



STREP – IST 033709 VERTIGO

Verification & Validation of Embedded System Design Workbench

Deliverable D2.6 High-level decision diagrams

DUE DATE 30 November 2007
ACTUAL DATE 5 December 2007
START OF PROJECT 01 June 2006 DURATION 30 months
ABSTRACT This document defines extensions of high-level decision diagram models to support behavioural modelling and verification.
AUTHOR, COMPANY Jaan Raik, Maksim Jenihhin, Raimund Ubar, Anton Chepurov, Tallinn University of Technology
WORKPACKAGE/TASK WP2
FILING CODE VERTIGO/Deliverables/D2.6_VERTIGO
KEYWORDS High-level decision diagrams, modelling, hardware description languages, property specification language

DOCUMENT HISTORY

Release	Date	Reason of change	Status	Distribution
1	5/12/2007		Final	Internal + EC

Table of Contents

1.	Introduction	1
2.	High-level decision-diagrams	2
2.1	Basic Definitions	2
2.2	Modelling RTL circuits	3
3.	HLDD extensions for behavioural modelling	8
3.1	Behavioural HLDD modelling	8
3.2	Behavioural HLDD simulation	9
3.3	Hardware design language interfaces.....	10
4.	HLDD extensions for assertion-based verification	11
4.1	PSL subset for HLDD generation	11
4.2	Converting VHDL checkers to HLDDs	13
4.3	Temporally extended HLDDs	14
5.	HLDD extensions for code coverage analysis	19
6.	Conclusions	22
7.	References	23

1. Introduction

Decision Diagrams (DD) have been used in verification for about two decades. Reduced Ordered Binary Decision Diagrams (BDD) [1] as canonical forms of Boolean functions have their application in equivalence checking and in symbolic model checking. In addition, a higher abstraction level DD representation, called Assignment Decision Diagrams (ADD) [2], have been successfully applied to, both, register-transfer level (RTL) verification and test [3, 4].

However, the main issue with the BDDs and assignment decision diagrams is the fact that they allow logic or RTL modelling, respectively. 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 Modelling) and behavioural. 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, 6].

In this project, a major focus of the research carried out by Tallinn University of Technology lies in developing extensions to the HLDD model in order to support behavioural modelling and validation. These activities have been performed in the frame of task T2.5 "High-level decision diagrams" and the results are presented in current deliverable.

The deliverable is organised as follows. Section 2 presents the basic definitions of HLDD models and explains RTL modelling. Section 3 discusses extensions developed in the VERTIGO project to allow behavioural modelling. Also the corresponding design interfaces including HLDD optimisation techniques that have been implemented in the project are provided here. Section 4 explains HLDD extensions for assertion-based verification. This topic is divided into two parts. First, HLDD conversion from VHDL checkers obtained by IBM FoCs software is considered. Subsequently, direct conversion from PSL properties is discussed. The latter includes definition of the new, Temporally extended HLDD (T-HLDD) model. Section 5 explains HLDD model extensions developed in order to support code coverage analysis. Finally, conclusions are made. The deliverable contains appendixes presenting the file formats developed in this project for representing the HLDD extensions.

2. High-level decision-diagrams

This section presents the basic definition of High-Level Decision Diagram (HLDD) models and describes representation of digital systems by HLDDs.

2.1 Basic Definitions

High-Level Decision Diagrams (HLDD) are graph representation of discrete functions. A discrete function $y = f(x)$, where $y = (y_1, \dots, y_n)$ and $x = (x_1, \dots, x_m)$ are vectors is defined on $X = X_1 \times \dots \times X_m$ with values $y \in Y = Y_1 \times \dots \times Y_n$, 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 High-Level Decision Diagram (HLDD) is a directed non-cyclic labelled graph that can be defined as a quadruple $G=(M,E,X,D)$, where M is a finite set of vertices (referred to as *nodes*), E is a finite set of *edges*, X is a function which defines the *variables labelling the nodes* and the variable domains, and D is a function on E .

The function $X(m_i)$ returns a pair (x_i, X_i) , where x_i is the variable letter which is labelling node m_i and X_i is the domain of x_i . Each node of a HLDD is labelled by a variable. A single variable can label multiple nodes. In special cases of HLDDs, nodes are labelled by constants, arithmetic expressions or vectors.

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 . Graphical interpretation of e is an edge leading from node m_1 to node m_2 . It is said that m_1 is a *predecessor node* of m_2 , and m_2 is a *successor node* of the node m_1 , respectively.

D is a function on E representing the activating conditions of the edges for the simulating procedures. The value of $D(e)$ is a subset of X_i , where $e = (m_i, m_j)$ and $X(m_i) = (x_i, X_i)$. It is required that $Pm_i = \{D(e) \mid e = (m_i, m_j) \in E\}$ is a partition of the set X_i . In other words, the subsets of the set X_i labelled on the edges starting from a node m_i must not overlap and their union must be equal to X_i .

$G_y=(M,E,X,D)$,
 $M=\{m_1, m_2, m_3, m_4, m_5\}$;
 $E=\{e_1, e_2, e_3, e_4, e_5\}$, $e_1=(m_1, m_2)$, $e_2=(m_1, m_4)$,
 $e_3=(m_1, m_5)$, $e_4=(m_2, m_3)$, $e_5=(m_2, m_4)$;
 $X(m_1)=X(m_5)=(x_2, \{0,1,2,\dots,7\})$,
 $X(m_2)=(x_3, \{0,1,2,3\})$, $X(m_3)=(x_4, \dots)$,
 $X(m_4)=(x_1, \dots)$;
 $D(e_1)=\{0\}$, $D(e_2)=\{1,2,3\}$, $D(e_3)=\{4,5,6,7\}$,
 $D(e_4)=\{2\}$, $D(e_5)=\{0,1,3\}$.

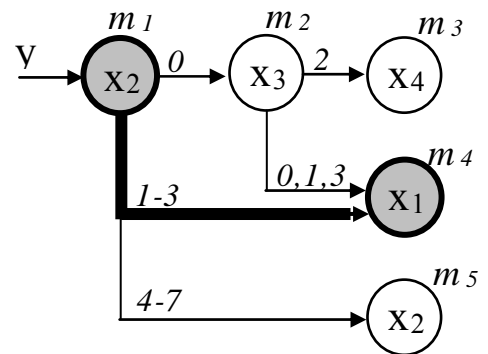


Figure 1. A HLDD for a function $y=f(x_1, x_2, x_3, x_4)$

HLDD has only one starting node (*root node*) m_0 , for which there are no preceding nodes. The nodes, for which successor nodes are missing, are referred to as *terminal nodes* M^T . Simulation on decision diagrams takes place as follows. Consider a situation, where all the node variables are fixed to some value. According to these values, for each non-terminal node a certain output edge will be chosen which enters into its corresponding successor node. Let us call such connections *activated edges* under the given values. Succeeding each other, activated edges form in turn *activated paths*. For each combination of values of all the node variables there exists always a corresponding activated path from the root node to some terminal node. We refer to this path as the *main activated path*. The simulated value of variable represented by the HLDD will be the value of the variable labelling the terminal node of the main activated path. Figure 1 presents an example of a graphical interpretation of a HLDD.

Let us explain the HLDD simulation process on the diagram example presented in Figure 1. Assuming that variable x_2 is equal to 2, a path (marked by bold arrows) is activated from node m_1 (the root node) to a terminal node m_4 labeled by x_1 . Let the value of variable x_1 be 4, thus, $y=x_1=4$. Note that this type of simulation is inherently event-driven since we have to simulate only those nodes that are traversed by the main activated path (marked by grey colour in Figure 1).

