



# STREP – IST 033709 VERTIGO

Verification & Validation of Embedded System Design  
Workbench

## Deliverable D4.2 – Tools for Static and Dynamic Verification of IP's Interaction

DUE DATE 31 May 2008  
ACTUAL DATE 18 June 2008  
START OF PROJECT 01 June 2006 DURATION 30 months  
ABSTRACT This deliverable presents tools and technologies concerning static and dynamic verification of IP's interaction and HW/SW Co-simulation.  
AUTHOR, COMPANY Graziano Pravadelli (UNIVR)  
WORKPACKAGE/TASK WP4/T4.1, T4.2, T4.3, T4.4  
FILING CODE VERTIGO/Deliverables/08\_D4.2\_VERTIGO\_R6  
KEYWORDS verification, interaction, IP, co-simulation, abstraction

### DOCUMENT HISTORY

Release	Date	Reason of change	Status	Distribution
1	12/05/08	UNIVR prepares the first draft	Draft	CO
2	30/05/08	Contribution from partners	Draft	CO
3	05/06/08	UNIVR prepares the second draft	Draft	CO
4	11/06/08	Update to partner contribution	Draft	CO
5	13/06/08	Final integration	Draft	CO

6	18/06/08	Final version	Final	CO
---	----------	---------------	-------	----

# Table of Contents

---

<b>1.</b>	<b>Introduction.....</b>	<b>1</b>
<b>2.</b>	<b>Static Verification (T4.1).....</b>	<b>2</b>
2.1	Verification Environment based on Petri Nets.....	2
2.1.1	Iterative verification with stubs.....	2
2.1.2	Transactor-based verification.....	4
2.2	Assertions languages in imPROVE-HDL.....	5
2.2.1	PSL support.....	5
2.2.2	SVA support.....	5
2.3	Bounded model checking in imPROVE-HDL.....	9
<b>3.</b>	<b>Dynamic Verification (T4.2).....</b>	<b>10</b>
3.1	HLDD manipulation for verification coverage.....	10
3.1.1	HLDD-based code coverage analysis.....	10
3.1.2	HLDD manipulation techniques.....	11
3.2	Assertain Workflow.....	13
3.2.1	Dynamic Verification.....	13
3.2.2	Verification closure.....	14
3.2.3	Workflow integration.....	14
3.3	Tool Integration in Assertain.....	15
3.3.1	Integration of imPROVE-HDL in Assertain.....	15
3.3.2	Integration of Laerte++/Decider in Assertain.....	15
<b>4.</b>	<b>HW/SW Co-simulation (T4.3).....</b>	<b>20</b>
4.1	The HSN Framework.....	20
4.1.1	General Framework.....	20
4.1.2	Multi ISS.....	21
4.2	Automatic Generation of Device Drivers.....	23
4.2.1	Multilevel Device Driver.....	24
4.2.2	Generation of 1° level device driver.....	25
4.2.3	Generation of 2° level device driver.....	25
<b>5.</b>	<b>Mixed Static/Dynamic Verification (T4.4).....</b>	<b>28</b>
5.1	Mixed Verification of System Integration Unit Lite (SIUL) module.....	28
5.1.1	Automatic property generation.....	30
5.2	Formal method-aided simulation.....	31
<b>6.</b>	<b>References.....</b>	<b>33</b>



# 1. Introduction

---

This deliverable presents the status of methodologies and tools related to static, dynamic and mixed static/dynamic verification of IP's interaction.

Section 2 is related to static verification. The emphasis is put on Petri nets-based verification and support for both assertions languages PSL and SVA on the imPROVE-HDL property checker.

Section 3 is related to dynamic verification. The emphasis is put on advancements about HLDD-base verification strategies, and on the integration of the dynamic verification tools developed in VERTIGO into the Assertain workflow.

Section 4 is devoted to the HW/SW co-simulation framework. Enhancements implemented inside HSN for running multiple instances of instruction set simulators are presented, and a set of rules for automatically generating device drivers are proposed.

Finally, Section 5 refers to the mixed static/dynamic verification issue. In particular ST will describe a verification flow for Alternate Function applied at SoC level.

## 2. Static Verification (T4.1)

This section is devoted to static verification, and in particular it presents progress related to Petri Nets-based verification and imPROVE-HDL.

### 2.1 Verification Environment based on Petri Nets

In this section, we present the verification environment, which has been developed in this project, based on the hierarchical Petri net notation, the PRES+. The environment consists of two main verification techniques, one performing iterative verification of IP's interaction based on stabs, while the other performing transactor-based verification.

In the developed verification environment, we consider systems that are built using pre-designed components (IP blocks). Figure 1 illustrates such a system. Each component, in the figures throughout this paper, is depicted as a box with circles on its edge. The circles represent the ports of the component, which it uses for communication with other components.

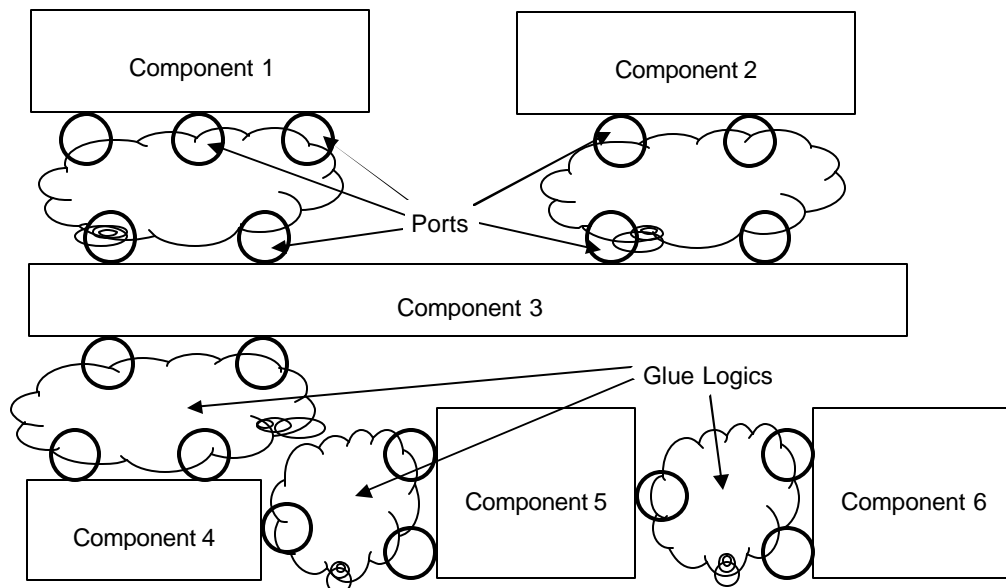


Figure 1: Targeted system topology

The glue logic inserted between communicating components is depicted in Figure 1 as clouds. For example, the interfaces of two or more components connecting to each other may use different incompatible communication protocols. Thus, they cannot communicate directly with each other. For that reason, it is necessary to insert an adaptation mechanism between the components in order to bridge this gap. This adaptation mechanism would then be the glue logic.

