Temporally Extended High-Level Decision Diagrams for PSL Assertions Simulation

Maksim Jenihhin, Jaan Raik, Anton Chepurov, Raimund Ubar
Department of Computer Engineering, Tallinn University of Technology, Estonia
E-mail: { maksim|jaan|raiub }@pld.ttu.ee, anton.chepurov@gmail.com

Abstract

The paper proposes a novel method for PSL language assertions simulation-based checking. The method uses a system representation model called High-Level Decision Diagrams (HLDD). Previous works have shown that HLDDs are an efficient model for simulation and convenient for diagnosis and debug. The presented approach proposes a temporal extension for the existing HLDD model aimed at supporting temporal properties expressed in PSL. Other contributions of the paper are methodology for direct conversion of PSL properties to HLDD and HLDD-based simulator modification for assertions checking support. Experimental results show the feasibility and efficiency of the proposed approach.

1. Introduction

Assertions have been found to be beneficial for solving a wide range of tasks in systems design ranging from modeling, verification and even manufacturing test [1]. In this paper we consider assertion-based verification which has been recognized as an efficient approach to cope with many difficulties in the state-of-the-art digital systems functional verification. Verification assertions can be used in both dynamic and static verification. Here, we consider only the first case, where assertions play the role of monitors for particular system behavior during simulation.

Property Specification Language (PSL) is a recently accepted IEEE standard language [11] that is commonly used to express the assertions. The research on the topic of converting PSL assertions to design representation such as HDL is gaining its popularity. There are several approaches published in recent time [2, 3, 4, 15]. The most widely known tool for this task is FoCs by IBM [5].

Our first attempt [14] of PSL properties translation to HLDD was implying the generation of VHDL checkers by IBM's FoCs as an intermediate step. However this experience has revealed particular limitations and inefficiency for HLDD-based assertions creation. Moreover, checkers synthesis from PSL properties are efficient mainly for the case where checkers are to be used in hardware emulation. The application of the same checker constructs for simulation in software may lack efficiency due to target language concurrency and poor means for temporal expressions. Synthesis of checkers hardware for emulation is out of the scope of current paper.

In this paper, we present an approach to checking PSL assertions using High-Level Decision Diagrams (HLDD). Here, the assertions are translated to HLDD graphs and integrated into fast HLDD-based simulation. The structure of an HLDD design representation with the temporal extension proposed in this paper allows straightforward and lossless translation of PSL properties. HLDDs are a convenient model for diagnosis and debug since they provide for easy identification of cause-effect relationships. The feasibility and efficiency of the proposed approach are demonstrated by the experimental results, where the proposed method is compared against a commercial simulator with PSL assertions support from a major CAD vendor.

High-Level Decision Diagrams have been proposed and further developed by the authors in [13]. For more than a decade this model of digital design representation has been successfully applied for design simulation and test generation research areas. Recently the group has started to apply HLDD model in its verification flow. The emphasis of this paper is put on assertion checking in simulation-based verification. However, several other tools working with HLDD are under development, including dynamic verification code coverage analysis, formal methods of stimuli generation and model checking. The latter are relying

on the engine of HLDD based ATPG known as DECIDER [7].

This paper is organized as follows. PSL and its supported subset are discussed in Section 2. Section 3 defines the existing HLDD graph model and describes the HLDD-based simulation process. Section 4 presents the temporal extension for HLDD model, discusses the hierarchical creation of THLDD representation of PSL properties and THLDD assertions checking process, Section 5 provides the experimental results, and finally conclusions are drawn.

2. PSL subset for assertion checking

Assertion-based verification popularity has encouraged a common *Property Specification Language* development by the Functional Verification Technical Committee of Accellera. After a process in which donations from a number of sources were evaluated, the Sugar language from IBM was chosen as the basis for PSL. The latest Language Reference Manual for PSL version 1.1 was released in 2004 [10]. The language became an IEEE 1850 Standard in 2005 [11].