When representing systems by decision diagram models, in general case, a network of HLDDs rather than a single DD is required. During the simulation in HLDD systems, the values of some variables labelling the nodes of a HLDD are calculated by other HLDDs of the system.

In this project, we apply HLDDs as a graph representation of RTL. There exist other word-level decision diagrams such as multiterminal DDs (MTDDs) [7], K*BMDs [8] and ADDs [2]. However, in MTDDs the nonterminal nodes hold Boolean variables only. K*BMDs, where additive and multiplicative weights label the edges are useful for compact canonical representation of functions on integers (especially wide integers). However, the main goal of HLDD representations considered in the VERTIGO project is not canonicity but simulation and implications. The principal difference between HLDDs and ADDs lies in the fact that ADDs edges are not labelled by activating values. They are rather used as connecting signals to represent structure. In HLDDs, the selection of a node activates a path through the diagram, which derives the needed value assignments for variables. Furthermore, ADD model includes four types of nodes (read, write, operator, assignment decision). In HLDD the nodes are divided into nonterminal (control) and terminal (data) ones. The following Subsection describes how HLDDs can be used for representing register-transfer level circuits.

2.2 Modelling RTL circuits

At the RT-level, the design is assumed to be partitioned into a datapath and a control part. Figure 2 explains this type of architecture. Here, the control part is a Finite State Machine (FSM) consisting of a state register (represented by variable x_S in the corresponding HLDD model), next state logic and output logic. Input signals for the FSM include the primary inputs of the design (variables x_I), conditional status bit signals originating from the datapath (variables x_N) and current value of the state variable x_S . Outputs of the FSM include the primary outputs of the design (variables x_O), control signals (variables x_C) and the next value of x_S .

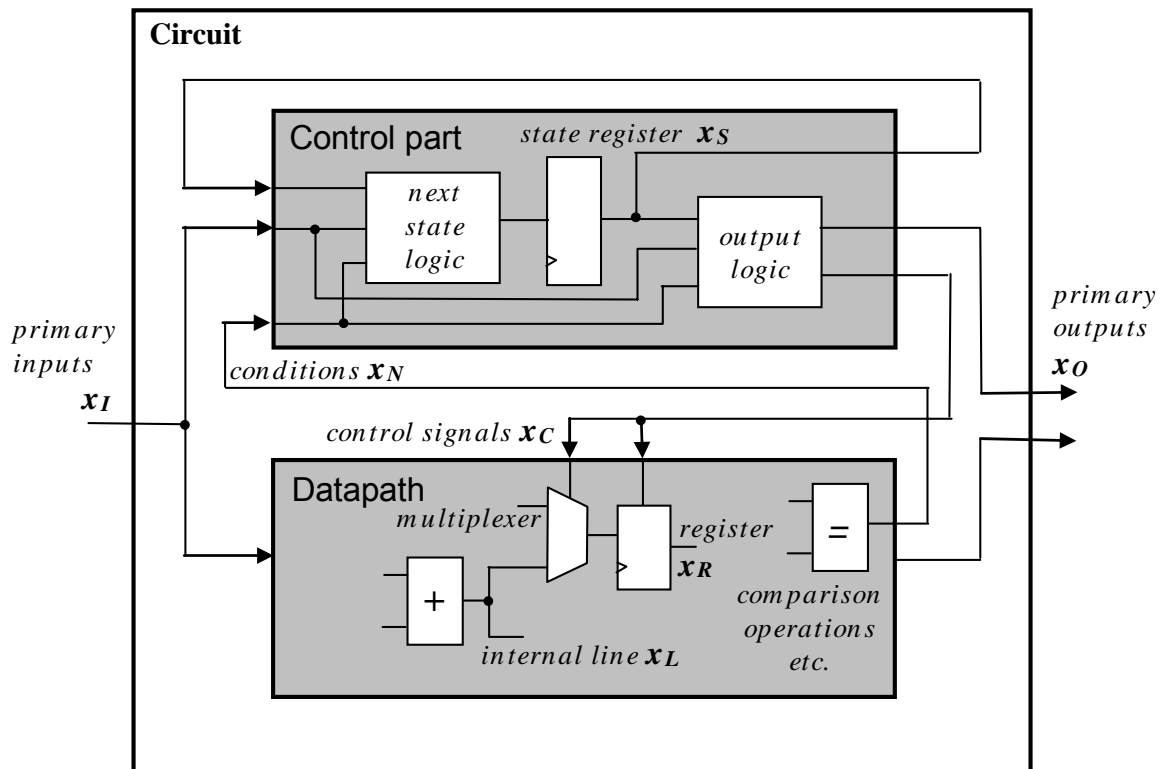


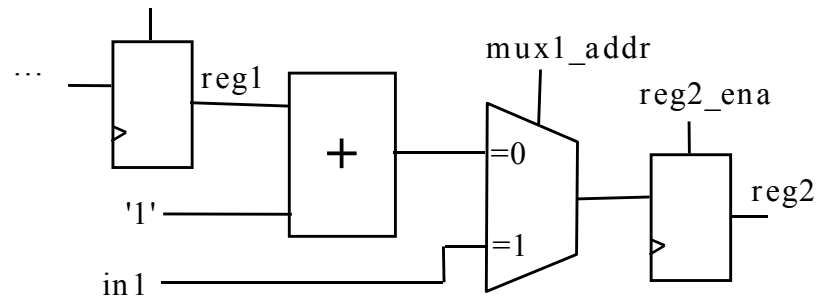
Figure 2. Register-transfer level view of a digital circuit

Datapath can be viewed as a network consisting of modules, or blocks. These include registers, multiplexers and functional units (for implementing operations). All the registers and some internal lines of the datapath can be represented by variables in the RTL DD model (variables x_R and x_L , respectively). Inputs for the datapath are the primary inputs x_I and control signals x_C (e.g. multiplexer addresses and register enable signals). Outputs are the primary outputs x_O as well as conditional signals x_N (e.g. from comparison operators) leading to the control part FSM. In the following we will explain how both of these parts can be represented by means of decision diagrams.

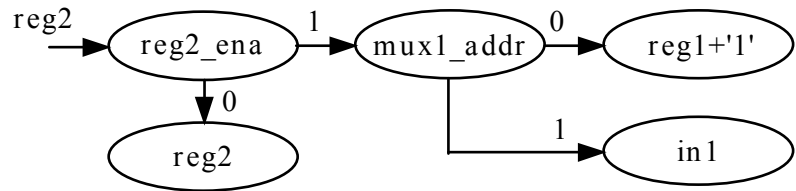
In HLDD models representing the datapath, the non-terminal nodes correspond to control signals (labelled by variables x_C). The terminal nodes represent operations (functional units). Register transfers and constant assignments are treated as special cases of operations. Figure 3 shows a simple example of a HLDD representation for a datapath fragment.

At the RT-level, datapath is generally represented by a system of HLDDs. Here, different partitioning strategies are possible. In the test generation approach presented in this paper we use partitioning, where for each primary output, fanout signal and register a HLDD corresponds. In addition, multiplexers that are connected to an input of an FU are represented by a separate HLDD. Figure 5 shows this type of HLDD system partitioning for the datapath given in Figure 4.