#### 2.1.1 Iterative verification with stabs

The glue logic inserted between two components is to be verified so that it satisfies the requirements imposed by the connected components. Figure 2 illustrates the basic procedure. Model checking is used as the underlying verification technique. The model of the glue logic together with models corresponding to the interface behaviour of the interconnected

components (called stubs) are given to the model checker together with the (T)CTL formulas describing the properties to be verified.

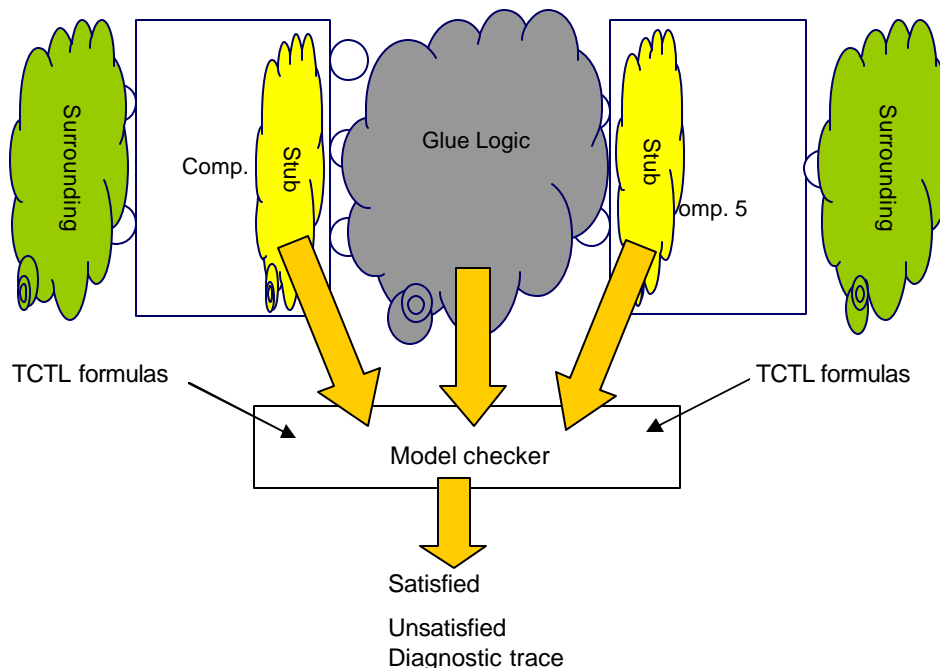


Figure 2: Overview of the proposed methodology

A stub is a model which behaves exactly in the same way as the component with respect to the interface consisting of ports connected to the glue logic under verification.

As a result of the verification, the model checker replies whether the properties are satisfied in the model or not. If they are not, the model checker provides a diagnostic trace in order to tell the designer what caused the properties to be unsatisfied.

As shown in Figure 2, the part of the system not included in the verification of the particular glue logic is called the *surrounding* of the glue logic.

In the discussion throughout this section, as well as in the toolset we have implemented, the glue logic, the stubs and the components are assumed to be modelled in a design representation called Petri-net based Representation for Embedded Systems (PRES+). It is a Petri-net based representation with several extensions. For more elaborate description of the modelling mechanism, we refer to deliverable D2.4.

Note that a stub is a model which behaves exactly in the same way as a certain component with respect to a particular interface. Furthermore, since a component generally has several interfaces, it also has several stubs, one for each interface.

An interface is defined as a set of ports. Hence, interfaces can be partially ordered with respect to the subset relation. As a consequence, stubs can also be partially ordered by the same relation, due to the fact that they are defined with respect to a particular interface. In the bottom of the hierarchy, so called empty stubs can be found. Such stubs either produce or consume (depending on whether the stub will be attached to an out-port or an in-port) events (tokens) containing any message (value) at any point in time, i.e. they behave completely randomly. On the other extreme, in the top of the hierarchy the top-level stub is found. It behaves exactly in the same way as the full component, since it is defined with respect to all ports of the component.

When verifying a particular glue logic using stubs given by the component provider, the designer has to select appropriate stubs from this hierarchy. Experiments have shown that using low-level stubs generally leads to considerably shorter verification times. However, using low-level stubs makes it more likely that the property is unsatisfied than if high-level stubs were used.

Based on these facts, an iterative verification technique has been developed [5]. Low-level stubs should initially be used in the verification. If the property is unsatisfied, a stub situated at a higher level in the hierarchy is used instead in the next verification round. The diagnostic trace obtained from the previous verification indicates which stubs violated the property and thus should be changed. This procedure iterates until either the property is verified to true, or the diagnostic trace indicated that the fault is situated in the glue logic (in which case a design error was found). It should be noted that the designer is always guided by the diagnostic trace.

A second scenario is that no stubs are given by the component providers, but they must be generated by the designer. An algorithm which automatically generates stubs, given the model of a component and an interface, has also been developed.

## 2.1.2 Transactor-based verification

This section presents a transactor-based verification technique for the verification at mixed abstract levels of systems consists of several communicating components, as indicated in Figure 1. During the development phase of such systems, it is often desirable to check if certain temporal logic properties are satisfied in the system under development. Such analysis can be obtained by feeding a model of the system into a model checking tool together with properties to be verified, as described in the previous section. At the same time, the components are iteratively refined and more and more details are added to the system. This naturally leads to a situation where some parts of the system are more refined than others. However, it is still desirable to occasionally verify the system to ensure that the recently performed refinement steps did not violate any, possibly critical, properties.