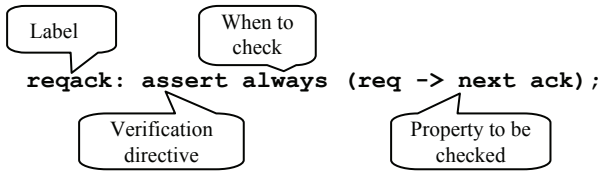
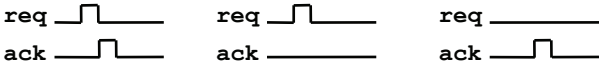


Figure 1. PSL property "reqack"

An example PSL property reqack structure is shown in Figure 1. Its possible timing diagram is also illustrated by Figure 2a. 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 2b. Figure 2c shows the case when the property was not activated.



a) PASS b) FAIL c) *not activated*

Fig. 2. Timing diagrams for the property "reqack"

For the convenience of verification engineers PSL is a multi-flavoured language, which means that it supports common constructs of VHDL, Verilog, IBM's GDL, SystemVerilog and SystemC [15]. PSL is also a multi-layered language [10]. The layers include:

- *Boolean layer* – the lowest one, consists of Boolean expressions in HLD (e.g. `a &&(b || c)`)
- *Temporal layer* – sequences of Boolean expressions over multiple clock cycles, also supports Sequential Extended Regular Expressions (SERE) (e.g. `{A[*3];B} |-> {C}`)
- *Verification layer* - it provides directives that tell a verification tool what to do with specified sequences and properties.
- *Modeling layer* - additional helper code to model auxiliary combinational signals, state machines etc. that are not part of the actual design but are required to express the property.

The temporal layer of PSL language has two constituents:

- Foundation Language (FL), that is Linear Temporal Logic (LTL) with embedded SERE
- Optional Branching Extension (OBE), that is Computational Tree Logic (CTL)

The second one considers multiple execution paths and models design behaviour as execution trees. CTL can only be used in formal verification. Therefore, in this paper we will consider only the FL part of PSL. In fact, only FL, or more precisely its subset known as PSL Simple Subset, is suitable for dynamic assertion checking. This subset is explicitly defined in [10] and loosely speaking it has two requirements for time: to advance monotonically and be finite and restrictions on types of operands for several operators. In this paper only several LTL operators without SERE support were implemented. However, as it will be shown later, the support for SERE as well as for any other language constructs can be easily added by just proprietary library extension.

3. High-level decision diagrams

Decision Diagrams (DD) have been used in verification for about two decades. Reduced Ordered Binary Decision Diagrams (BDD) [8] 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) [9], 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 TLM (Transaction Level Modeling) and 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 [6].

3.1. 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 $y = f(x)$.

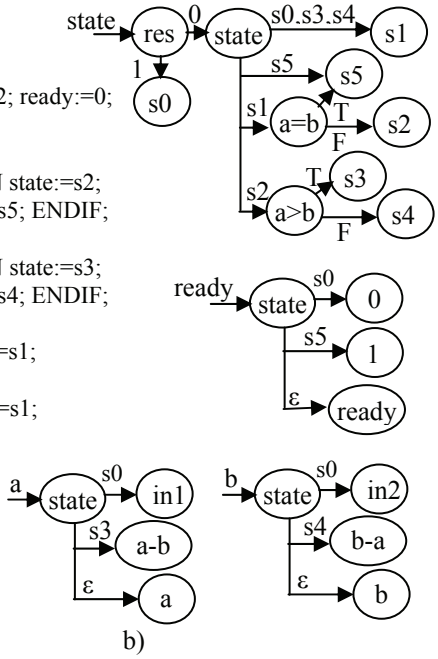
Definition 1: A HLDD representing a discrete function $y=f(x)$ is a directed non-cyclic labeled graph that can be defined as a quadruple $G=(M,E,Z,\Gamma)$, where M is a finite set of vertices (referred to as *nodes*), E is a finite set of *edges*, Z is a function which defines the *variables labeling 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* $M^T \in M$.

3.2. 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. 3b presents the HLDD system for the RTL pseudocode shown in Fig. 3a 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.

```
IF res = 1 THEN state:=s0;
ELSE
  CASE state
  WHEN s0:
     a:=in1; b:=in2; ready:=0;
     state:=s1;
  WHEN s1:
     IF a≠b THEN state:=s2;
     ELSE state:=s5; ENDIF;
  WHEN s2:
     IF a>b THEN state:=s3;
     ELSE state:=s4; ENDIF;
  WHEN s3:
     a:=a-b; state:=s1;
  WHEN s4:
     b:=b-a; state:=s1;
  WHEN s5:
     ready:=1;
     state:=s5;
  END CASE
END IF
```



a) b)

Fig. 3. a) An RTL pseudocode and b) its HLDD

3.3. Systems simulation using HLDDs

The basis for assertion coverage analysis in this paper is a simulator engine relying on HLDD models. In our earlier works [6], 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 3.1 for HLDD data structure definition).