However, it is possible to use alternative partitionings. For example, Figure 6 shows an approach, where for each register of the datapath exactly one decision diagram corresponds. We refer to this type of partitioning as Register-Oriented HLDDs (RODD). Choice between the types of partitionings is a matter of trade-off. The first partitioning (Fig. 5) preserves the datapath structure while RODD partitioning (Fig. 6) is convenient for cycle-based modelling.



a)



b)

Figure 3. A datapath fragment a) and its HLDD representation b)

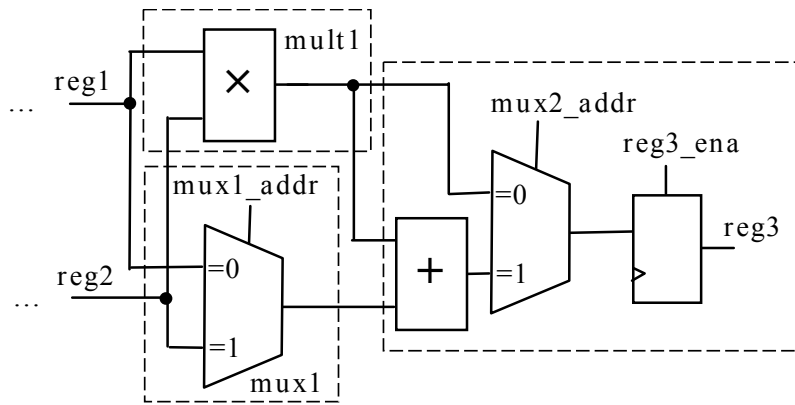


Figure 4. Partitioning datapath into HLDDs

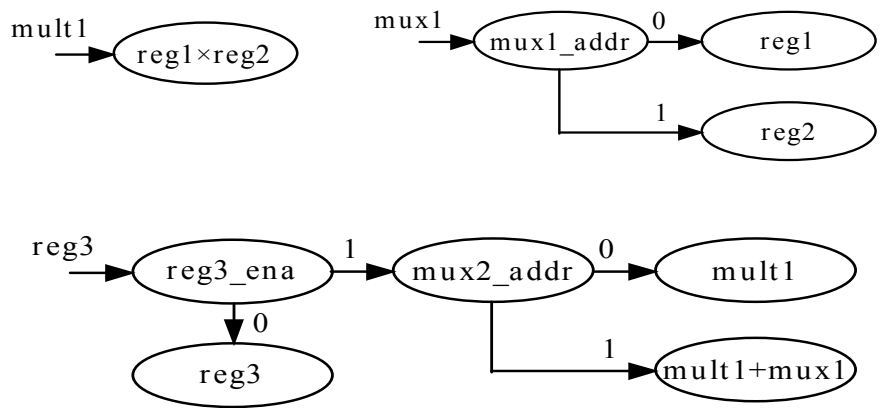


Figure 5. System of HLDDs representing the datapath in Fig. 4

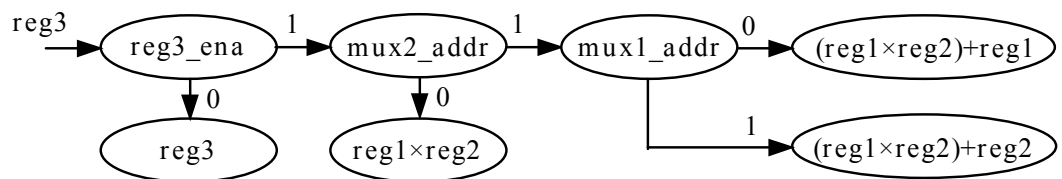


Figure 6. RODD representation of the datapath in Fig. 4

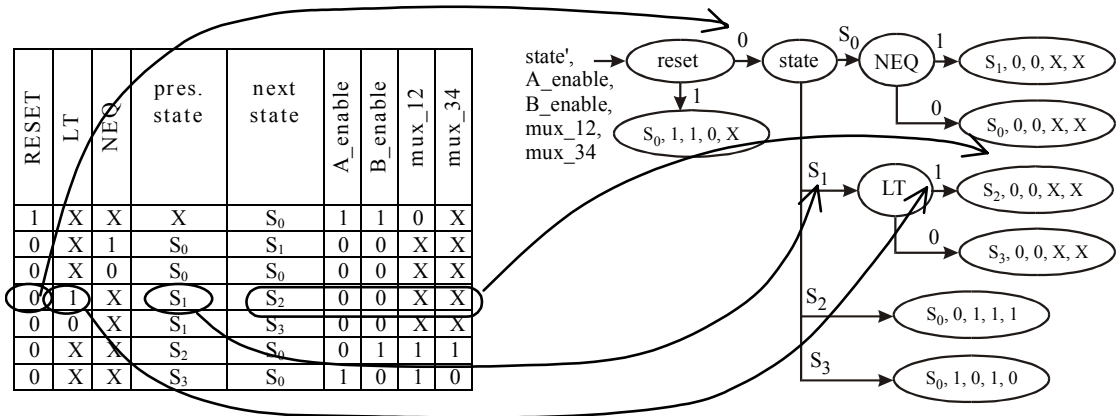


Figure 7. Converting FSM state table into HLDD

Similar to the datapath, the control part of an RTL design can be represented by a HLDD. This HLDD calculates the values for a vector consisting of the state variable and control signals. In the HLDD, the non-terminal nodes correspond to current state (labelled by variable x_s) and conditional signals originating from the datapath (variables x_N). Terminal nodes hold vectors with the values of next state and control signals x_C .

Figure 7 shows an FSM state table and its corresponding HLDD representation. In the DD, $state'$ denotes the next state and $state$ denotes the current state value. Variables A_enable , B_enable , mux_12 and mux_34 are FSM outputs and belong to the control signals x_C . Variables $RESET$, LT and NEQ are FSM inputs and belong to x_N . The black circles and arrows in Figure 7 present the correspondence between the edges and the terminal node for the fourth row of the state table.

3. HLDD extensions for behavioural modelling

This section presents extensions developed for HLDDs to allow behavioural modelling. In addition, the corresponding design interfaces including HLDD optimisation techniques that have been implemented in the project are listed.

3.1 Behavioural HLDD modelling

The traditional RTL HLDD model as described in Section 2 assumes design to be partitioned into a control part and a datapath. The types of variables (signals) in the model include primary input variables x_i , primary output variables x_o , data registers x_R , state variable x_s , status bit variables x_N , and control signals x_C . In the VERTIGO project, the model was generalized for the behavioural abstraction level. We still consider a cycle-accurate model. However, binding of operations to functional units is not required and the system can contain an arbitrary number of finite state machines that are handled in a uniform manner with data. The signal types are restricted to three: primary input variables x_i , primary output variables x_o , and the internal signals x_R .