When refining the components, the interfaces of those components are simultaneously refined. However, the interfaces are shared or connected with other components, which are not yet refined. This creates an incompatibility of interfaces between the involved components and channels. In order to overcome this problem, the channel is replaced by a transactor between the incompatible interfaces. A transactor can thus be seen as a channel connecting components at different levels of abstraction, or a semi-refined channel. The transactor shall encapsulate the same external behaviour as the channel it replaces with respect to delays, noise etc.

The transactor takes high-level requests and translates them into low-level ones, and vice versa. It is described in Timed Sequential Extended Regular Expressions (TSERE), which is both intuitive and sufficiently expressive for this purpose. The TSEREs (and thereby also the transactors) are given either by the designer himself, or, in a standardised context, by a third-party provider.

Usually, a transactor can be said to be a mix of the two versions of the channel. It, however, also contains additional protocol information not explicit in the channels, e.g. how to split the high-level message and the time separation between the address and data transmission. Therefore, the information captured in the channels is not sufficient for formulating the TSEREs. In addition, the transactor respects the external timing behaviour of the channels.

In the developed verification methodology, an abstraction of the model is first obtained with respect to the components and channels referred to by the properties. The abstracted model is then input to the UPPAAL model checker, by first translating our Petri-net model into Timed Automata, the input language of UPPAAL. If the result of the model checking was false, the

model might need to be refined (relative to the abstraction done in the verification methodology, not the design itself) based on diagnostic information obtained from the model checker. In case the refinement of the abstraction fails, the properties are concluded not to be satisfied. If, on the other hand, the model checking result was true, it can be concluded that the properties hold in the model.

To generate a transactor is a two-step process. First, the behaviour of the transactor must be described with TSEREs. This must be done in such a way that each high-level request is mapped onto low-level ones, while preserving the external behaviour, e.g. timing. Once a TSERE for the transactor is developed, that TSERE is automatically translated into an equivalent PRES+ model. The resulting PRES+ model, together with the PRES+ model capturing the behaviour of the whole system, can then be analysed by our Petri net based formal verification technique described before [6].

## 2.2 Assertions languages in imPROVE-HDL

A major effort for AERIE under WP4 was to provide strong support for both assertions languages PSL and SVA on our property checker imPROVE-HDL. We provide below the final status on the work achieved under the Vertigo project. Please refer to the D4.1a document for the general explanations and references.

### 2.2.1 PSL support

Our PSL support is built around the FoCs PSL compiler from IBM. As stated in D4.1b this support is now fully tested and verified. Several improvements were made since the D4.1b report and the following features are now supported:

- property checking with operator @
- PSL in mixed mode (i.e., with a design including both Verilog and VHDL)
- hierarchical path for VHDL
- no limitation on the "abort" keyword

### 2.2.2 SVA support

As stated in document D4.1b we decided to develop our own SVA synthesis rather than doing a translation from SVA to PSL and relying on FoCs.

This translation is now finished and has been integrated in imPROVE-HDL. Validation is on-going. Following is the table of SVA features that are supported today:

Features	reference	Supported	Not supported
Bit		X	
logic		X	
reg		X	
integer			X
time		X	
Shortint			X
Longint			X
int			X
byte			X

Table 1: SVA support part 1

Features	reference	Supported	Not supported	
<b>sequence declared in</b>	Module	lrm p. 206	X	
	Interface			X
	Program			X
	clocking block			X
	Package			X
	compilation-unit scope			X
<b>seq1 ##n seq2</b> (n = int > 0)	lrm p.205	X		
<b>seq1 ##[n:m] seq2</b>		X		
<b>seq1 ##[n:\$] seq2</b>		X		
<b>seq1 ##n seq2 ##m `true</b>		X		
<b>&lt;bool&gt; ##0 &lt;bool&gt;</b>	lrm p. 205	X		
<b>&lt;seq&gt; ##0 &lt;seq&gt;</b>			X	
<b>(seq1 ##n seq2) ##0 (seq3 ##m seq4)</b>			X	
<b>sequence used in another sequence</b>	lrm p.208	X		
<b>seq1 [*n]</b>	lrm p.210	X		
<b>seq1 [*0:\$]</b>		X		
<b>seq1 [*n:m]</b>		X		
<b>seq1 [*0:m]</b>		X		
<b>seq1 [*n:\$]</b>		X		
<b>seq1 [*0]</b>		X		
<b>(seq1 ##n seq2) [*m]</b> (n, m > 0)		lrm p.211	X	
<b>(seq1 ##n seq2) [*m:k]</b> (n, m, k > 0)			X	
<b>seq1 [-&gt;n]</b>	lrm p.209, 211	X		
<b>seq1 [-&gt;n:m]</b>		X		
<b>seq1 [=n]</b>	lrm p.209, 212	X		
<b>seq1 [=n:m]</b>		X		
<b>seq1 and seq2</b>	lrm p.214	X		
<b>seq1 intersect seq2</b>	lrm p.217	X		
<b>seq1 or seq2</b>	lrm p.218	X		
<b>first_match(seq1)</b>	lrm p.221	X		
<b>first_match(seq1, x = e)</b>			X	
<b>bool1 throughout seq2</b>	lrm p.222	X		
<b>seq1 within seq2</b>	lrm p.223	X		
<b>endpoint sequence</b>			X	

Table 2: SVA support part 2

<b>data manipulation in sequence</b>	local reg signal variable	lrm p.226	X	
	local reg vector variable		X	
	design signals and vectors as formal parameters		X	
	(posedge clock) as formal parameter			X
	! as formal parameter			X
	any formal parameters meant to be used in [n:m]			X
	local variables of a sequence as formal parameter of another sequence			X
	local reg signal variable with 'seq1 and seq2'	lrm p.227	X	
	local reg vector variable with 'seq1 and seq2'		X	
	local reg signal variable with 'seq1 or seq2'		X	
	local reg vector variable with 'seq1 or seq2'		X	
	local reg signal variable with 'seq1 intersect seq2'			X
	local reg vector variable with 'seq1 intersect seq2'			X
<b>complex sequences</b>	seq1 ##n (seq2 and seq3)			X
	seq1 ##n (seq2 or seq3)			X

