checking to ATPG

7. Conclusions

The paper presented a new framework for digital systems verification. The framework has been developed in Tallinn University of Technology and it is called APRICOT. It supports a wide range of verification tasks including assertion checking, code coverage analysis, simulation, test generation and property checking and it is also easy to set up and use. Therefore it is highly suitable for supporting higher education and research in verification. The novelty of APRICOT lies in a system representation model called High-Level Decision Diagrams (HLDD). APRICOT has also interfaces to commonly used design formats such as VHDL, SystemC, PSL and EDIF.

Acknowledgements

The work has been supported partly by EC FP 6 STREP research project VERTIGO FP6-2005-IST-5-033709, Enterprise Estonia funded ELIKO Development Center, Estonian Information Technology Foundation (EITSA) and Estonian SF grants 6717, 7068 and 5910.

References

[1] International Technology Roadmap for Semiconductors 2006 report, [URL] www.itrs.net, 2006

[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, Pages 36-45.

[3] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35, 8:677-691, 1986

[4] V. Chayakul, D. D. Gajski, L. Ramachandran, "High-Level Transformations for Minimizing Syntactic Variances", *Proc. of ACM/IEEE DAC*, pp. 413-418, June 1993.

[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] J. C. Miller, C. J. Maloney, *Systematic Mistake Analysis of Digital Computer Programs*. Communications of the ACM, 1963, Pages 58-63.

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

[8] William K. Lam, "*Hardware Design Verification: Simulation and Formal Method-Based Approaches*", Pearson Education Inc., 2005, 585 pages.

[9] Y. Oddos, et al. Prototyping Generators for On-line Test Vector Generation Based on PSL Properties, *DDECS*, 2007.

[10] IEEE-Commission, "IEEE standard for Property Specification Language (PSL)," 2005, IEEE Std 1850-2005.

[11] Maksim Jenihhin, et al. Temporally Extended High-Level Decision Diagrams for PSL Assertions Simulation. *Proceedings of the 13th IEEE European Test Symposium*, 2008.

[12] Jaan Raik, Raimund Ubar, Taavi Viilukas, Maksim Jenihhin. Mixed Hierarchical-Functional Fault Models for Targeting Sequential Cores. *Elsevier Journal of Systems Architecture*.