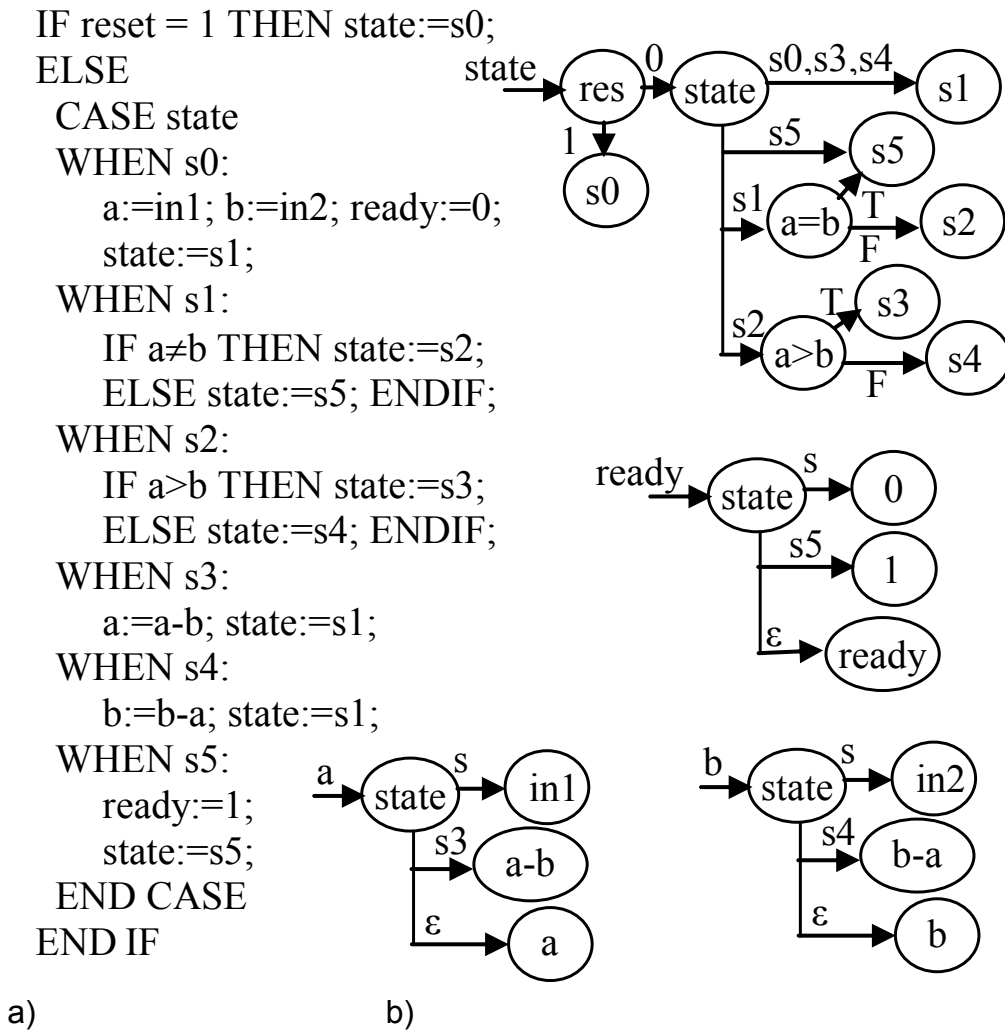


Figure. 8. a) VHDL and b) its behavioural HLDD model

Consider the VHDL fragment in Fig. 8a implementing the greatest common divisor algorithm and its corresponding behaviourally extended HLDD depicted in Fig. 8b. Note, that the control signal, status bit variables have been omitted and also the control unit state variable is handled uniformly with data registers. The example contains a single FSM but in fact a network of FSMs can be easily represented by behavioural HLDDs. In the Figure, T and F stand for true and false, respectively. The ϵ character denotes default edges.

3.2 Behavioural HLDD simulation

We have developed an algorithm supporting, both, Register-Transfer Level (RTL) and behavioural design abstraction levels. In the RTL style, the algorithm takes the previous time step value of variable x_j labelling 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 behavioural HDL coding style HLDDs are generated and ranked in a simulation order to ensure causality. For variables x_j labelling 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, behavioural and mixed HDL description styles.

Algorithm 1. RTL/behavioural simulation on HLDDs

```
For each diagram G in the model
  mCurrent = m0
  Let xCurrent be the variable labelling mCurrent
  While mCurrent is not a terminal node
    If is xCurrent clocked or its DD is ranked after G then
      Value = previous time-step value of xCurrent
    Else
      Value = present time-step value of xCurrent
    End if
    If Value ∈ D(eactive), eactive = ( mCurrent, mNext) then
      mCurrent = mNext
    End if
  End whi
  Assign xCurrent to the DD variable xG
End for
```

3.3 Hardware design language interfaces

A set of hardware description language interfaces supporting HLDDs have been implemented in the VERTIGO project. Phase1 tool from UNIVR is used in order to generate HIF, the central format of VERTIGO tools, from SystemC and VHDL. The HLDD trees generated by Phase1 are then collapsed by HIF2AGM tool and converted into AGM format, which is the internal format to represent HLDD models in TUT VERTIGO tools. Usage of these interfaces has been explained in detail in Section 7 of Deliverable D2.2.

The main HLDD interface is accompanied by three other utilities that have also been implemented in the project. BEH2RTL is a tool that converts behaviourally extended HLDDs into the “old” RTL style HLDD format. This is necessary for backward compatibility with the TUT tools that existed before VERTIGO. Also two direct VHDL interfaces for specific VHDL subsets have been implemented. One for behavioural subset (BEH2AGM) and another for the RTL subset (RTL2AGM). Fig. 9. shows the HLDD generation flow for RTL and behavioural HLDDs.

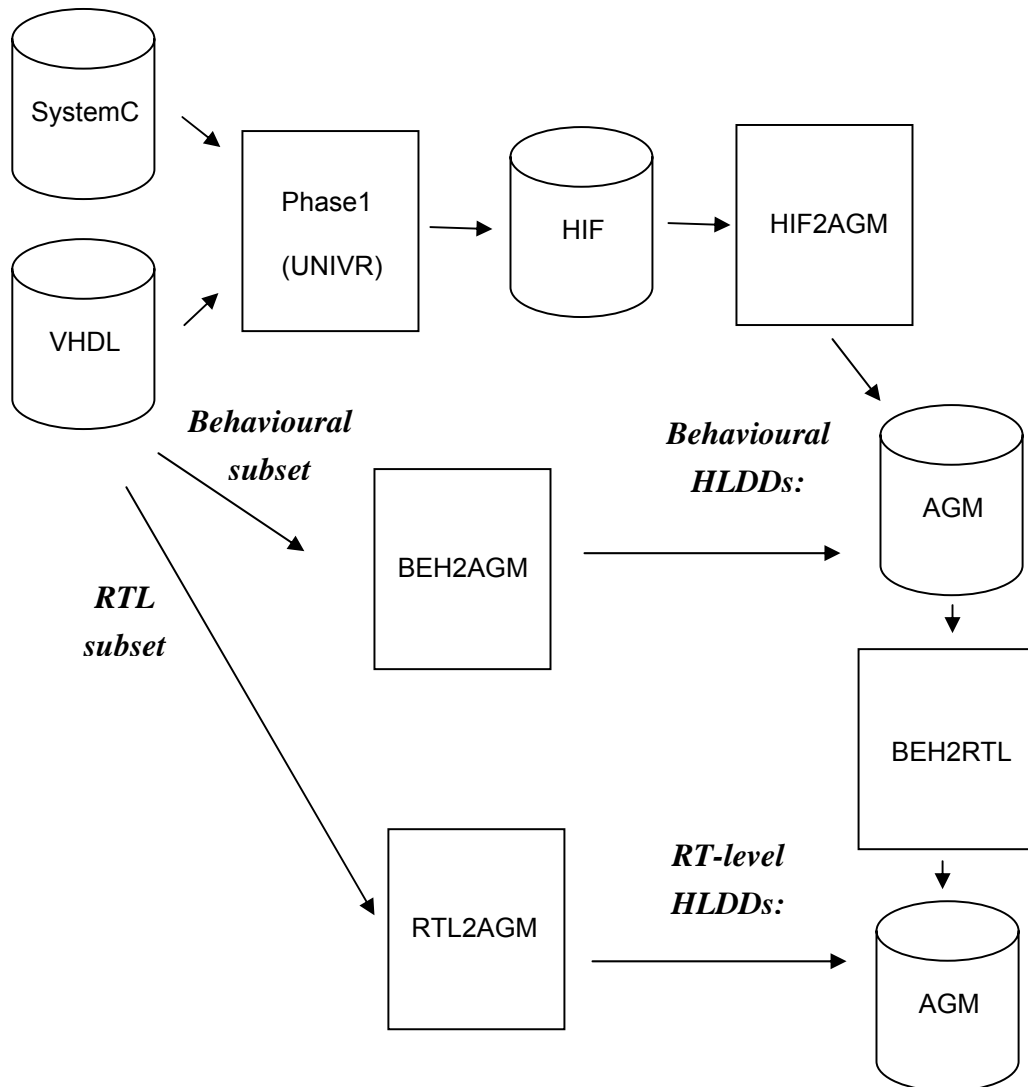


Figure 9. HLDD generation flow for RTL and behavioural HLDDs.