Table 3: SVA support part 3

features		reference	Supported	Not supported
<b>properties declared in</b>	module	lrm p.229	X	
	interface			X
	programs			X
	clocking block			X
	package			X
	compilation-unit scope			X
<b>not(prop1)</b>		lrm p.230	X	
<b>prop1 or prop2</b>		lrm p.231		X
<b>prop1 and prop2</b>				X
<b>if (bool1) prop1</b>				X
<b>if (bool1) prop1 else prop2</b>				X
<b>seq1  -&gt; prop2</b>				X
<b>seq1  =&gt; prop2</b>				X
<b>property used in another property</b>		lrm p.231	X	
<b>data manipulation in property</b>	local reg signal	lrm p.231	X	
	local reg vector			X
	design signals and vectors as formal parameters	lrm p.231	X	
	§ as formal parameter			X
	any formal parameters meant to be used in [n:m]			X
<b>disable iff(bool1) prop1</b>		lrm p.231	X	
<b>nested implications</b>		lrm p.235	X	
<b>recursive properties</b>		lrm p.236		X

Table 4: SVA support part 4

features		reference	Supported	Not supported
<b>assert, assume, cover</b>	no formal parameters	lrm p.247-249	X	
	with design signals and vectors as formal parameters		X	
<b>always assert property(prop1) always cover property(seq1)</b>		lrm p.250		X
<b>assertions embedded in procedural code</b>		lrm p.250		X
<b>module parameters used in the properties</b>		svah p.101	X	
<b>generate if/for ... assert/cover &lt;prop&gt; ... end</b>		svah p.101		X
<b>multiclocks</b>	@(posedge clk1) for prop/seq1 @(posedge clk2) for prop/seq2	lrm p.240-244	X	
	property or sequence with @(posedge clk1) and @(posedge clk2)			X
	matched	lrm p.246		X
<b>bind</b>	binding inside the module itself	lrm p.258	X	
	binding a module m1 with an assertions module m2 from a third module m3		X	
	binding a module m1 with an assertions module m2 outside any modules			X
	binding an interface			X
	binding a compilation-unit scope			X

Table 5: SVA support part 5

features	reference	Supported	Not supported
<b>Direct instantiation of SVA module into VHDL test-bench</b>	svah p.150		X
<b>Binding of SVA Verification Module to VHDL Model</b>	svah p.151		X
<b>VHDL Model in a SystemVerilog Testbench with SVA Module</b>	svah p.152		X

Table 6: SVA support part 6

All SystemVerilog synthesizable data-types and constructs are supported.

Note that synthesis of assertions implies that values are sampled after blocking assignment whereas in simulation the assertions should be evaluated with the values of the signals sampled before any assignments have been applied. This can exceptionally imply some differences in the assertion result between formal verification and simulation. See the SystemVerilog 3.1a LRM for details.

## 2.3 Bounded model checking in imPROVE-HDL

imPROVE-HDL is build on top of a bounded model checker (BMC), implemented by SOTON. The design and implementation of the BMC module has been done by SOTON, with support from AerieLogic. Recent work included optimizations to the data structures, namely the representation of unfoldings, and the use of incrementality. Another effort addressed the computation of interpolants. The performance improvements have been reported in D3.1 and D3.2.

## 3. Dynamic Verification (T4.2)

---

This section is devoted to describe progresses achieved on HLDD-based verification and to present the Assertain workflow. Moreover, the integration of imPROVE-HDL and Laerte++ into Assertain is described.

### 3.1 HLDD manipulation for verification coverage

---

This subsection presents new tools developed for High-Level Decision Diagram (HLDD) based verification of IPs' interaction. Previously, in deliverables D3.1b and D4.1b the following HLDD verification tools have been presented:

- HLDD-based code coverage analysis tool [3]
- Test generation engine combining HLDD/EFSM formalism
- HIF to HLDD interface

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

In this deliverable we focus on the new tool, which have been implemented in order to improve the quality of verification by introducing new coverage metrics. Namely, the HLDD manipulation tool for code coverage analysis [4] will be presented. TUT is currently working on integrating the coverage analysis module into the Assertain workflow.

#### 3.1.1 HLDD-based code coverage analysis

In order to analyze quality of verification of hardware designs translated to HLDD-s, 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. As an example, Figure 3 depicts HLDD representations of state and data register variables of 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.

Similar to the statement coverage, branch coverage has also very clear representation in HLDD simulation. The ratio of every edge activated in the simulation process constitutes to HLDD branch coverage. For example, the branch coverage item corresponding to `DATA_IN > RMAX = true` in the VHDL code of the b04 design maps to the edge denoted by a bold arrow in the HLDD in Figure 3. The statement `RMAX := DATA_IN` is represented by the terminal node surrounded by bold circle in the corresponding HLDD.

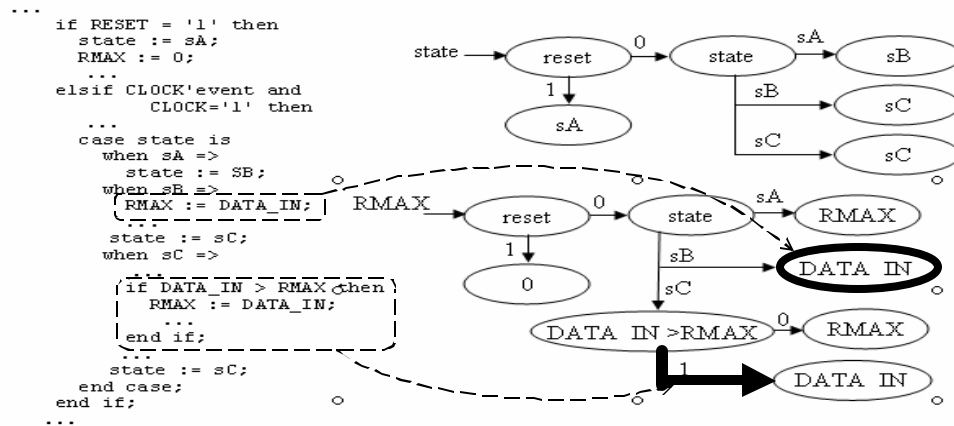


Figure 3: Mapping code coverage items to HLDD

### 3.1.2 HLDD manipulation techniques

The main contribution of this deliverable is the new HLDD manipulation technique allowing efficient code coverage analysis. In fact, if HLDD is generated for each output variable and the generation process is terminated at the primary input signals then code coverage analysis for the diagram will be equivalent to the path coverage metric. However, as it was mentioned above, enumerating all the paths through a design is infeasible and it is easy to see that the corresponding HLDD may be of exponential size.

