



# STREP – IST 033709

## VERTIGO

Verification & Validation of Embedded System Design  
Workbench

### Deliverable D3.2 – Tools for static and dynamic verification of IPs

DUE DATE 31 May 2008  
ACTUAL DATE 15 Jun 2008  
START OF PROJECT 01 June 2006 DURATION 30 months  
ABSTRACT Final status on the development of dynamic and static techniques for the verification of the correct system-level modelling and RTL synthesis of IP cores.  
AUTHOR, COMPANY S.Scholefield, TransEDA – S.Dellacherie, AerieLogic – J. Marques-Silva, SOTON – G.Pravadelli, UNIVR – J.Raik, TUT – Z.Peng, LIU – U.Rossi, STM  
WORKPACKAGE/TASK WP3/T3.3  
FILING CODE VERTIGO/Deliverables/08\_D3.2\_VERTIGO\_R1

#### DOCUMENT HISTORY

Release	Date	Reason of change	Status	Distribution
1	23/05/2008	first version	Draft	CO
2	29/05/2008	added AerieLogic contribution	intermediate	CO
3	5/06/2008	added ST contribution	intermediate	CO

4	10/06/2008	added UNIVR and TUT	intermediate	CO
5	13/06/2008	Added SOTON contribution	Final	CO

# Table of Contents

---

<b>1.</b>	<b>Introduction</b> .....	<b>1</b>
<b>2.</b>	<b>Static Verification (task 3.1)</b> .....	<b>2</b>
2.1	Advances on improve -HDL developments .....	2
2.1	SAT and Model Checking.....	3
<b>3.</b>	<b>Dynamic Verification (task 3.2)</b> .....	<b>5</b>
3.1	Property coverage checker .....	5
3.1.1	Evaluation of property completeness .....	5
3.1.2	Removal of redundant properties.....	7
3.1.3	Detection of vacuous properties .....	7
3.2	Assertion checking using HLDDs .....	9
3.2.1	PSL to HLDD interface .....	10
3.2.2	HLDD-based assertion checking using PSL.....	11
<b>4.</b>	<b>Mixed Verification (task 3.3)</b> .....	<b>13</b>
4.1	Tools integration in Assertain.....	13
4.2	Assertain Workflow .....	13
4.3	Enhancing Equivalence Check TL – RTL.....	20
<b>5.</b>	<b>References</b> .....	<b>22</b>

# 1. Introduction

---

This document provides the final status of WP3 activities aimed at developing both dynamic and static techniques for the verification of the correct system-level modelling and synthesis of IP-cores.

This document contains the updated improve-HDL – Chapter 2 – and the consortium tool integration into Assertain – Chapter 3 from the D3.1b deliverable.

Latest developments on Property Coverage Checker (PCC) and High Level Decision Diagrams (HLDD) are reported in Chapter 3.

## 2. Static Verification (task 3.1)

### 2.1 Advances on improve-HDL developments

The major evolutions on imPROVE-HDL have been achieved and reported in D2.3, D3.1b and D4.2. During the last period we were nevertheless able to enhance further the performance of the tool. The following table includes the newest results compared to those reported previously in D2.3.

Design Name	Number of assertions	Total time at iso Depth (in s - max depth 100 cycles)			Average Depth at iso Time (in cycles - time out 1 hour)		
		Symbad Final	Vertigo D2.3	Vertigo D3.2	Symbad Final	Vertigo D2.3	Vertigo D3.2
TEST-0011	17	729	640	675	27	33	35
TEST-0012	11	7906	2753	2145	63	66	69
TEST-0013	12	304	181	176	max	max	max
TEST-0014	10	1115	422	296	max	max	max
TEST-0015	11	14151	2892	3169	33	100	100
TEST-0016	15	3803	987	935	max	max	max
TEST-0010	17	4713	4744	5337	45	54	64
TEST-0018	5	crash	14509	-	crash	45	-
TEST-0001	11	3503	3068	2716	60	88	96
TEST-0002	8	102	131	159	max	max	max
TEST-0003	8	17128	15462	16398	16	16	16
TEST-0005	7	56	63	86	max	max	max
TEST-0006	9	449	325	332	max	max	max
TEST-0007	13	1478	926	969	max	max	max
TEST-0008	10	2162	432	451	42	72	88
TEST-0009	25	8797	4665	4040	50	50	80
TEST-0149	8	1160	420	471	max	max	max
TEST-0150	16	136	169	179	max	max	max
TEST-0151	7	727	561	528	67	82	86
TEST-0152	7	4485	1124	743	62	73	74
TEST-0153	7	3892	2833	2935	max	max	max
TEST-0154	7	1181	476	275	80	130	130
TEST-0155	15	175	206	182	max	max	max

TEST-0156	16	1500	656	521	68	90	100
<b>Average depth</b>	<b>(272 ass.)</b>	-	-	-	<b>56 cycles</b>	<b>75 cycles</b>	<b>78 cycles</b>
<b>Total time</b>	<b>(272 ass.)</b>	<b>79652 s</b>	<b>44136 s</b>	<b>43518s</b>	-	-	-

TEST-0018 is not included in the TOTAL and Average figures.

The memory consumption was reduced further by 20% compared to the best results reported in D2.3 while keeping a similar average time and a better average depth. This enables imPROVE-HDL to be used on big modules (+800K gates) or on serial blocks (depth of several hundreds of cycles) under 4Gbytes.

## 2.1 SAT and Model Checking

---

SOTON's contributions can be organized into two main parts, the development effort of the model checker used in improveHDL and the research on model checking, SAT and related topics.

The development effort focused on streamlined data structures, incrementality and improved interpolant computation. This allowed improving the performance of the model checker, as mentioned above and in D3.1.

The research work focused on the following main areas:

1. Robust SAT solvers. This line of research aims to develop SAT solvers capable of tackling large complex problem instances.
2. Maximum Satisfiability. This line of research aims to allow solving multiple model checking problems simultaneously. Besides model checking, maximum satisfiability finds a number of other applications, including design debugging.
3. Portfolios of Algorithms. This line of research aims to allow dynamic selection of the best algorithm for a given problem instance.
4. Model Checking Formal Specifications. This line of research aims model checking high level specifications of designs.

The outcome of these lines of research work are summarized in the following publications:

1. De Oliveira Cantante De Matos, P. and Marques-Silva, J. (2008) Model Checking Event-B by Encoding into Alloy. In: ABZ2008, September 2008, London, UK. (In Press).
2. De Oliveira Cantante De Matos, P., Planes, J., Letombe, F. and Marques-Silva, J. (2008) An Algorithm Porfolio for MAX-SAT. In: European Conference on Artificial Intelligence (July 21st to 25th), Patras, Greece, July 21st-July 25th 2008, Patras, Greece. (In Press)
3. Marques-Silva, J. (2008) Model Checking with Boolean Satisfiability. Journal of Algorithms in Cognition, Informatics and Logic, 63 (1-3). pp. 3-16. (In Press).