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 {Γ | Value ∈ Γ(e_active), e_active =( m_Current, m_Next)}
        m_Current = m_Next
      End if
    End while
    Assign x_G = x_Current
  End for
End SimulateHLDD
```

3.4. Advantages of HLDD-based modeling

As an example, consider a datapath of a digital system and its HLDD depicted in Figure 4. 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 and input bus, y_1, y_2, y_3 and y_4 serve as input control variables, and *a, b, c, d* and *e* denote internal buses, respectively. 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 a 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 grey nodes, a full activated path in the HLDD is shown from $Z(m_0)$=y_4 to $Z(m^T \in M^T)=R_1*R_2$, which corresponds to the pattern $y_4=2$, $y_3=3$, and $y_2=0$. The activated part of the network at this pattern is denoted by grey boxes.

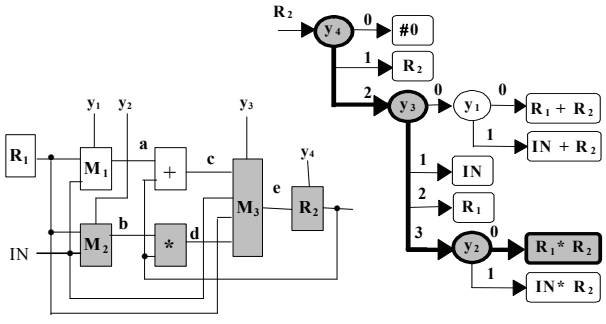


Figure 4. An HLDD for a datapath

The main advantage and motivation of using HLDDs compared to the netlists of primitive functions is the increased efficiency of simulation and diagnostic modeling because of direct and compact representation 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 $a = R_1$, $b = R_1$, $c = a + R_2$, $d = b * R_2$, e = d, and $R_2 = e$, we only need 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 and y_2 without any diagnostic analysis inside the corresponding RTL netlist. As a result of such a quick reasoning the debugging of a system can be considerably simplified. A detailed analysis inside the RTL netlist is needed only if all the values of y_4, y_3 and 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.

4. Temporally extended HLDDs

The novel contribution of this paper is an extension for the traditional HLDD model defined in Section 3 with the aim to support temporal logic properties. The extension is referred to as Temporally extended HLDDs (THLDD). This Section presents the definition of THLDDs and proposes an algorithm for hierarchical generation of the model based on a concept of templates of PSL constructs referred to as Primitive Property Graphs.

4.1. Basic definitions

Unlike the HLDD described in the previous Section, the temporally extended High-Level Decision Diagrams are aimed at representing temporal logic properties. A temporal logic property P at the time-step $t_i \in T$ denoted by $P_{ti} = f(x,T)$, where $x = (x_1, \ldots, x_n)$ is a vector defined on a finite domain $X = X_1 \times \ldots \times X_n$ and $T = \{t_1, \ldots, t_m\}$ is a finite set of time-steps. In order to represent the temporal logic assertion $P_{ti} = f(x,T)$, a temporally extended high-level decision diagram G_P can be used.

Definition 2**:** A Temporally extended High-Level Decision Diagram (THLDD) is a directed labeled graph that can be defined as a sixtuple $G_P=(M,E,T,Z,\Gamma,\Phi)$, where M is a finite set of nodes, E is a finite set of edges, T is a finite set of time-steps, Z is a function which defines the variables labeling the

nodes and their domains, Γ is a function on E representing the activating conditions for the edges, and Φ is a function on M defining temporal relationships for the labeling variables.

The graph G_P has exactly three terminal nodes $M^T \in M$ labeled by constants, whose semantics is explained below:

- FAIL – assertion P has been simulated and does not hold;
- PASS – the assertion has been simulated and holds;
- CHECKING – P has been simulated and it does not fail, nor does it pass non-vacuously (See Section 4.2 for discussion on vacuity).

The function $\Phi(m_i)$ represents the relationship indicating at which time-steps $t \in T$ the node labeling variable $x_l \in Z(m_i)$ should be evaluated. More exactly, the function returns the range of time-steps relative to current time t_{curr} where the value X_l of variable x_l must be read. We denote the relative time range by Δt and calculus of variable x_k using the time-range $\Phi(m_i)= \Delta t$ by $x_l^{\Delta t}$. We distinguish three cases:

- $\Delta t=k$, where k is a constant. In other words, the value of the variable x_l has to be taken k time-steps from current time-step t_{curr}.
- $\Delta t=\forall\{j,...,k\}$, meaning that $x_l^{\Delta t_j} \wedge ... \wedge x_l^{\Delta t_k}$ is true, i.e. variable x_l is true at every time-step between t_{curr+j} and t_{curr+k}.
- $\Delta t=\exists\{j,...,k\}$, meaning that $x_l^{\Delta t_j} \vee ... \vee x_l^{\Delta t_k}$ is true, i.e. variable x_l is true at least at one of the time-steps between t_{curr+j} and t_{curr+k}.

For Boolean, i.e. non-temporal variables $\Delta t=0$.

One of the motivations for introduction of PSL was the poor ability of standard HDL languages to express temporal relations between expressions in assertions. The main instruments for this purpose used in PSL are repetition operators of its own and of Sequentially Extended Regular Expressions (SERE). A powerful part of the repetition operators are their auxiliary suffixes (e.g for next* family they are next_a, next_e!, next_event etc). In current paper we propose a temporal extension for HLDD model that supports the following 3 PSL constructs (See Table 1).

Table 1. Temporal relationships in THLDDs

Sample PSL construct	Semantics	Equivalent THLDD notation
`next[ k] x`	x holds at t_k time-steps from now	$x^{\Delta t=n}$
`next_a[j :k] x`	x holds at all timestep within t_j to t_k	$x^{\Delta t=\forall\{j,... k\}}$
`next_e[j :k] x`	x holds at least once within t_j to t_k range	$x^{\Delta t=\exists\{j,... k\}}$

Additionally we introduce the notion of t_{end} as the final time-step that occurs at the end of simulation and is implicitly determined by one of the following cases:

- Number of test vectors
- The amount of time provided for simulation
- Simulation interruption

Note, that THLDD is an extension of HLDD defined in Section 3 as it includes temporal operators and permits cycles inside the graph. The main purpose of the proposed temporal extension is transferring additional information and directives to the HLDD Simulator that will be used for assertions checking.

4.2. Recursive generation of THLDDs

The idea of the proposed method relies on the principle of 'divide and conquer'. The method is based on partitioning PSL properties into elementary entities containing only one operator. There are two main stages in the approach. The first one is preparatory and consists of *Primitive Property Graphs Library* creation for elementary operators. The second stage is recursive *hierarchical construction* of the Temporally extended HLDD (THLDD) for a complex property using the PPG Library elements.

Prior to the THLDD construction procedure a *Primitive Property Graph* (PPG) should be created for every PSL operator supported by the proposed approach. All the created PPGs are combined into one *PPG Library*. The library is extensible and should be created only once. It implicitly determines the supported PSL subset. The method currently supports only weak versions of PSL operators. However, by means of the supported operators a large set of properties expressed in PSL can be derived.

Primitive Property Graph is a special type of HLDD graph. Compared to the basic HLDD model used for representing the design (defined in Section 3), these graphs have two distinctions. The first distinction is the requirement for all the PPGs to have a standard interface. The second distinction is usage of the HLDD model with a temporal extension. In the following the distinctions will be discussed in detail.

The *standard interface* for all PPGs was introduced in order to support the hierarchy in a recursive complex property construction described in the next subsection. PPG has one root node and exactly 3 terminal nodes (CHECKING, FAIL and PASS, respectively), as opposed to an arbitrary number of terminal nodes in usual HLDD graph. It has also an optional time range, which shows between which time-steps t_{min} and t_{max} the assertion has to be checked. The standard PPG interface is shown in Figure 5.