Therefore, another approach is adopted that differs from the traditional one of generating a diagram for each primary output. When representing systems by decision diagram models, a network of HLDD-s is implemented where each internal HDL variable has its corresponding HLDD. During the simulation in HLDD systems, the values of some variables labelling the nodes of a HLDD are calculated by other HLDD-s of the system. Such partitioning helps avoiding the node explosion problem of DD-s and keeps the size requirements for resulting HLDD systems acceptable.

The method proposed for generating HLDDs suitable for code coverage analysis consists of the following steps:

1. Generate a HLDD tree for each system variable
2. Reduce nodes with identical succeeding subgraphs
3. Unite identical terminal nodes

The above steps are explained by an example presented in Figure 4, which depicts HLDD manipulations for the 'state' variable of the b04 design presented in Figure 3. As the first step, a HLDD tree for variable v is generated by traversing the full control flow graph of the design and collecting the values assigned to v at each control step. If the value of v does not change at current control step then terminal node with the present value of variable will be created. Figure 4a shows the HLDD generated for the variable state in b04.

Then, reduction rules are applied to eliminate nodes for which all successor nodes (in general case, succeeding subgraphs) are identical. As a result a reduced HLDD is obtained (Figure 4b). Finally, we create a minimized reduced HLDD by uniting identical terminal nodes (Figure 4b). HLDD generation experiments on a set of ITC99 benchmarks show that around 45-80% of nodes are removed by the reduction step from the initial HLDD tree. Further 40-60% of nodes will be eliminated by the minimization step.

We propose reduced HLDD-s as a suitable model for code coverage analysis because it provides for more stringent coverage metrics than minimized HLDD-s. At the same time it is a more compact representation than full HLDD trees. Furthermore, in terms of speed of simulation reduced HLDD offers equal performance when compared to the minimized model. This is because of the fact that by both models the number of edges to be traversed is exactly the same. However, in full trees the number of diagram edges would be considerably higher.

Comparative experiments between the HLDD-based code coverage analysis tool implemented in this paper and a popular HDL commercial simulation tool were carried out on circuits belonging to the ITC99 benchmark family. While there was no definite advantage of the speed of basic logic simulation of benchmarks to either of the tools, the overhead of coverage checking in the popular commercial tool is much higher than in the case of HLDD-s. When HLDD-s have penalty for coverage calculation time in a 1% to 4% range, the commercial simulator uses from 28% up to 78% extra time for coverage assessment .

Table 1 presents the characteristics of the different HLDD representations introduced. The columns *min*, *red.* and *tree* show the number of nodes/edges in minimized HLDD, reduced HLDD and HLDD tree models, respectively. From the Table, it can be seen that around 45-80 % of nodes are removed by the reduction step from the initial HLDD tree. Further 40-60 % of nodes will be eliminated by the minimization step.

Table 2 compares code coverage analysis comparing statement coverage and branch coverage assessment results on reduced HLDD-s (red.), on minimized HLDD-s (min) and on a well-known commercial tool using the same set of input stimuli for all three models. As it can be seen from the experiments, the reduced HLDD model always achieves the best (i.e. most stringent results) of all three. The minimized HLDD has the poorest outcome for statement coverage and traditional HDL simulator is the weakest for measuring branch coverage in most cases.

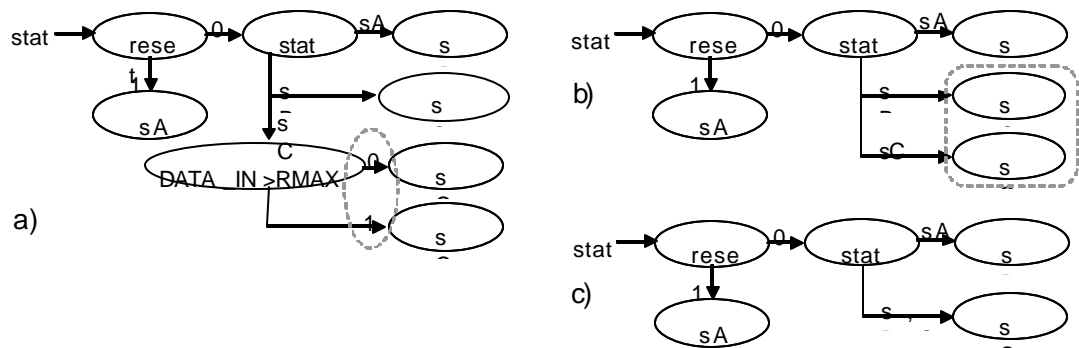


Figure 4: a) HLDD tree, b) reduced HLDD and c) minimized reduced HLDD

Design	# of nodes			# of edges	
	min	red.	tree	min	red.
b01	30	57	267	52	62
b02	16	26	48	24	24
b06	47	116	440	83	111
b09	44	69	125	62	64

Table 7: Characteristics of different HLDD manipulations

Design	Test len.	Statement coverage, %			Branch coverage, %		
		red.	min	HDL	red.	min	HDL
b01	14	<b>86.0</b>	100	93.8	<b>74.2</b>	84.6	88.9
	23	<b>96.5</b>	100	100	<b>90.3</b>	100	100
b02	10	<b>92.3</b>	100	96.3	<b>91.7</b>	<b>91.7</b>	93.8
	14	<b>100</b>	<b>100</b>	<b>100</b>	<b>100</b>	<b>100</b>	<b>100</b>
b06	11	<b>80.2</b>	100	85.5	<b>79.3</b>	89.2	87.5
	52	<b>98.3</b>	100	100	<b>98.2</b>	100	100
b09	23	<b>87.0</b>	100	100	<b>85.9</b>	87.1	100
	33	<b>100</b>	<b>100</b>	<b>100</b>	<b>100</b>	<b>100</b>	<b>100</b>

Table 8: Comparison of code coverage analysis results

## 3.2 Assertain Workflow

Assertain will analyse your design; extract the required Finite State Machine data; make instrumented copies of your files (your original files are unaltered) and perform the linting checks. Progress will be displayed in the *Log* window.