4. Marques-Silva, J. (2008) Practical Applications of Boolean Satisfiability. In: Workshop on Discrete Event Systems (WODES'08), May 2008, Goteborg, Sweden.
5. Heras, F., Manquinho, V. and Marques-Silva, J. (2008) On Applying Unit Propagation-Based Lower Bounds in Pseudo-Boolean Optimization. In: International FLAIRS Conference, May 2008, Florida, USA.
6. Marques-Silva, J. and Manquinho, V. (2008) Towards More Effective Unsatisfiability-Based Maximum Satisfiability Algorithms. In: International Conference on Theory and Applications of Satisfiability Testing, May 2008, Guangzhou, China.
7. Letombe, F. and Marques-Silva, J. (2008) Improvements to Hybrid Incremental SAT Algorithms. In: International Conference on Theory and Applications of Satisfiability Testing, May 2008, Guangzhou, China.
8. Marques-Silva, J. and Planes, J. (2008) Algorithms for Maximum Satisfiability using Unsatisfiable Cores. In: Design, Automation and Test in Europe Conference and Exhibition, 6-10 March, Munich, Germany.
9. Lynce, I. and Marques-Silva, J. (2007) Breaking Symmetries in SAT Matrix Models. In: International Conference on Theory and Applications of Satisfiability Testing, May 2007, Lisbon, Portugal.
10. Marques-Silva, J. (2007) Interpolant Learning and Reuse in SAT-Based Model Checking. *Electronic Notes in Theoretical Computer Science*, 174 (3). pp. 31-43. ISSN 1571-0661.
11. Lynce, I. and Marques-Silva, J. (2007) Random Backtracking in Backtrack Search Algorithms for Satisfiability. *Discrete Applied Mathematics*, 155 (12). pp. 1604-1612.
12. Bombieri, N., Fummi, F., Pravadelli, G. and Marques-Silva, J. (2007) Towards Equivalence Checking Between TLM and RTL Models. In: Fifth ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), May 2007, Nice, France.
13. Marques-Silva, J. and Lynce, I. (2007) Towards Robust CNF Encodings of Cardinality Constraints. In: International Conference on Principles and Practice of Constraint Programming, September 2007, Providence, RI, USA.

In addition, for the line of work on Maximum Satisfiability, two solvers (msu1.2 and msu4.0) were submitted to the 2008 MaxSAT evaluation (<http://www.maxsat.udl.cat/08/>). In the industrial category, for plain MaxSAT, msu1.2 ranked as the best solvers. For the partial MaxSAT, msu1.2 ranked as the second best solver.

## 3. Dynamic Verification (task 3.2)

---

### 3.1 Property coverage checker

---

Model checking and, more generally speaking, assertion-based verification (ABV) require the definition of formal properties, according to a given logic (e.g., CTL, LTL, etc.), to establish whether a system satisfies its specification. Either case, model checking as well as ABV, the effectiveness of the verification depends on the quality of the defined properties.

The properties are usually written starting from informal specification of the design under verification (DUV). Therefore, their formalization requires a direct interaction of the verification engineers with both architects and designers, and it is based on the developed expertise of the verification group. However, this does not assure to define a consistent, complete and minimal set of properties.

*Consistency* is implicitly verified by the model checking process. A set of properties which hold on the DUV model are surely consistent, since they are satisfiable. On the contrary, not satisfiable properties require either a refinement of the DUV model or a modification of the properties themselves.

*Completeness* is desirable since a consistent, but incomplete, set of properties can miss to reveal possible design errors. A first methodology for evaluating property completeness has been defined during the SYMBAD project and implemented into the property coverage checker (PCC) tool.

In VERTIGO, UNIVR enhanced the PCC by refining the property completeness methodology [1] and it added new features for addressing the aspect of *minimality*.

In particular, two new strategies have been defined and implemented into the PCC to:

- remove redundant properties, i.e., properties that are logical consequences of other properties already checked [2];
- detect vacuous properties, i.e., properties that are trivially satisfied (e.g., an implication whose antecedent is always false) [3].

These new strategies allows the PCC tool to provide verification engineers with a powerful set of techniques to increase the quality of properties and reduce the time required for their verification.

#### 3.1.1 Evaluation of property completeness

Formal properties are defined to describe the expected behavior of the design, thus they represent the golden model for the implementation. However, this golden model could be incomplete, with respect to the specification of the design, since some requirements could be only partially formalized into properties. In this case, too few properties have been defined and the implementation could be wrong, even if it fulfills all the defined properties. Some papers [4]-[12] in the literature have addressed the problem of property completeness, but the proposed solutions present several limitations that prevent their applicability in real industrial cases. In particular, the main problem of existent approaches is due to the adoption of symbolic algorithms that suffer of the state explosion problem. In VERTIGO, we developed a different approach based on fault modelling and dynamic verification as shown in the upper side of Figure 1.

The degree of completeness of the golden model is measured by defining a *property coverage* based on the capability of the properties to identify faults that perturb the design implementation. We can conclude that the golden model is complete, with respect to the selected fault model, if the model checker refutes at least one property in presence of each fault affecting the outputs of the implementation. Otherwise, the golden model is incomplete. In fact, if a fault does not cause at least a failure among the defined properties, it means that the golden model is satisfied by two different implementations: the fault-free and the faulty one.