4. HLDD extensions for assertion-based verification

Assertion-based Verification can be classified as Design-for-Verifiability (DFV) technique. The goal is to assist both formal methods and simulation-based verification and allow discovering Design under Verification (DUV) misbehaviour (causing an assertion violation) earlier and more effective. Another important advantage of ABV is its aid to debug process.

In case of dynamic verification assertions provide better *observability* on the design what allows detecting bugs earlier and closer to their origin. At the same time in case of static verification with model checking, the assertions increase the *controllability* of the design and direct verification to the area of interest. Each assertion violation discovered by model checking is reported as a counter-example.

The question of the origin of assertions can be formulated as a separate topic for research itself. An important aspect here is the problem of *completeness*. Usually assertions do not describe all the possible properties of design what would mean translation of a complete design specification to a formal assertion description language such as PSL (Property Specification Language) or SVA (System Verilog Assertions). Instead of this only design areas of concern, sometimes referred as *verification hot spots*, are targeted. In practice they are often provided by design engineer and require deep knowledge of the DUV behaviour.

In this Section, we discuss HLDD extensions for assertion-based verification. The topic is divided into two parts. First, HLDD conversion from VHDL checkers obtained by IBM FoCs software is explained. Then, direct conversion from PSL properties into a new representation, Temporally extended HLDD (T-HLDD) model, is presented.

4.1 PSL subset for HLDD generation

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].

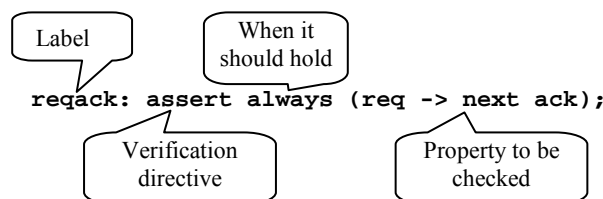


Figure 10. PSL property reqack.

An example PSL property `reqack` structure is shown in Figure 10. Its Timing diagram is illustrated by Figure 11a. 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 11b. Figure 11c shows the case when the property was not activated.

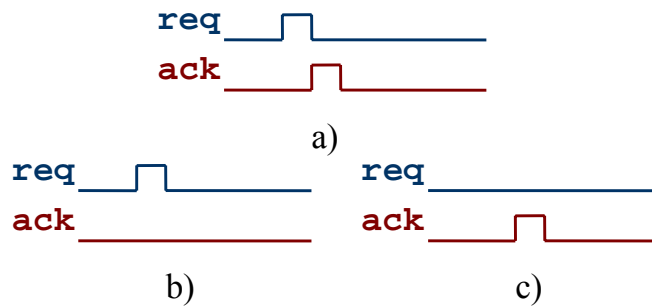


Figure 11. Timing diagrams for the property “reqack”

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. PSL is also a multi-layered language. The layers include:

- *Boolean layer* – the lowest one, consists of boolean expressions in HLD (e.g. $a \ \&\&(b \ || \ c)$)
- *Temporal later* – 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.
- *Modelling 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 this part of PSL is left for future work related to HLDD-based model checking implementation. In this project we will consider only FL part of PSL. However, only a subset of FL is applicable for translation to HLDD assertions.

The certain advantage of PSL is the ability to specify any desired property in many ways and flavours provided for the designer’s convenience. However the support for an entire language subset is not always reasonable or mandatory. Our goal, and also the VERTIGO project requirement, is the support for FL subset known as PSL Simple Subset. This subset is gaining its popularity and is supported by many verification and simulation tools. It 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.

The following two subsections will describe two approaches for PSL properties conversion to HLDD graphs. The first one implies commercial tool IBM FoCs and has support for the entire PSL Simple Subset. The second approach considers a temporal extension for HLDDs and therefore the supported PSL subset applies several additional constraints for the PSL Simple Subset. For this approach we use only VHDL flavour of PSL and only weak FL operators. The event-driven versions of the temporal LTL style operators such as *next_event* are also postponed for the future work.

4.2 Converting VHDL checkers to HLDDs

This Subsection describes PSL properties conversion to HLDD simulation monitors, implying FoCs by IBM for an intermediate step of VHDL checkers generation. To start, let us first consider an example of a PSL assertion p as given below:

```
p: assert always ({a; [*2] ;b} | => {c})
```

The precondition of this assertion is the sequence of system behaviour when at the beginning a becomes high, followed by a whatever-sequence 2 clock cycles long and then b becoming high. This precondition activates the assertion and requires c to become high just after it (non-overlapping implication) in order for assertion to be satisfied. The VHDL of the checker generated by FoCs has the following form (Figure 12).

```
PROCESS (clk)
BEGIN
  IF ( ( clk = '1' ) ) THEN
    focs_ok <=
      ( focs_vout(4) OR NOT( c ) ) ;
  ELSE
    focs_ok <= '1' ;
  END IF;
END PROCESS;

PROCESS
...
VARIABLE focs_vout : std_logic_vector(4 DOWNT0 0);
BEGIN
  WAIT UNTIL (clk'EVENT AND clk = '1');
  ...
  focs_vout(4 DOWNT0 0) := reverse( ( ( ( ( ( (
    focs_v(0) AND a ) ) & ( ( focs_v(1) AND '1' )
  ) ) & ( ( focs_v(2) AND '1' ) ) ) ) & ( (
    focs_v(3) AND b ) ) ) & ( ( focs_v(4) AND
    NOT( c ) ) ) ) ) );
  ...
END PROCESS;
```

Figure 12. VHDL describing the checker for assertion p

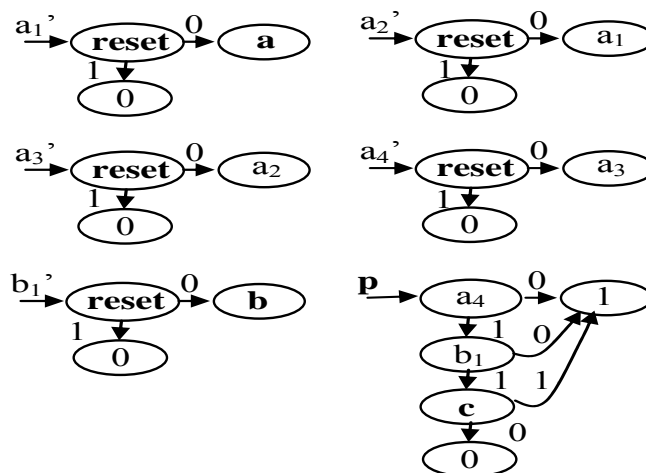


Figure 13. HLDD for the assertion checker VHDL in Figure 5

The resulting VHDL code can be converted to HLDD graphs and added on top of the design under verification (DUV) as shown in Figure 13. In the Figure we used a notation where trailing quote character after diagram variable denotes one clock cycle delay. The HLDD variables corresponding to the inputs and outputs of the checker (i.e. reset, a, b, c and p) are shown by bold font. Generation of respective HLDDs from similar checkers described in VHDL can be easily automated.