The *Dynamic Verification Results* window displays a design *Hierarchy View* in the upper part of the window, and a *Metrics View* in the lower part of the window.

The *Dynamic Verification Results - Details* window displays a *Code View* in the upper part of the window, and a *Detail View* in the lower part of the window.

Verification closure summarises each step and tests against a sign-off threshold to guide the engineer towards closure.

All tests are written to a database and then combined with partners tests to show verification closure across all tool-sets.

### 3.2.1 Dynamic Verification

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

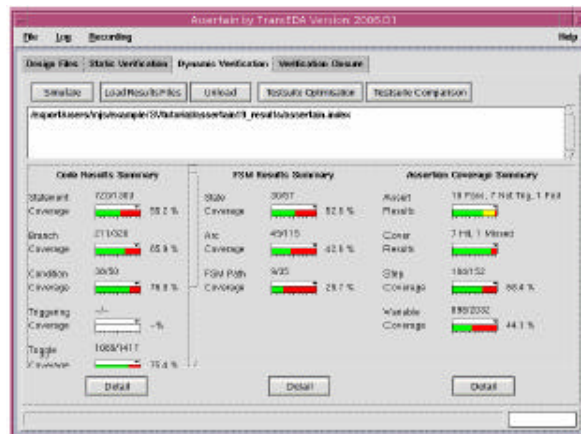


Figure 5: Dynamic verification results summary



### 3.3 Tool Integration in Assertain

The tools developed in T4.1 and T4.2 are brought together in the Assertain workflow as a loose integration of each partners executables tied by run-time switches. The output is rendered in HTML with each run held in a back-end database.

#### 3.3.1 Integration of imPROVE-HDL in Assertain

Assertain includes imPROVE-HDL as a static analysis tool for several purposes which are described in the documents related to Assertain. This integration needed numerous specific adaptations from imPROVE-HDL that have been achieved over the last months, especially for call routines and output results.

#### 3.3.2 Integration of Laerte++/Decider in Assertain

The ATPG verification framework derived from the integration of Laerte++ and Decider ATPG engines (and more generally, all tools involved in the HIFsuite) has been included in the Assertain workflow. Laerte++/Decider has been integrated into the Assertain GUI by decoupling the ATPG engine from its command interface and from the results manager. The engine interacts with a command interface implemented by means of TCL/TK scripts. On the contrary, the command interface interacts with a graphical flash application running inside the Assertain framework as shown in Figure 8 and Figure 9.

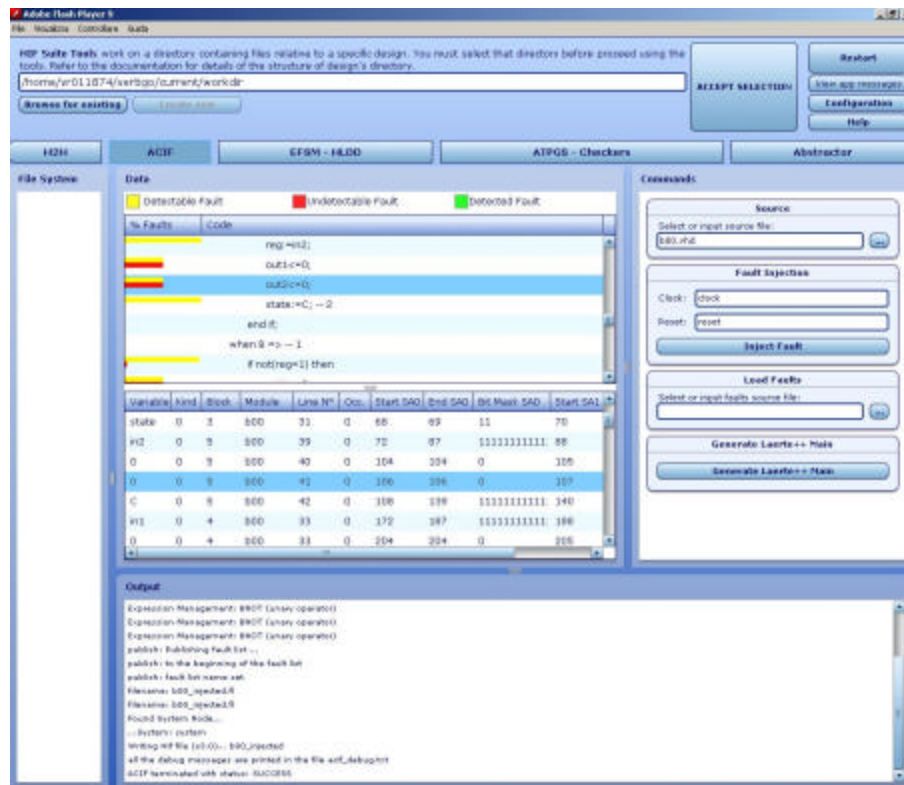


Figure 8: Laerte++/Decider (fault injection tab) inside Assertain

The results of Laerte++/Decider verification sessions are then saved in the Assertain database. The record memorized in such a database are described in Figure 10 and Figure 11. In this way, Assertain can take care of Laerte++/Decider's output and it can use the corresponding test sequences for running coverage measure analysis.