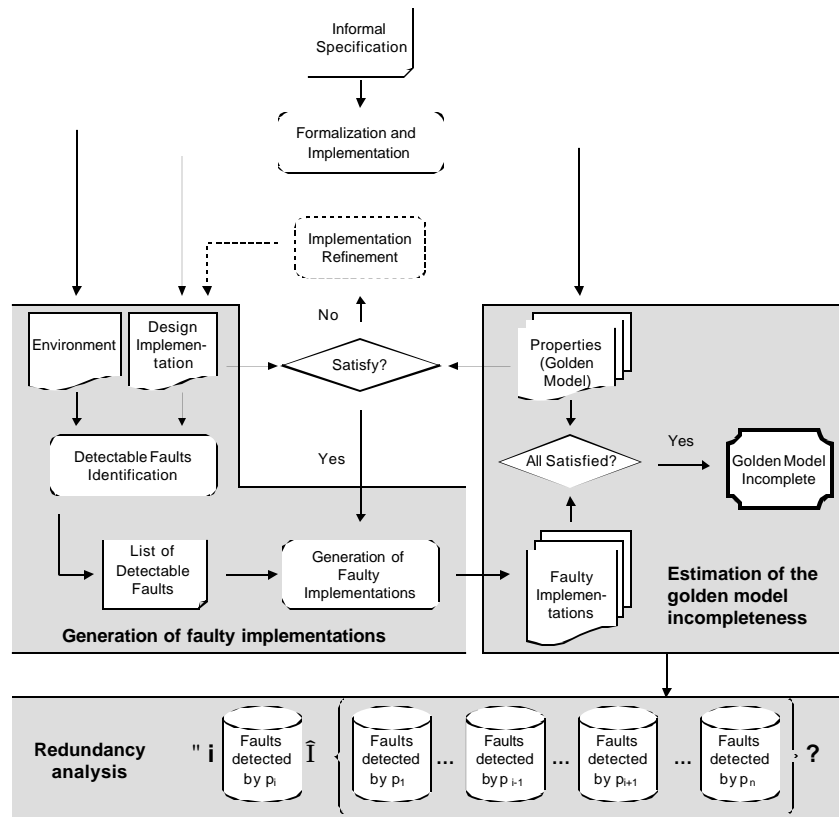


Figure 1: Property completeness evaluation and removal of redundant properties

The computation of property coverage basically consists of two phases: the automatic generation of faulty implementations, and the estimation of the golden model completeness.

- Generation of faulty implementations.** A functional fault model, strictly related to design errors, is selected to create a list of faults. Perturbations of the design implementation are generated by automatically injecting these faults in its functional representation. The fault list must include only detectable faults to achieve an accurate estimation of the golden model completeness. In fact, undetectable faults cannot cause failures on the properties, since they do not produce perturbations that affect the outputs of the implementation.
- Estimation of the golden model completeness.** While the properties representing the golden model are satisfied by the fault-free implementation, at least one property should be refuted if checked on a faulty implementation. On the contrary, the golden model would be incomplete. The computation of property coverage requires too long time/space resources if it is based on a pure model checking process. For this reason, we proposed a different metric, called *witness coverage*, which can be computed by exploiting dynamic verification, i.e., fault simulation and ATPG, instead of static verification, i.e., model checking. In particular, we showed that, theoretically,

witness coverage equals property coverage, and the adoption of dynamic verification allows us to greatly reduce the computation time of the property completeness analysis. However, dynamic verification is not exhaustive as static verification is, and relying only on witness coverage can lead to overestimate the property incompleteness. Thus, static verification cannot be completely removed to accurately evaluate the property coverage. For this reason, we proposed, an incremental methodology that evaluates the property coverage by joining dynamic and static verification. This allows to drastically reduce the evaluation time without incurring in the possibility of overestimating the property incompleteness.

Theoretical details and experimental results are available in [1]

### 3.1.2 Removal of redundant properties

Property minimization is strictly related to the concept of logical consequence among properties. Let  $M$  be a model of a DUV, and  $P = \{p_1, \dots, p_n\}$  a set of temporal properties defined to prove the correctness of  $M$  by model checking. We can say that  $P$  is surely not minimal if there exists  $p_i \in P$  such that  $p_i$  is a logical consequence of  $P \setminus \{p_i\}$ . In this case,  $p_i$  can be removed from  $P$ , without affecting the quality of model checking process, and we say that  $p_i$  is redundant. In fact, each behavior of  $M$  verified by  $p_i$  can be verified by  $P \setminus \{p_i\}$  too. According to the previous observation, a set of properties can be minimized by exploiting theorem proving. However, the complexity of theorem proving is exponential in the worst case. Moreover, theorem proving is not completely automatized, since human interaction is very often required to guide the theorem prover during the proof. On the contrary, in VERTIGO, we developed an alternative approach to decide about logical consequence of properties. It relies on functional fault simulation, and it restricts the use of theorem proving to a limited number of cases. In this way, a set of properties is reduced by adopting a faster, but theoretically founded, approach.

The main idea consists of identifying the “role” of each property of  $P$  in checking the correctness of the model  $M$  of the DUV. Thus,  $p_i$  can be removed from the set  $P$ , without affecting the model checking process, if its “role” is provided also by  $P \setminus \{p_i\}$ . Intuitively, we define such a “role” as the capability of a property  $p_i$  to check the correctness of a subset of the DUV functionalities, i.e., its property coverage. Thus, the same property coverage-based methodology used to measure the degree of completeness of the DUV model, constitutes the basis to evaluate also the degree of redundancy of the same golden model without using theorem proving. The property coverage computation provides, for each property  $p_i \in P$ , the set of faults  $F_i \subseteq F$  which are detected by  $p_i$ . Thus, it provides a metric to measure the importance of a property in the model checking process. The property redundancy analysis is performed by comparing the set of faults  $F_i$  detected by each property  $p_i \in P$ , as shown in the lower side of Figure 1

The proposed methodology does not guarantee to obtain a minimal set of properties. However, it allows to reduce the number of properties by removing the redundant ones. Theoretical details and experimental results are available in [2].

### 3.1.3 Detection of vacuous properties

Vacuum cleaning is a mandatory process when an implementation is verified with respect to a specification modelled by means of formal properties. In fact, vacuum cleaning looks for properties that, passing vacuously may lead verification engineers to a false sense of safety. Current approaches to vacuum cleaning, generally, exploit formal methods to search for an interesting witness proving that a property does not pass vacuously [13]-[16]. However, such approaches are, generally, as complex as model checking, and they require to define and

model check further properties, thus sensibly increasing the verification time. On the contrary, in VERTIGO, we developed an alternative approach based on fault simulation of checkers derived from the defined properties. A checker is automatically generated for each property, and a small set of interesting faults is injected into the checker to search for interesting witnesses proving that the corresponding property does not pass vacuously. If such witnesses are not identified, the approach highlights sub-formulae of the property that can lead to vacuous passes, thus guiding the verification engineers either in the refinement of the property, or the DUV, or the environment, or the testbenches (when properties are adopted for dynamic ABV).

Our methodology consider the notion of vacuity proposed by Beer et al. in [16], which states that a property  $p$  passes vacuously in a model  $M$  if  $M \models p$  and  $p$  includes a sub-formula  $q$  that does not affects  $p$  in  $M$ . In this case, it is said that  $p$  is  $q$ -vacuous in  $M$ .

According to such a definition, the majority of techniques proposed in the literature check the vacuity of a property  $p$  by using formal methods, that, generally, require to model check new properties obtained from  $p$  by substituting its sub-properties in some way.

On the contrary, in VERTIGO, we developed an alternative strategy to reason about vacuum cleaning, that, starting from the Beer's definition, uses fault simulation of checkers instead of model checking to speed-up the vacuity analysis.