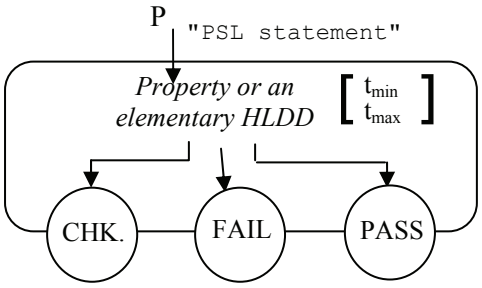


Figure 5. Standard PPG interface

Note, that in this paper we support only weak operators. In order to extend the subset to support strong operators a third output PENDING would be needed. Example PPGs created for 4 PSL operators are shown in Figure 6. Note, that the logic implication operator '->' in Fig. 6b exits to the terminal node 'CHECKING' when the precondition Pa fails. This is due to the fact that in assertion checking the designer is not interested in non-vacuous passes of the property. Also, consider the PPG for operator 'until' shown in Fig. 6d. It is the SERE and 'until' operators that create cyclic THLDDs. In the following subsection an assertion checking algorithm is presented that is capable of handling such cycles.

Complex properties are hierarchically constructed from elementary graphs in PPGs Library in the following way. At first, the property should be parsed. During the parsing phase the PSL property is partitioned into entities containing one operator only. The hierarchy of operators is determined by the PSL operators precedence specified by IEEE1850 Standard.

Hierarchical construction is performed in the top-down manner. It starts for the operators with lowest precedence where the sub-operations are then recursively substituted with the operators having higher precedence. For example, always and never operators have the lowest level of precedence and consequently their corresponding PPGs have the highest level in the hierarchy. The sub-properties (operands) are step-by-step substituted by lower level PPGs until the lowest level where sub-properties are pure signals or HLD operations.

Let us consider an example PSL property for GCD implementation given in Figure 3.

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

The resulting THLDD graph describing this property is shown in Figure 7.

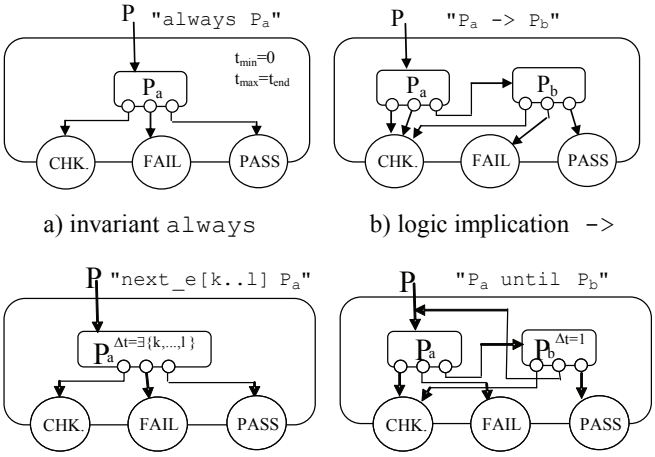


Figure 6. PPGs for a set of PSL operators

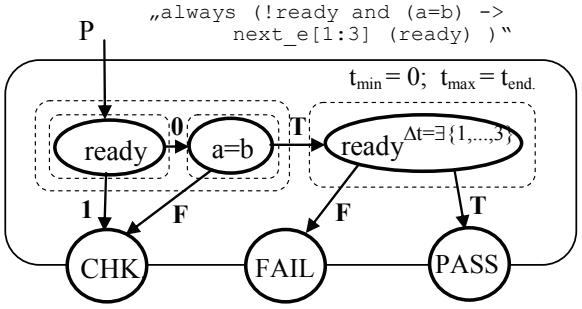


Figure 7. PPGs for a set of PSL operators

Algorithm 2. Assertion checking based on THLDDs

AssertionCheck()
  For each diagram G in the model
    For $t=t_{min}...t_{max}$
      $m_{Current} = m_0$; $t_{now} = t$
      Let $x_{Current}$ be the variable labeling $m_{Current}$
      While $m_{Current} \notin M^T$ and
        If $t_{now} > t_{max}$ then
          Exit the function
        End if
        $\Delta t = \Phi(m_{Current})$
        $Value = x_{Current}$ at time-step $t_{now+\Delta t}$
        $t_{now} = t_{now+\Delta t}$
        End case
        For $\{\Gamma \mid Value \in \Gamma(e_{active}), e_{active} = (m_{Current}, m_{Next})\}$
          $m_{Current} = m_{Next}$
        End if
      End while
      Assign $x_G = x_{Current}$ at time-step t_{now}
    End for /* $t= t_{min}...t_{max}$ */
  End for
End AssertionCheck

4.3. Assertion checking using THLDDs

The algorithm for assertion checking using an extension to the basic HLDD simulation presented in Section 3 is presented in Algorithm 2. This step is preceeded by executing Algorithm 1 which calculates the simulation trace that is a starting point for assertion checking. Algorithm 2 is an extension of HLDD simulation as it takes into account temporal information at the nodes and has an exit condition in order to avoid eternal loops that are due to the cyclic nature of the general case of THLDDs.

5. Experimental results

Table 2 shows the experimental results of assertion checking execution times comparison between THLDD simulator and a state-of-the-art commercial tool from a major CAD vendor. The experimental benchmarks are GCD implementation given in Figure 3 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) );
```