The HLDD of the checker would be seamlessly integrated and added on top of the design under verification (See Figure 14). This allows uniform model representation for, both, the DUV and the assertion checker. Simulation of such integrated HLDD model is expected to speed up the assertion checking process considerably [10].

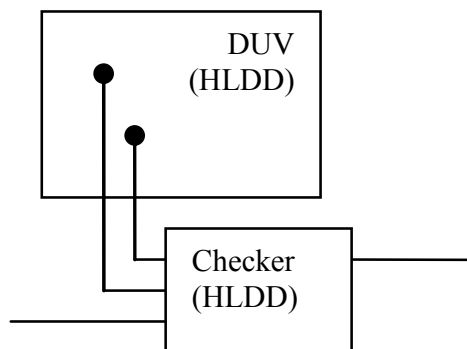


Figure 14. PSL assertion integration to HLDD model

Converting VHDL assertion checkers obtained by FoCs into HLDD models as shown in Fig. 13 is straightforward and allows usage of converters developed in VERTIGO project (See Section 3) [14]. However, the resulting HLDDs could be large and therefore there is a need for representing PSL properties directly. In the following Subsection we discuss an extension to HLDD model, called Temporally extended HLDDs (T-HLDD), which are specially designed to include temporal information from properties.

4.3 Temporally extended HLDDs

The idea of the proposed method of generating Temporally extended HLDDs (T-HLDD) 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 (T-HLDD) for a complex property using the PPG Library elements.

Prior to the T-HLDD 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, FAILED and PASSED, respectively), as opposed to an arbitrary number of terminal nodes in usual HLDD graph. The standard PPG interface is shown in Figure 15.

The terminal nodes in PPG have the following meaning:

- CHECKING – the property is being calculated but with no result yet
- FAILED – the property has been activated and doesn't hold
- PASSED – the property has been activated and holds

In this project, we support only weak operators. In order to extend the subset to support strong operators a third output PENDING would be needed. Its addition would influence the proposed modification of the Simulator as explained at the end of this Subsection. Example PPGs created for 4 PSL operators are shown in Figure 16.

One of the main motivations for PSL introduction was 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 extension for HLDD

PSL construct	Explanation	Equivalent HLDD extension for a Variable
next[n]	property holds at time step n	Variable'[n..n]_e
next_a[j:k]	property holds at all time steps within j to k range	Variable'[j..k]_a
next_e[j:k]	property holds at least once within j to k time steps range	Variable'[j..k]_e

Additionally we introduce the notion of END as a special case of value of k in the expression Variable'[j..k]_sfx (where _sfx is one of _a or _e). The maximum bound of the time steps sequence may take value END if it is not explicitly determined. The time point END occurs at the end of simulation and implicitly determined by:

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

The main purpose of the proposed temporal extension is transferring additional information and directives to the HLDD Simulator that will check assertions. Let us refer to the HLDD graphs with the described extension as Temporally Extended HLDDs (T-HLDD).

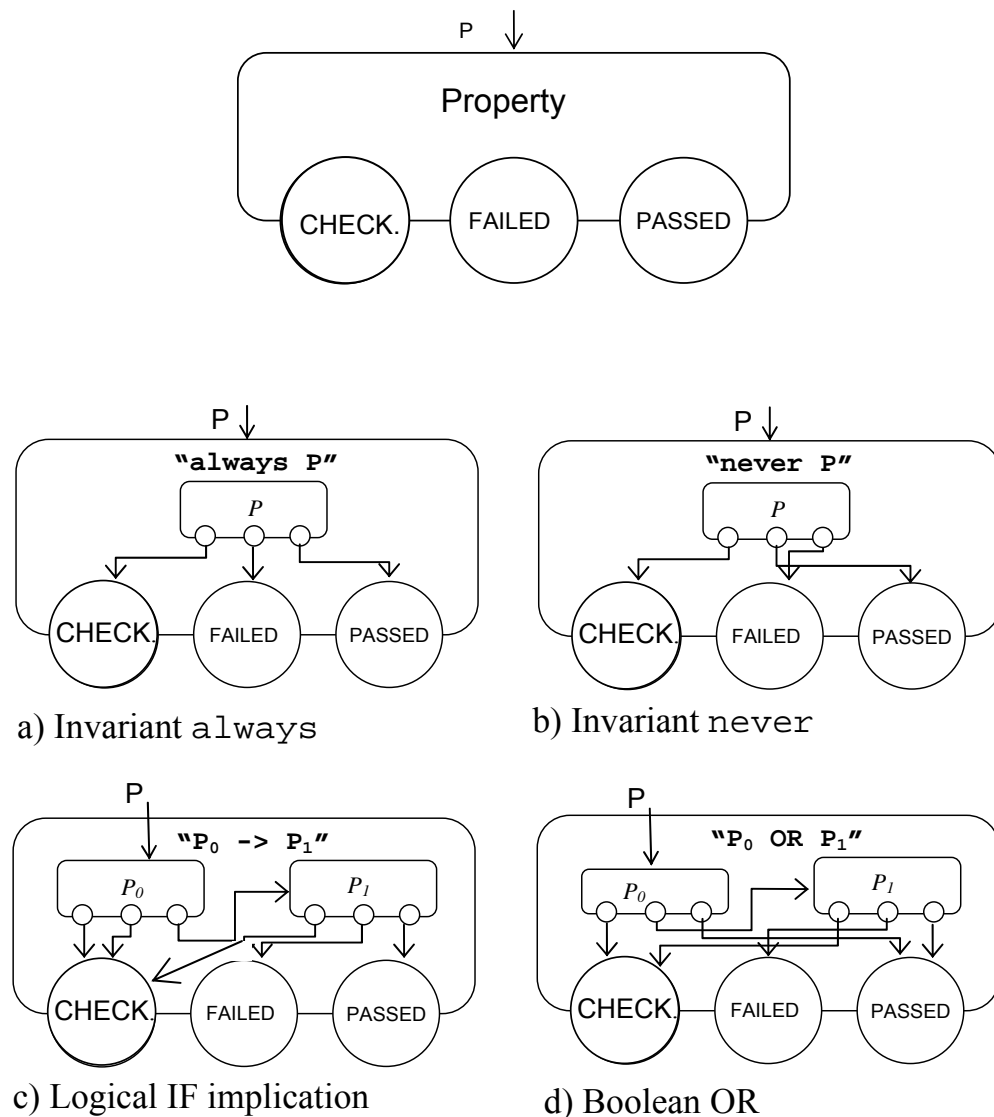


Figure 16. PPGs for a set of PSL operators

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 P1:

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

The resulting T-HLDD graph describing this property is shown in Figure 17.