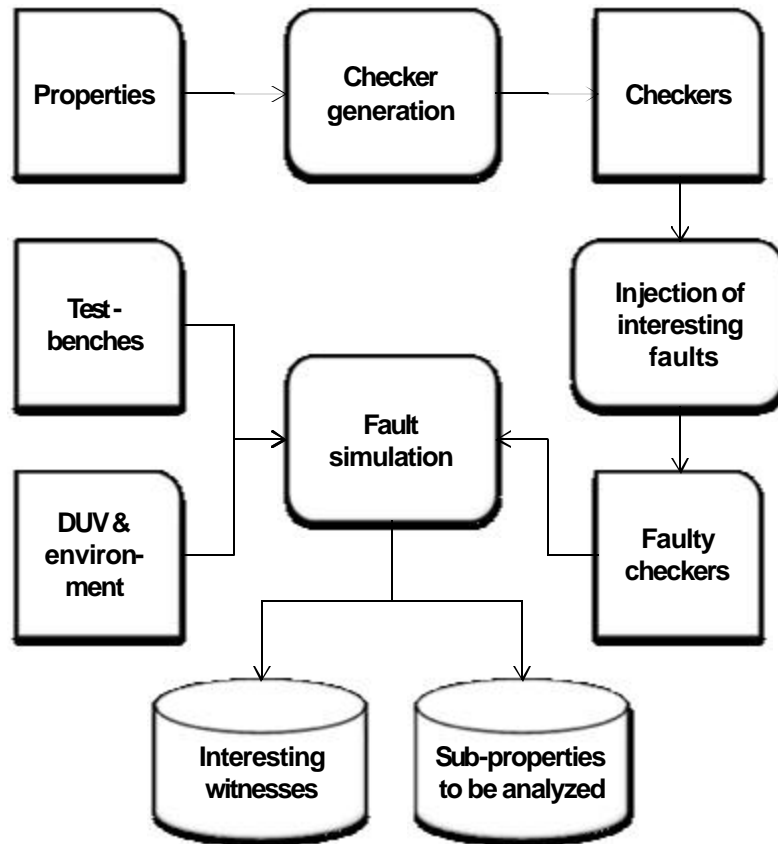


Figure 2: Detection of vacuous properties.

The proposed fault simulation-based methodology works as follows (Figure 2):

1. Given a set of properties that hold in the model of the DUV, a checker is automatically generated for each property  $p$ .

2. A set of interesting faults is injected in the so obtained checkers. In particular, for each property  $p$ , we inject, in the corresponding checker, one fault for each minimal sub-property of  $p$ . Intuitively, an interesting fault, perturbs the checker behaviour similarly to what happen when a sub-property  $q$  is substituted by *true* or *false* in  $p$ .
3. The faulty checkers are connected to the model composed of the DUV and the related environment. Then, testbenches are used to simulate the DUV. Note that, when dynamic ABV methodologies are adopted, testbenches are already available. Alternatively, when formulae are used in static model checking, testbenches for the corresponding checkers can be generated by using an automatic test pattern generator.
4. The vacuity analysis relies on the observation of the fault simulation result. A checker failure due to the effect of an interesting fault  $f$  corresponds to prove that the sub-property  $q$  associated to (perturbed by) fault  $f$  affects the truth value of  $p$ . Consequently, the sequence of values generated by the testbench that causes the checker failure (i.e., the test sequence of  $f$ ) is an interesting witness proving that  $p$  is not  $q$ -vacuous. On the contrary, faults that do not cause checker failures (i.e., not detected faults) must be thoroughly analyzed, since the incapability of detecting a fault is symptom of vacuity. In this case, our methodology provides the verification engineer with feedback to analyze such a symptom by relating each undetected fault to a single sub-property of  $p$ .

Experimental results have shown the high efficiency of the proposed methodology and its consistency with the vacuum cleaning technique proposed by Beer. Theoretical details and experimental results are available in [3].

## 3.2 Assertion checking using HLDDs

---

This subsection presents new tools developed for High-Level Decision Diagram (HLDD) based verification for IPs. In previous deliverables, the following HLDD verification tools have been presented:

- HLDD-based code coverage analysis tool (D3.1b, [17])
- HLDD-based assertion checker using VHDL checkers from FoCs (D3.1b, [18])
- Test generation engine combining HLDD/EFSM formalism (D3.1b)
- HIF to HLDD interface (D2.6)

The two last-mentioned activities were carried out in close co-operation with UNIVR.

In this deliverable we introduce two new tools, which have been implemented in order to better conform with the requirements of the project by adding support to PSL assertions. These tools include:

- PSL simple subset to HLDD interface
- HLDD-based assertion checker for PSL [19]

### 3.2.1 PSL to HLDD interface

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) (See deliverable D2.6) 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, 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 (defined in D2.6).

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. 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 1.

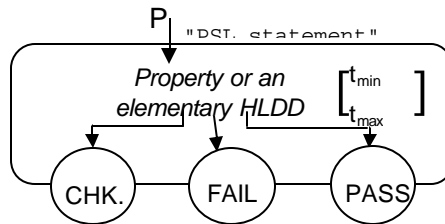


Figure 1. Standard PPG interface

Example PPGs created for 4 PSL operators are shown in Figure 2. Note, that the logic implication operator ‘ $\rightarrow$ ’ in Fig. 2b exits to the terminal node ‘CHECKING’ when the precondition  $P_a$  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. 2d. 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 PPG 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.

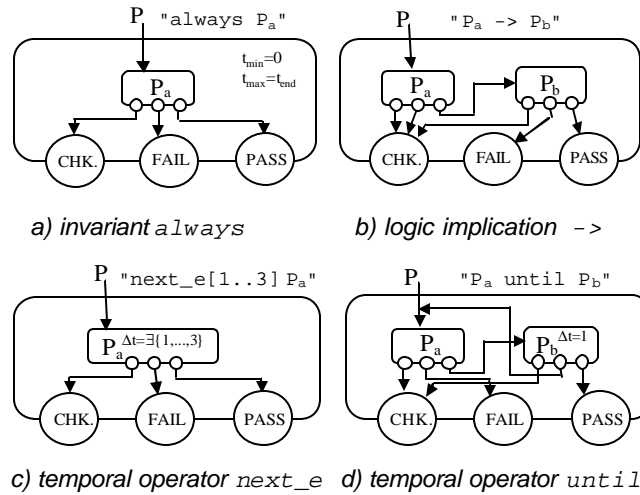


Figure 2. PPGs for a set of PSL operators

### 3.2.2 HLDD-based assertion checking using PSL

The HLDD-based assertion checker envisioned in deliverable D2.6 has been implemented. Table 1 shows the experimental results of assertion checking execution times comparison between THLDD simulator and a commercial tool QuestaSim from Mentor Graphics. The experimental benchmarks are GCD benchmark 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 1. The third and fourth columns show the simulation and assertion checking execution times required for the THLDD simulator, respectively. The fifth (highlighted) and the sixth columns are the total execution time taken by the proposed approach and QuestaSim correspondingly. The values in the sixth column include approximately 0.5 sec of QuestaSim simulation initialization time that was impossible to exclude from the measurement. Both tools showed 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 1. Assertion checking experiments

Design	Stimuli Length (clcks)	The proposed approach			QuestaSim
		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

## 4. Mixed Verification (task 3.3)

### 4.1 Tools integration in Assertain

TEDA's beta product 'Assertain' links the dynamic verification skills of TEDA with the Static and formal tools of AERIE/SOTON.