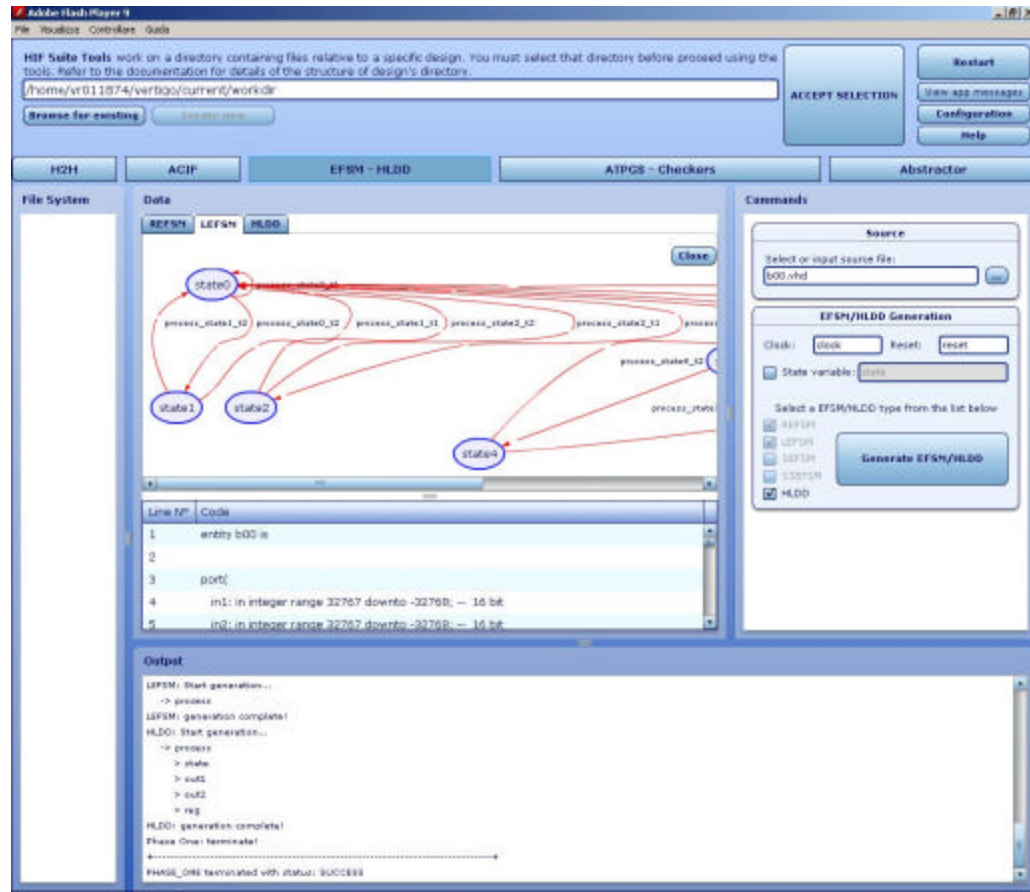


Figure 9: Laerte++/Decider (EFSM tab) inside Assertain

```

1. //////////////////////////////////////
2. //      TPG PARAMETER
3. //
4. // LAST MODIFIED      : May 29 2008
5. // VERSION            : 3.1.94
6. // EXPERIMENT DATE    : Thu May 29 08:49:50 2008
7. // COMMAND LINE       : ./laerte_am2910.x -m fault_list.fl -s 50 -l 50
8. // ONLY FROM FILE     : False
9. // INPUT TESTSET FILE : None
10. // VCR ACTIVE         : False
11. // EXIT ON CONS. VIOL. : False
12. // TIME LIMIT         : 100000
13. // MAXSEQLEN          : 50
14. // MAXSEQGEN          : 50
    
```

```

15. // STUCKERR          : 3608
16. // RND GEN. SEED    : 1212043587
17. //
18. //-----
19. //
20. //          SIM/ATPG RESULTS
21. //
22. // EXECUTION TIME      : user = 696667 usec - system 156009 usec
23. // GENERATED SEQUENCES : 50
24. // FAULT COVERAGE      : 71.8 %
25. // ATPG EFFECTIVENESS  : 86 %
26. // AVG. FLIP RATIO     : 0 %
27. // SKIPPED FAULT SIM.  : 0
28. // CONS. VIOLATION     : 0
29. // TESTSET LEN (#SEQ)  : 43
30. // TESTSET LEN (#VECT) : 1638
31. // TOTAL APPLIED ROUTINES : 0
32. // OUTPUT TESTSET FILE  : tset.out
33. //
34. //////////////////////////////////////
35. ## i[4,I]
36. ## ccen_bar[1,Bv]
37. ## cc_bar[1,Bv]
38. ## rld_bar[1,Bv]
39. ## ci[1,Bv]
40. ## oebar[1,Bv]
41. ## d[13,I]
42. 24
43. 111000011110001101010101
44. [omissis]
45. 101100110011100101010101
46. RESET
47. 110110110010001010111101
48. [omissis]
49. 001000011110101001011101
50. END
    
```

Figure 10: Laerte++/Decider output results for Assertain database

Line	Field name	Description
2 -16		<i>Test pattern generator parameters section</i>
4	LAST MODIFIED	The date on which the design has been linked to Laerte++
5	VERSION	The Laerte++ version in the form X.Y.Z

		<ul style="list-style-type: none"> <li>• X is the major release number</li> <li>• Y is the minor release number</li> <li>• Z is the subversion revision number</li> </ul>
6	EXPERIMENT DATE	The date on which the experiment has been done
7	COMMAND LINE	The command line used to execute the experiment
8	ONLY FROM FILE	The testsequences can be loaded from a file
9	INPUT TEST SET FILE	The file from which Laerte++ reads the testsequences
10	VCR ACTIVE	VCR mode
11	EXIT ON CONS. VIOL.	Exit condition in case of constraint violation
12	TIME LIMIT	The experiment terminates after a time threshold
13	MAXSEQLEN	The max sequence length
14	MAXSEQGEN	The max number of generated sequences
15	STUCKERR	The number of stuck-at
16	RND GEN. SEED	The random seed used to testset generation
20-32		<i>Test pattern generator results section</i>
22	EXECUTION TIME	The experiment execution time
23	GENERATED SEQUENCES	The number of generated sequences
24	FAULT COVERAGE	The achieved fault coverage
25	ATPG EFFECTIVENESS	The effectiveness of the generated testset
26	AVG. FLIP RATIO	Ratio between 0 and 1 in the test set
27	SKIPPED FAULT SIM.	Number of skipped faults
28	CONS. VIOLATION	Constraint violation
29	TEST SET LEN (#SEQ)	The number of effective testsequences
30	TEST SET LEN (#VECT)	The number of testvectors in the testset
31	TOTAL APPLIED ROUTINES	The number of assembly routines used in case of routine-mode-execution for programmable devices
32	OUTPUT TESTSET FILE	The file which contains the information of Figure 1.
35-41		<p><i>Primary inputs section</i></p> <p>The syntax is</p> <pre>## _primary_input_name[_size_,_type_]</pre> <ul style="list-style-type: none"> <li>• <code>_primary_input_name_</code> is the name of the primary input</li> <li>• <code>_size_</code> is the size of the primary input (number of bits)</li> <li>• <code>_type_</code> is the type of the primary input                         <ul style="list-style-type: none"> <li>○ I is integer type</li> <li>○ U is unsigned integer type</li> <li>○ Bv is a bit vector or bit</li> </ul> </li> </ul>