Both simulators were supplied with the same sequences of realistic stimuli providing a good coverage for the assertions. The test lengths are shown in the second column of Table 2. The third and fourth columns show the simulation (Algorithm 1) and assertion checking (Algorithm 2) execution times required for the THLDD simulator. The fifth (highlighted) and the sixth columns are the total execution time taken by the proposed approach and the commercial tool, respectively. The values in the sixth column include approximately 0.5 sec of simulation initialization time for the commercial tool that was impossible to exclude from the measurement. The both tools have shown the identical responses about the assertion satisfactions and violations.

The experimental results show the feasibility of the proposed approach and a significant speed-up (2 times) in the execution time required for design simulation with assertion checking by the proposed approach compared to state-of-the-art commercial tool.

Table 2. Execution time comparison

Design	Stimuli Length (clocks)	The proposed approach			Commercial tool
		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

6. Conclusions

Current paper proposed a novel method for checking Property Specification Language (PSL) assertions using a new model representation called Temporally extended High-Level Decision Diagrams (THLDDs). Previous works have shown that HLDDs are 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. In this paper, the model was extended to support temporal operations inherent in PSL properties and also to directly support assertion checking. We presented a hierarchical approach to generate THLDDs based on a library of Primitive Property Graphs (PPG). Basic algorithms for THLDD based assertion checking were discussed.

As a future work we see development of assertion checking methods based on event-driven simulation using THLDDs and their integration to design error diagnosis and debug solutions.

Acknowledgements

The work has been supported by EC FP 6 project VERTIGO FP6-2005-IST-5-033709, Estonian SF grants 7068 and 7483, and Estonian Information Technology Foundation (EITSA).

References

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

[2] S. Gheorghita and R. Grigore, "Constructing Checkers from PSL Properties," 15th International Conference on Control Systems and Computer Science (CSCS15), vol. 2, pp. 757–762, 2005.

[3] Bustan D., Fisman D., Havlicek J. Automata construction for PSL. The Weizmann Institute of Science, Technical Report MCS05-04.

[4] M. Boulé and Z. Zilic. Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties.Proc. HLDVT'06, 2006.

[5] IBM AlphaWorks, "FoCs Property Checkers Generator ver. 2.04," [*www.alphaworks.ibm.com/tech/FoCs*], 2007.

[6] 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.

[7] J. Raik, R. Ubar, Fast Test Generation for Sequential Circuits Using Decision Diagrams Representations, *JETTA,* 2000.

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

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

[10] Accellera, "Property Specification Language Reference Manual", v1.1, June 9, 2004.

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

[12] C. Eisner, D. Fisman, "A Practical Introduction to PSL", Springer Science, 2006.

[13] R. Ubar. "Test Synthesis with Alternative Graphs", *In IEEE Design and Test of Computers*, pp. 48–57. 1996.

[14] Maksim Jenihhin, Jaan Raik, Anton Chepurov, Raimund Ubar. Assertion Checking with PSL and High-Level Decision Diagrams. *IEEE WRTLT'07*, October 12-13, 2007.

[15] K. Morin-Allory, D. Borrione. Proven correct monitors from PSL specifications, *Proc. DATE*, 2006.