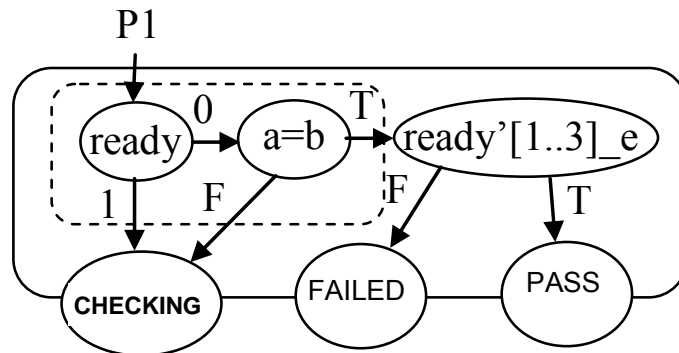


Figure 17. T-HLDD for property P1

In order to understand assertion checking on T-HLDDs consider the PSL assertion example P1 provided in Figure 17. The assertion represents a property to be checked against the GCD implementation given in Figure 8 (See Section 3!). It states that always when ready is low and a is equal to b then after 1 to 3 time-steps (clock cycles) ready will be asserted.

Let us introduce the concept of minimal and maximal time-window of an assertion. We say that a minimal (maximal, resp.) time window is the minimum (maximum) number of time steps from current step to the end step when assertion has to be evaluated. Assertion's minimal and maximal time-window is denoted by w_{min} and w_{max} , respectively. For example, the assertion shown in Fig. 17 have time windows of $w_{min} = 2$ and $w_{max} = 4$ because its evaluation starts at current time moment and ends 1-3 time-steps later. Boolean expressions have a time window of 1. Moreover, operators until, before and eventually! also have $w_{min} = w_{max} = 1$ because their calculation is performed using the temporal nodes presented in Table 1.

We have developed the following algorithm for HLDD based assertion checking:

Algorithm 2. Checking assertion P with HLDD simulation

```

For time-step k = 1 ... end-cycle
  For interval w =  $w_{min}$  ...  $w_{max}$ 
    For time-step j = k ... k+w-1
      Simulate T-HLDD of P at time-step j /* See Algorithm 1! */
    End for
  End for
End for
    
```

The advantage of Algorithms 2 is that they require very little memory: only slightly more than a high-level simulator. However, event-driven evaluation of assertions to speed up the checking process has not been considered in current work and there lies the focus of our future research. While there already exist commercial assertion checking tools taking advantage of events the core benefits of developing an event-driven HLDD based solution lies in fast evaluation by graph traversal and for easy identification of cause-effect relationships provided by the model that is especially useful in debug and diagnosis.

Currently the T-HLDD based assertion checking is not yet implemented. However, we have compared HLDD model simulation (Algorithm 1) to a state-of-the-art HDL simulator. Table 2 presents run-times of the HLDD models and corresponding VHDL models simulation. The simulators compared include the HLDD-based simulator [9] and an efficient commercial cycle-based HDL simulation tool Cyclone (Synopsys). The experiment was run on a 366 MHz SUN UltraSPARC 60 workstation with 512 MB RAM under Solaris 2.5.1 operating system. During the experiments, real test stimuli generated by test generator DECIDER [6] were used in order to activate all possible states of the circuit behavior (in contrast to

random simulation vectors, which in reality do not allow to simulate all possible behaviors). In order to achieve a better timing resolution all the test sets were multiplied by ten. For optimal performance, Synopsys tools cyslab and cysim were run with -perf and -2state options. The decision diagram event-driven cycle-based simulation tool implementation offers the gain in simulation time between 2.5 and more than 14 times in comparison to the cycle-based HDL simulator.

Table 2. HLDD and HDL-based simulation comparison

Circuit	Simulation time [s]		Ratio
	HLDD simulation	HDL cycle-based	HDL/HLDD
gcd	0.20	0.51	2.55
mult8x8	0.32	1.00	3.13
diffeq	0.25	1.26	5.04
huff_enc	0.34	1.40	4.12
circ1	0.14	2.05	14.64

5. HLDD extensions for code coverage analysis

Over the previous 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 etc. The statement coverage metric measures the number of times every instruction is exercised by the program stimuli. Toggle coverage shows whether and how many times nodes in the design toggle, i.e. how many bits change their state from 0 to 1 or vice versa. In the case of branch coverage, we measure the number of times each branch in the control flow graph of the code is taken or not taken under the set of program stimuli. Path coverage shows the number of times every path in the control flowgraph is exercised by the set of program stimuli. A potential goal of software testing is to have 100% path coverage, which implies branch and line coverage. However, full path coverage is a very stringent requirement as the number of paths in a program may be exponentially related to program size.

In this project, we present an extension to High-Level Decision Diagrams (HLDD) model for efficient code coverage analysis and show how those classical coverage metrics map to HLDD constructs. We show that a speed-up of several orders of magnitude is achieved by HLDD based tool in comparison to a state-of-the-art commercial simulator. In order to analyze quality of verification of hardware designs translated to HLDDs, three traditional coverage metrics were chosen and built in to the HLDD based simulation tool. These include statement coverage, branch coverage and toggle coverage.

The statement coverage maps directly to the ratio of nodes traversed during the HLDD simulation presented in Section 3. For example, see Fig. 18 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 than 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. 18 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 activated in the HLDD simulation process constitutes to HLDD branch coverage.

HLDD toggle coverage is calculated similarly to traditional HDL toggle coverage. However, in this project 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.

In order to understand code coverage analysis on HLDD models consider the example in Fig. 18. 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 18. The statement `RMAX := DATA_IN` is represented by the terminal node surrounded by bold circle in the corresponding HLDD. The appropriate code coverage values are calculated from the ratio of corresponding items traversed during HLDD simulation with respect to all items.

To achieve this goal, three extensions to existing HLDD data structure were introduced:

1. Addition of a traverse flag to the node structure of the HLDD model;
2. Addition of a traverse flag to the edge structure of the HLDD model;
3. Addition of 0- and 1-bitmasks to the node structure of the HLDD model for toggle coverage analysis.