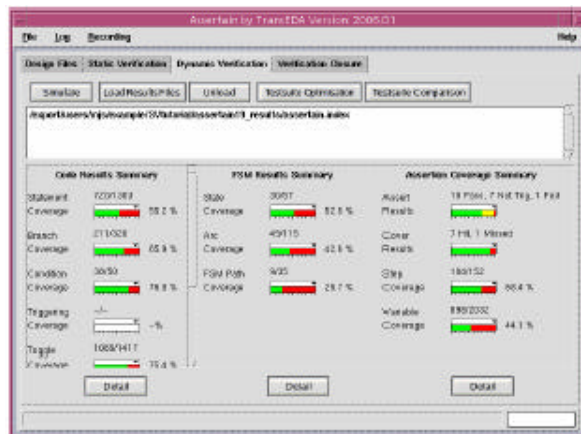
Users of the tool follow through the dynamic verification of the TEDA tool in the traditional way. Users then use the formal tool to start again on the entire problem rather than focus on specific tightly bounded problems.

The key metrics are risk reduction, time to market reduction, fast project implementation, repeat results, and stability of supply.

Other key metrics are that the tools are easy to configure and can be supported without recourse to the vendor.

### 4.2 Assertain Workflow

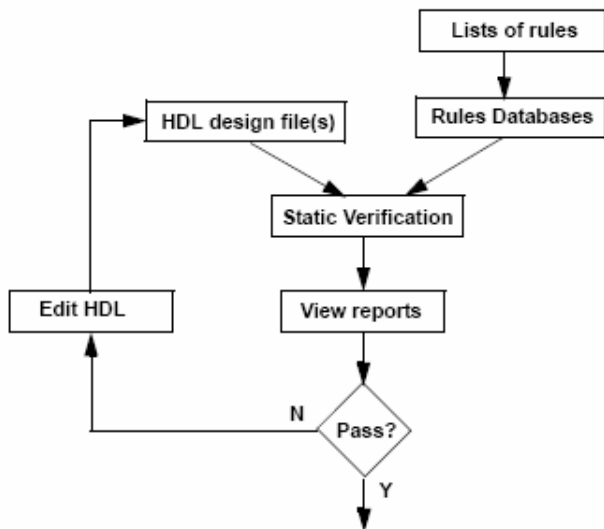
#### Dynamic Verification



Users use the tool working through the tabs from left to right : Setup, Dynamic verification , Static verification and design sign-off.

## Inputs for Static Verification

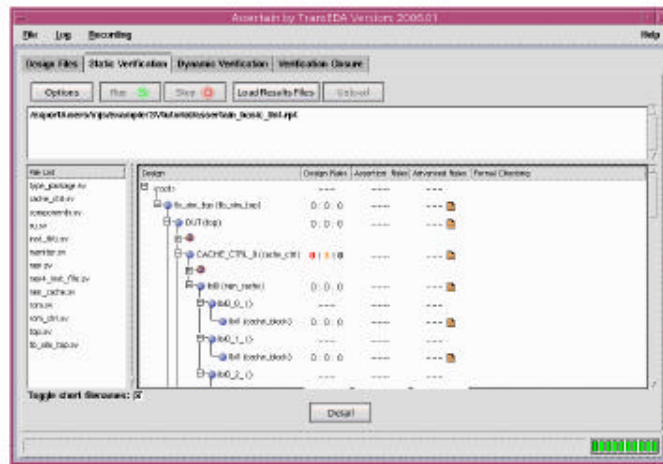
The inputs to Static Verification are one or more HDL design files and one or more Rules Databases. A number of Rules Databases are supplied as part of the Assertain installation and are comprehensive collections of individual rules that focus on specific coding requirements. You can use the Rules Databases supplied by TransEDA “as-is”; you can edit the supplied Rules Databases, or you can create your own Rules Databases by building them by selecting from the hundreds of supplied Rules.



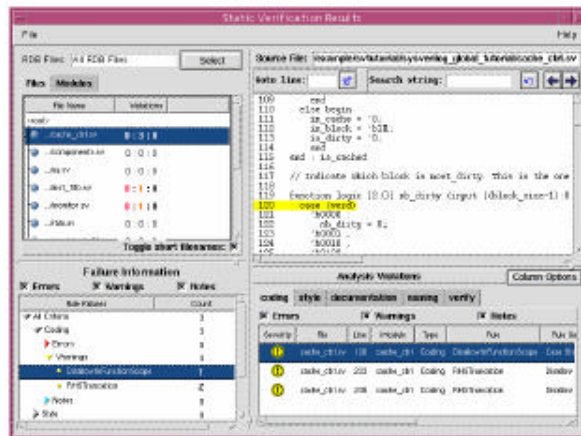
## Outputs from Static Verification

The results of the Static Verification checks can be examined using the Assertain GUI or provided as reports that list any rule violations and identify the lines of HDL source that caused the violations.

## Running Static Verification



The design files that you loaded and compiled will be listed in the pane at the left-hand side of the window with the results from the basic linting operation shown in the Design



### Verification closure

Results are brought together in a GUI, but reports can be generated as ASCII text or as HTML for examination using a browser.



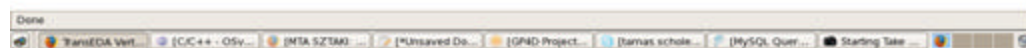
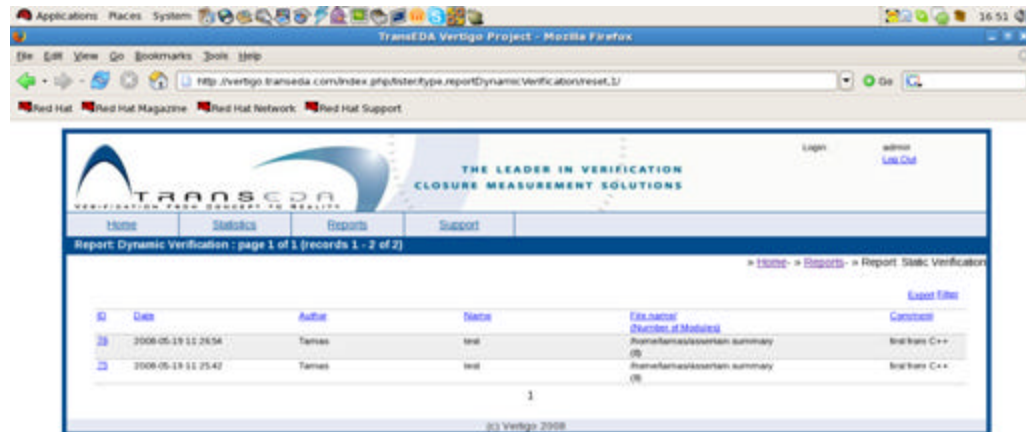
The Results Summary window shows the percentage of instances that have achieved their target coverage for the selected coverage criteria.

It should be noted that this is a different measure from the individual coverage figures because it is related to the number of instances achieving the required coverage – rather than a measure of the coverage achieved for the entire design.

## Workflow integration

Currently Assertain integration is tied by a series of switches and parameter files which allows each partner to work independently of each other.

TransEDA have built an Apache,MySQL,PHP application on Linux RHE3 which allows all of the tests from the Assertain workflow to be stored and queried to correlate progress through the workflow.

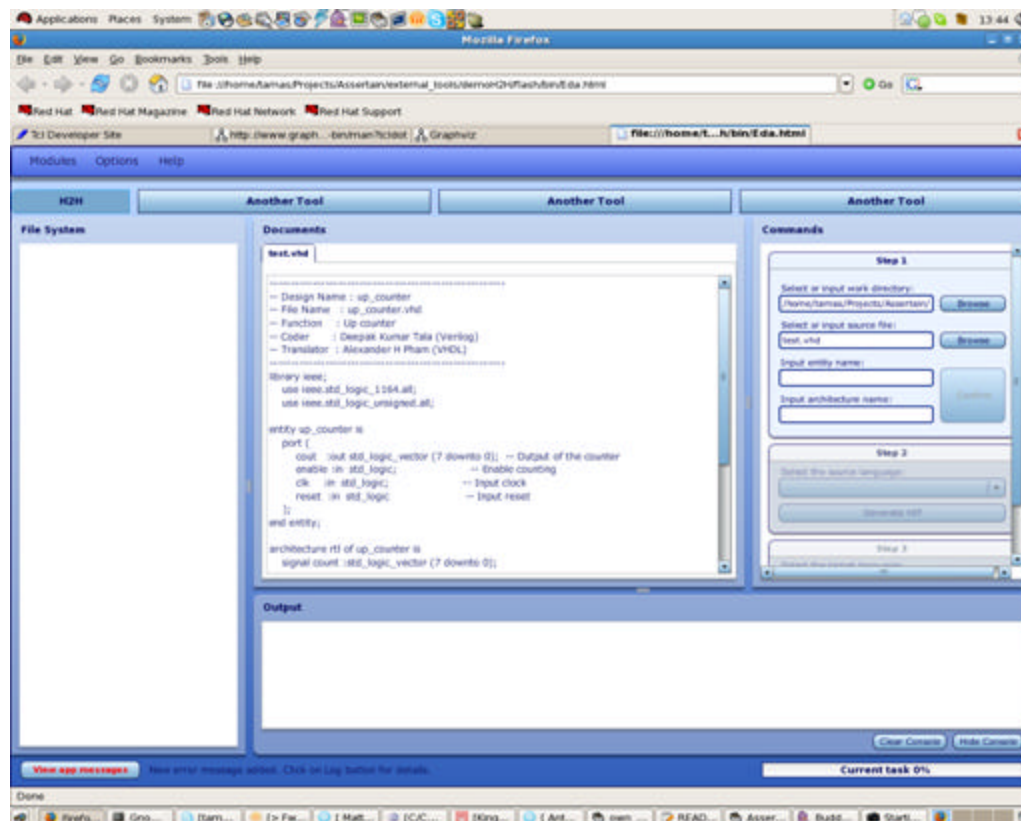


The results are rendered in HTML allowing output as text or Adobe Flash:

The screenshot shows the TransEDA Veritigo Project web interface. The browser window title is 'TransEDA Veritigo Project - Mozilla Firefox'. The URL is 'http://vertigo.transeda.com/index.php?liberType=reportDynamicModules&act=filter?report\_id=70'. The page header features the TransEDA logo and the tagline 'THE LEADER IN VERIFICATION CLOSURE MEASUREMENT SOLUTIONS'. Below the header is a navigation menu with 'Home', 'Statistics', 'Reports', and 'Support'. The main content area displays a report for 'Modules: page 1 of 1 (records 1 - 8 of 8) (filtered)'. The report includes a table with columns for 'ID', 'Name/NameID', 'Statement Branch', 'Condition', 'Transitions', 'Tasks', 'Tasks.C', 'Path.C', 'Excluded', 'MIL.C', 'AL.C', 'EM\_PATH', 'As. Res.', 'Cover', 'As. Res.', 'As. VM.C', and 'As. VM.C'. The table shows various modules and their associated statistics. Below the table, there is an 'Assertion Results' section with a summary: 'Instance: Passed: 30 Not triggered: 0 Failed: 5 Filtered: 0 With subcomponents: Passed: 30 Not triggered: 0 Failed: 5 Filtered: 0'. The table below this summary shows individual assertion results for various modules.