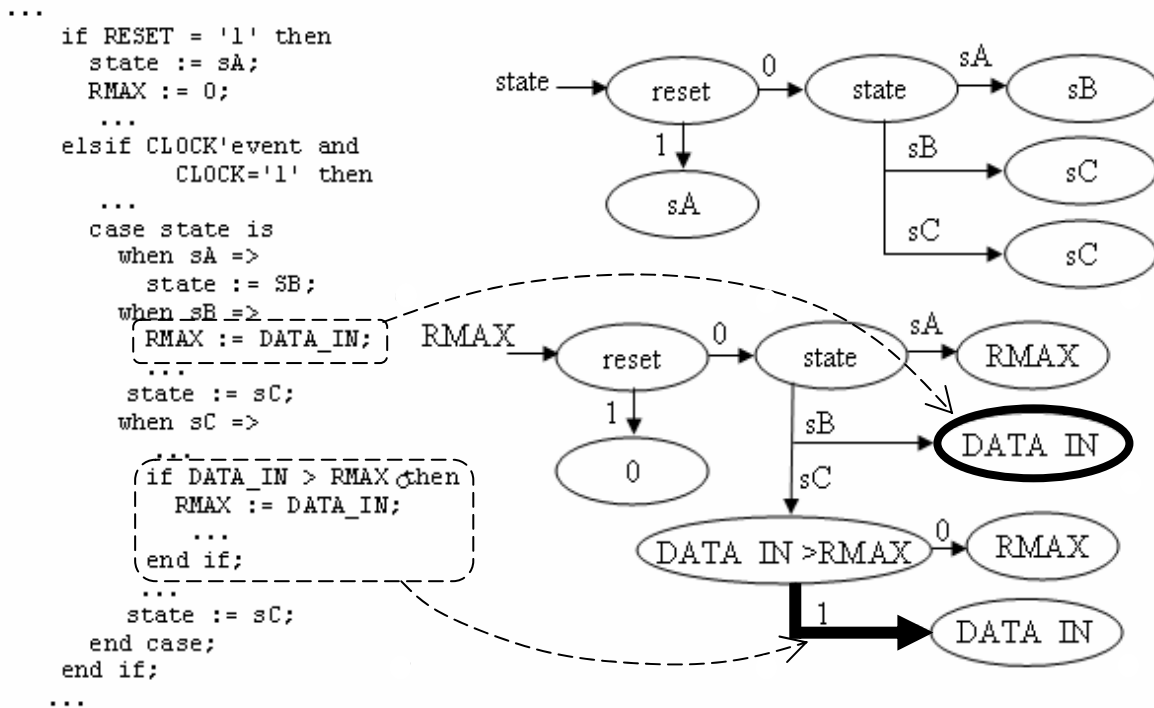


Figure 18. Mapping code coverage to HLDD constructs

Comparative experiments between the HLDD-based code coverage analysis approach and a popular HDL commercial simulation tool were carried out. Table 3 presents the circuits chosen from the ITC99 benchmark family to run the experiments. Table 4 shows the comparison between traditional code coverage assessment (statement, branch and toggle coverage) carried out by a state-of-the-art commercial HDL simulator Modelsim by Mentor Graphics and by the HLDD-based simulator (implementing node, edge, toggle coverage, respectively). The code has been exercised using random set of stimuli of different length.

Previous experiments comparing HLDD simulation times to those of state-of-the-art commercial simulators have shown that it outperforms the commercial event-driven tools by a factor of 10 [5] and their cycle-based counterparts by a factor of 3-4 [9]. In the code coverage measurement experiments we detected an additional speed-up, which ranged from 55 to nearly 1500. Average code coverage evaluation speed-up achieved by the HLDD simulator was more than 300 times for the given set of benchmarks.

This considerable improvement in run times is mainly due to two reasons. First, traditional code coverage items have one-to-one mapping to HLDD structural constructs traversed during the simulation procedure allowing seamless code coverage analysis. Second, HDL based commercial simulators are forced to break internal optimizations when considering code coverage, thus, resulting in performance loss.

Table 3. ITC99 benchmark circuits

design	# of lines	# of inputs	# of outputs	# of signals	# of HLDD nodes
b00	76	4	2	7	37
b04	84	6	1	14	58
b09	102	4	1	9	44
b10	169	10	3	14	116

Table 4. Comparison of traditional and HLDD code coverage measurement execution times

design (1)	test length (2)	Commercial HDL simulator			HLDD simulator			HLDD speed-up, in times (4)/(7)
		simulation time, s		ratio (4)/(3) (5)	simulation time, s		ratio (7)/(6) (8)	
		w/o coverage (3)	w coverage (4)		w/o coverage (6)	w coverage (7)		
b00	50	57	211	3.7	1.0	1.6	1.6	131.9
	100	59	238	4.0	1.6	3.0	1.9	79.3
	500	60	693	11.5	1.6	3.0	1.9	231.0
	1000	62	1839	29.4	3.2	6.2	1.9	296.6
b04	50	58	223	3.8	1.6	3.2	2.0	69.7
	100	58	287	4.9	1.6	3.2	2.0	89.7
	500	61	1104	18.1	3.0	6.2	2.1	178.1
	1000	65	3568	54.6	3.0	6.4	2.1	557.5
b09	50	53	177	3.3	1.0	3.2	3.2	55.3
	100	58	197	3.4	1.6	3.2	2.0	61.6
	500	58	269	4.6	1.6	3.2	2.0	84.1
	1000	58	301	5.2	1.6	3.2	2.0	94.1
b10	100	42	432	10.2	1.6	3.2	2.0	135.0
	500	56	2721	48.3	3.0	6.4	2.1	425.2
	1000	59	9528	162.7	3.2	6.4	2.0	1488.8
	5000	92	23262	253.4	15.6	18.8	1.2	1237.3

6. Conclusions

The report presented HLDD extensions developed in the frame of the VERTIGO project task T2.5 "High-level decision diagrams". As a result of this activity the following extensions to the basic HLDD model were implemented:

- Extensions were developed for HLDD representations to allow behavioural modelling. In addition, the corresponding design interfaces (from HIF and VHDL) including new HLDD optimisation techniques were implemented.
- PSL properties conversion to HLDD simulation monitors has been implemented, using FoCs by IBM for an intermediate step of VHDL checkers generation.
- Additionally, a conversion technique from PSL language properties direct translation to HLDDs has been developed. For this purpose, an extension to HLDD data structure, called Temporally extended HLDD model has been proposed.
- A technique was implemented for mapping classical code coverage metrics to HLDD constructs. Experiments on ITC99 benchmark circuits indicated the feasibility of the proposed approach. Moreover, a speed-up of several orders of magnitude was achieved by HLDD based tool in comparison to a state-of-the-art commercial simulator.

7. References

- [1] R. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, C-35, 8:677-691, 1986
- [2] V. Chayakul, D. D. Gajski, L. Ramachandran, "High-Level Transformations for Minimizing Syntactic Variances", Proc. of ACM/IEEE DAC, pp. 413-418, June 1993.
- [3] I. Ghosh, M. Fujita, "Automatic Test Pattern Generation for Functional RTL Circuits Using Assignment Decision Diagrams", Proc. of ACM/IEEE DAC, pp. 43-48, 2000.
- [4] L. Zhang, I. Ghosh, M. Hsiao, "Efficient Sequential ATPG for Functional RTL Circuits", Int. Test Conf., pp.290-298, 2003.
- [5] Raimund Ubar, Adam Morawiec, Jaan Raik. Cycle-based Simulation with Decision Diagrams, Proceedings of the DATE Conference, pp. 454-458, 1999.
- [6] J. Raik, R. Ubar, Fast Test Generation for Sequential Circuits Using Decision Diagrams Representations. Journal of Electronic Testing: Theory and Applications 16, Kluwer Academic Publisher, 2000, pp. 213-226.
- [7] E. Clarke, M. Fujita, P. McGeer, K.L. McMillan, J. Yang, X. Zhao, „Multi terminal BDDs: an efficient data structure for matrix representation“, Proc. of Int'l Workshop on Logic Synth., pp. P6a:1-15, 1993.
- [8] R. Drechsler, B. Becker, S. Ruppertz, „K*BMDs: a new data structure for verification“, Proc. of European Design & Test Conf., pp. 2-8, 1996.
- [9] 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.
- [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] Jaan Raik, Raimund Ubar, Taavi Viilukas, Maksim Jenihhin. Mixed Hierarchical-Functional Fault Models for Targeting Sequential Cores. Elsevier Journal of Systems Architecture, 2008.
- [13] Giuseppe Di Guglielmo, Franco Fummi, Maksim Jenihhin, Graziano Pravadelli, Jaan Raik, Raimund Ubar. On the Combined Use of HLDDs and EFSMs for Functional ATPG. 5th IEEE East-West Design & Test Symposium (EWDTS), Yerevan, Armenia, September 7-10, 2007.
- [14] Maksim Jenihhin, Jaan Raik, Anton Chepurov, Raimund Ubar. Assertion Checking with PSL and High-Level Decision Diagrams. Proceedings of the IEEE 8th Workshop on RTL and High Level Testing (WRTL'07), October 12-13, 2007, Beijing, P.R.China.