ID	Name/NameID	Statement Branch	Condition	Transitions	Tasks	Tasks.C	Path.C	Excluded	MIL.C	AL.C	EM_PATH	As. Res.	Cover	As. Res.	As. VM.C	As. VM.C
42	root	67.51%	62.90%	%	%	%	%	%	%	%	%	66.67%	%	%	%	%
Instance only		Pass	Failed	Filtered	Pass	Failed	Filtered	Pass	Failed	Filtered	Pass	Failed	Filtered	Pass	Failed	Filtered
With subcomponents		Pass	Failed	Filtered	Pass	Failed	Filtered	Pass	Failed	Filtered	Pass	Failed	Filtered	Pass	Failed	Filtered
42	root	100.00%	%	%	%	%	%	%	%	%	%	%	%	%	%	%
43	app.exe(1)	80.00%	62.90%	%	%	%	%	%	%	%	%	%	%	%	%	%
44	app.exe(2)	80.00%	62.90%	%	%	%	%	%	%	%	%	%	%	%	%	%
45	app.exe(3)	80.00%	62.90%	%	%	%	%	%	%	%	%	%	%	%	%	%
46	app.exe(4)	80.00%	62.90%	%	%	%	%	%	%	%	%	%	%	%	%	%
47	app.exe(5)	80.00%	62.90%	%	%	%	%	%	%	%	%	%	%	%	%	%
48	app.exe(6)	80.00%	62.90%	%	%	%	%	%	%	%	%	%	%	%	%	%

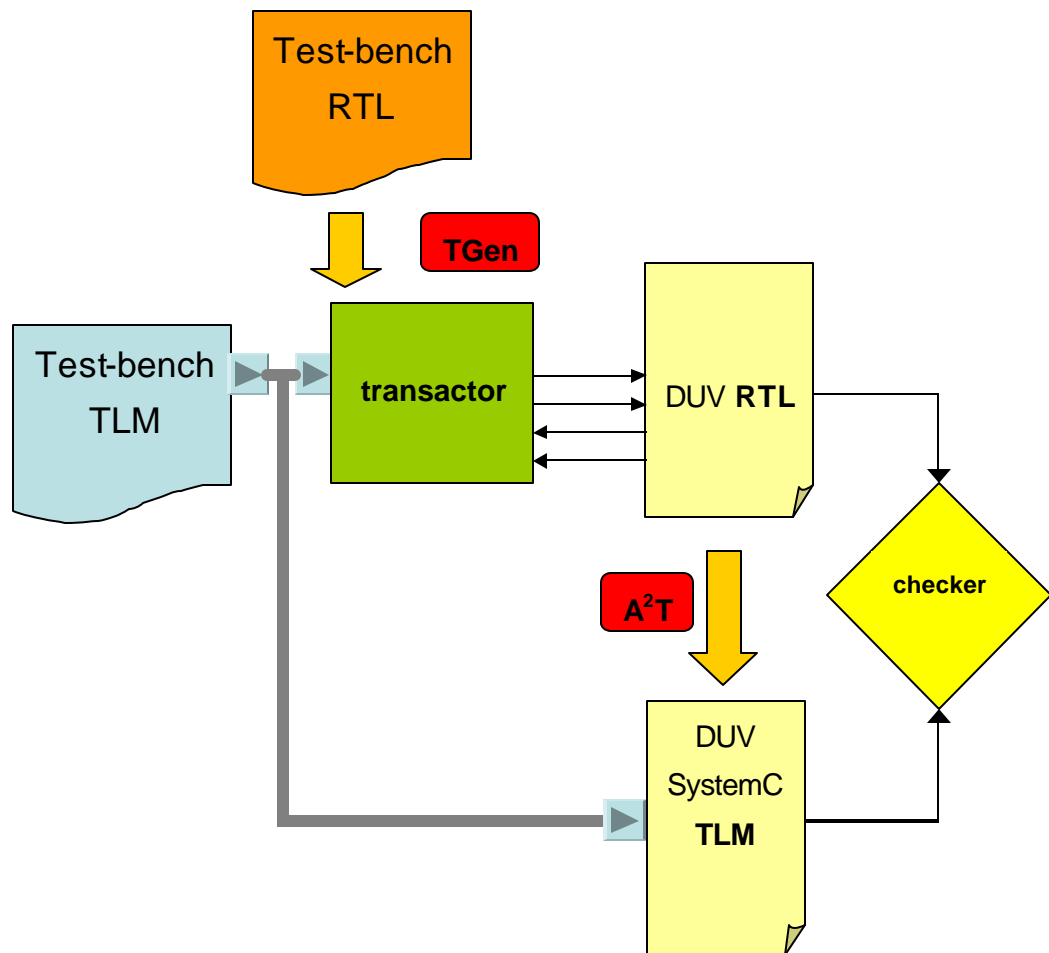
Tools from the consortium are accessible from an Adobe Flash interface:



The current release of Flash provides a GUI, Glue logic at the presentation layer and the ability to call external DLL's and executables and render the output in a consistent user interface.

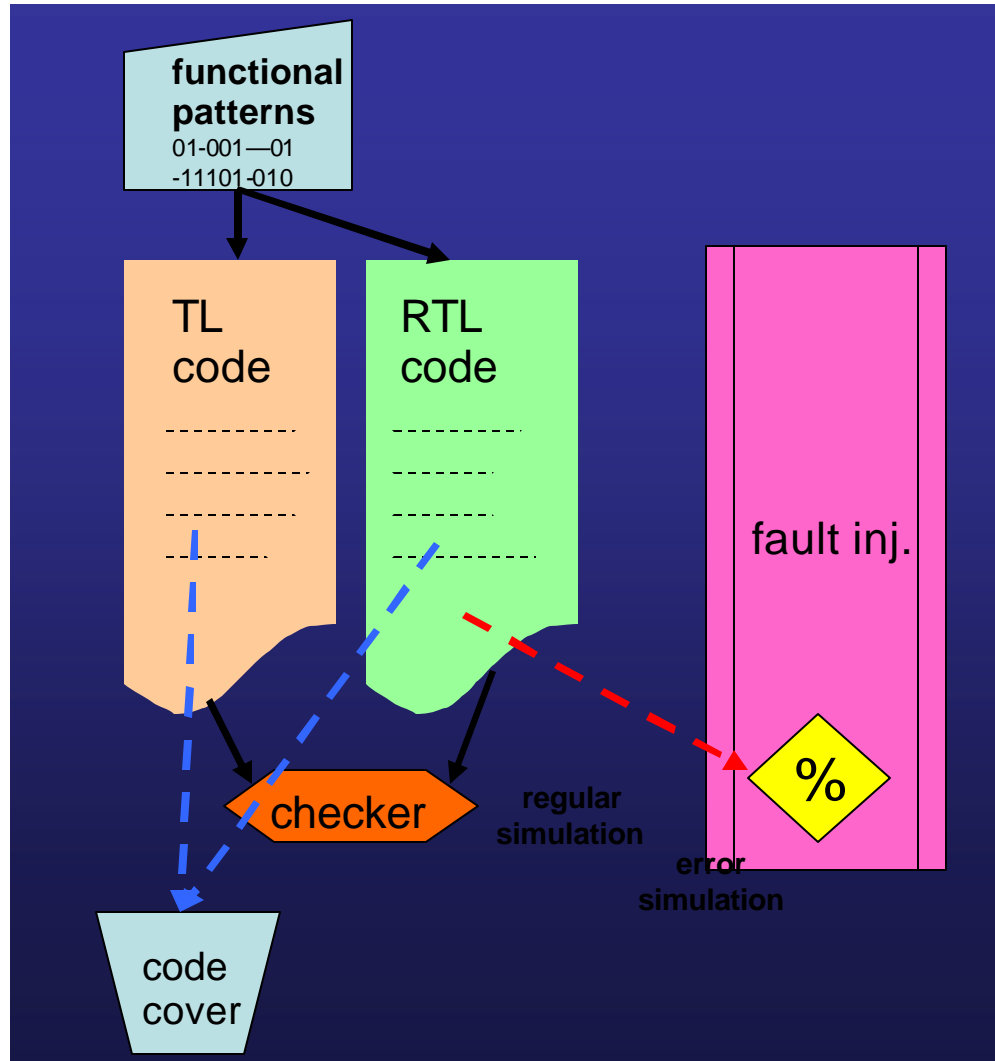
## 4.3 Enhancing Equivalence Check TL – RTL

Proving Equivalence between TLM and RTL models is an open problem, that still requires a precise definition; such definition should consider the different semantics of the used languages, e.g. SystemC on one side and VHDL, VERILOG, SYSTEM VERILOG on the other side, and the different abstraction levels that can be implemented at RTL – therefore the equivalence cannot be checked at the cycle precision but requires the use of a transaction. The overall framework for TLM and RTL simulation is depicted in the picture.



The testbench is written at TLM and drives both the RTL and the TLM description. The blocks in red colour report two tools: TGen provides the transactor TLM / RTL for slave protocol, A<sup>2</sup>T provide some automation in the TLM abstract model generation. For the verification setting we may as well assume that the transactor is generated by hand and the SystemC TLM is provided by a separate flow. Also in the checker a recognition capability must be implemented in order to synchronize RTL and TLM response, generally based on an event mechanism.

Some criteria to grade the level of quality of this equivalence checking process must be put in place. In addition to the classical code coverage criterion the error coverage criterion may be employed, as depicted in the figure.



Classical code coverage can be extracted both on the TL code and the RTL code. In the VERTIGO frame we can apply also error simulation – to the RTL model only. The high coverage patterns produced by means of error simulation may be used as input to the dynamic equivalence checking process, performed by the regular checker. Although not formally proven, this process increases our confidence in the equivalence robustness.

Alternatively, we may consider the output equivalence expressed by the checker as a combinational property that must hold throughout the whole simulation. If the error injected RTL does not differ from the TL code, that reveals a gap in the coverage of the patterns that should certify the equivalence of the two descriptions. Such use of the checker during the error simulation is very easy to setup and shorten the process turnaround time.

## 5. References

---

- [1] Fedeli A., Fummi F., Pravadelli G. (2007), Properties Incompleteness Evaluation by Functional Verification. In *EEE Transactions on Computers*, vol. 56 , n. 4, pp. 528-544.
- [2] Fummi F., Pravadelli G. (2007), Too Few or too Many Properties? Measure it by ATPG! In *Springer Journal of Electronic Testing*, vol. 23 , n. 5 , pp. 373-388.
- [3] Di Guglielmo L., Fummi F., Pravadelli G. (2008), Vacuity Analysis by Fault Simulation. In *Proc. of ACM/IEEE MEMOCODE 2008*.
- [4] Chockler H, Kupferman O, Kurshan RP, Vardi MY (2001) A practical approach to coverage in model checking. In: *Proc. of CAV*, pp 66–78.
- [5] Chockler H, Kupferman O, Vardi MY (2001) Coverage metrics for temporal logic model checking. In: *Proc. of international conference on tools and algorithms for the construction and analysis of systems*, vol 2031 of LNCS. Springer, New York, NY, pp 528–542.
- [6] Chockler H, Kupferman O, Vardi MY (2003) Coverage metrics for formal verification. In: *Correct hardware design and verification methods*, vol 2860 of LNCS. Springer, New York, NY, pp 111–125.
- [7] Hoskote Y, Kam T, Ho PH, Zao X (1999) Coverage estimation for symbolic model checking. In: *Proc. of ACM/IEEE DAC*, pp 300–305.
- [8] Jayakumar N, Purandare M, Somenzi F (2003) Dos and don'ts of CTL state coverage estimation. In: *Proc. of ACM/IEEE DAC*, pp 292–295.
- [9] Katz S, Grumberg O, Geist D (1999) Have I written enough properties? - A method of comparison between specification and implementation. In: *Correct hardware design and verification methods*, vol 1703 of LNCS. Springer, New York, NY, pp 280–297.
- [10] Lee T-C, Hsiung P-A (2004) Mutation coverage estimation for model checking. In: *Proc. of international symposium on automated technology for verification and analysis*, vol 3299 of LNCS. Springer, pp 534–368.
- [11] Xu , Kimura S, Horikawa K, Tsuchiya T (2005) Transition traversal coverage estimation for symbolic model checking. In: *Proc. of ACM/IEEE MEMOCODE*, pp 259–26
- [12] Xu X, Kimura S, Horikawa K, Tsuchiya T (2006) Transition-based coverage estimation for symbolic model checking. In: *Proc. of ACM/IEEE ASP-DAC*, pp 1–6.
- [13] Beer I., Ben-David S., Eisner U., and Rodeh Y (1997) Efficient detection of vacuity in ACTL formulas. In *Proc. of CAV*, vol. 1254 of LNCS, pp. 279–290.
- [14] Kupferman O. and Vardi M.Y. (1999) Vacuity Detection in Temporal Model Checking. In *Proc. of Conference on Correct Hardware Design and Verification Methods*, pp. 82–96.
- [15] Kupferman O. and Vardi M.Y. (2003) Vacuity Detection in Temporal Model Checking. *International Journal on Software Tools for Technology Transfer*, vol. 4(2):pp. 224–233, 2003.
- [16] Beer I., Ben-David S., Eisner U., and Rodeh Y (2001) Efficient Detection of Vacuity in Temporal Model Checking. *Formal Methods in System Design*, vol. 18(2):pp. 141–163, 2001.
- [17] Jaan Raik, Uljana Reinsalu, Raimund Ubar, Maksim Jenihhin, Peeter Ellervee. Fast Code Coverage Analysis using High-Level Decision Diagrams. *Proceedings of the 11th IEEE Workshop on Design and Diagnostics of Electronic Systems (DDECS)*, IEEE Computer Society, April, 2008.
- [18] 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
- [19] Maksim Jenihhin, Jaan Raik, Anton Chepurov, Raimund Ubar. Temporally Extended High-Level Decision Diagrams for PSL Assertions Simulation. *Proceedings of the 13th IEEE European Test Symposium*, IEEE Computer Society, Los Alamitos, USA, May, 2008.