		<p>(<i>_size_</i> = 1) type</p> <ul style="list-style-type: none"> <li>o Lv is a logic vector or logic (<i>_size_</i> = 1) type</li> </ul>
42		The testvector size. It is the sum of all primary input sizes plus clock signal and reset signal sizes (generally)
43-50		<p><i>Testset section</i></p> <p>This section contains the Laerte++ ATPG testset result.</p> <p>Each testset contains zero or more testsequences and each testsequence contains one more testvectors.</p> <p>Each testvector is a sequence of 0 and 1 .</p> <p>The last two bit of the testvector is the reset and clock bit, generally the reset is set 0 and the clock is set 1 .</p> <p>The primary input order is provided by <i>Primary inputs section</i>.</p> <p>For example at line 43 there is 101100110011100101010101 this means to assign</p> <ul style="list-style-type: none"> <li>• i &lt;= 1011</li> <li>• ccen_bar &lt;= 0</li> <li>• cc_bar &lt;= 0</li> <li>• rld_bar &lt;= 1</li> <li>• ci &lt;= 1</li> <li>• oebar &lt;= 0</li> <li>• d &lt;= 0111001010101</li> <li>• reset &lt;= 0</li> <li>• clock &lt;= 1</li> </ul>
46	RESET	The RESET instruction is used to separate each testsequence and is equivalent to the testvector 0000000000000000000011
50	END	The END instruction is used to identify the end of the testset

Figure 11: Meaning of Laerte++/Decider output recorded in the Assertain database

## 4. HW/SW Co-simulation (T4.3)

This section reports advancements with respect to Deliverable D4.2 regarding techniques and tools for dynamic verification of HW/SW interaction. First, enhancements developed in the HSN HW/SW co-simulation framework are presented [1]. Then, a set of rules are proposed for automatic generation of device drivers for HW modules starting from the unpartitioned system-level description [2].

### 4.1 The HSN Framework

In this section, we describe the extension implemented in HSN to allow a co-simulation between SystemC and multiple instances of ISS. This allows us to integrate the Laerte++ ATPG and the Property Coverage Checker (PCC) into HW/SW co-simulation. In fact, both Laerte++ and the PCC requires two instances of the SystemC model representing the design under verification: one fault-free and one with injected faults. Thus, two instances of SW are required too: one for driving the fault free SystemC design and the other for driving the faulty SystemC.

#### 4.1.1 General Framework

The proposed co-simulation scheme targets a generic architectural template in which an application SW (that will eventually run without changes on an actual board), running on top of an OS, accesses one or more hardware devices that have to be designed.

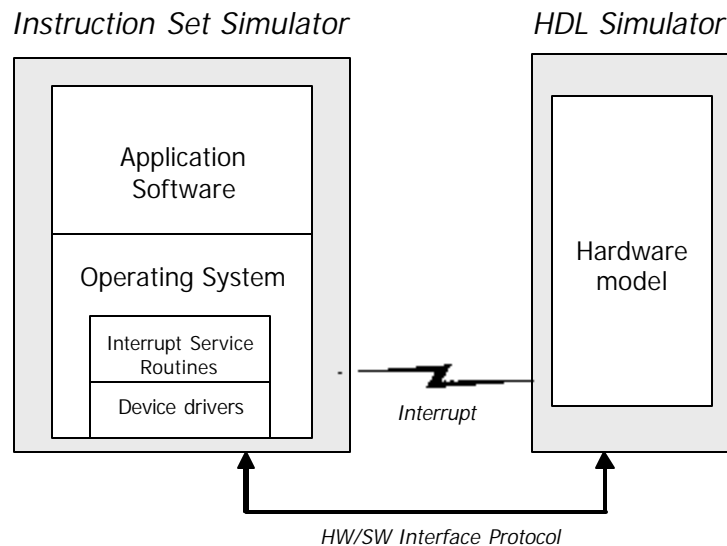


Figure 12: HW/SW co-simulation architecture.

This scenario, depicted in Figure 12 maps onto a co-simulation model, called ISS-centric, that consists of an ISS (inclusive of the host operating system) running the application, interfaced to hardware models, specified in a HDL at some levels of abstraction. The application is executed by the ISS, using the underlying software layers, and it accesses various devices by means of their drivers. The HW under design, specified as a HDL description executed by the

HDL simulation engine, is accessed through a device driver like any other device. In order to avoid the development of custom device drivers from scratch, this schema implements a memory-based communication between the ISS and the HDL simulator. It exploits the typical internal structure of an ISS. A generic ISS, regardless of the actual target of the simulation, maps the address space of the application and of the OS (which would correspond to physical memory) onto an internal data structure (which we will simply call "memory" hereafter). Memory can thus be considered as an "invariant" with respect to the target. The proposed HW/SW co-simulation uses this fact by considering the HW device as a memory-mapped device. Hence, by mapping device drivers in the memory space, an executable model of an entire board can be modelled, allowing the same SW developed for the board to run on top of the model. Indeed, even if the driver is related to a specific OS, whose execution is bound to a specific target, the mapping is preserved across different targets by the automatic mapping described above. Thus, the driver can be ported among different targets, because it relies uniquely on the memory mapping of the driven HW device. The actual implementation of this arrangement requires further details, and it implies modifications in both the ISS and the HDL simulator, as described in the Deliverable D4.1b.

## 4.1.2 Multi ISS

The multi ISS co-simulation mechanism is based on the communication protocol established between the two simulators through a straightforward message passing including data or timing information.

To summarize, the communication between the HW simulator and the ISS is realized by means of three new type of ports added to the SystemC library: `iss_in`, `iss_out` and `iss_interrupt`. The `iss_in` port is derived from the standard `sc_in` port and it is used to read data coming from ISS, while the `iss_out` port, extending the SystemC `sc_out` port, allows to send data from SystemC to ISS. These special types of ports are connected to the device registers, allowing the ISS to read and write them. The connection between the two sides of the co-simulation is performed by binding the ISS memory space to the SystemC `iss_in` and `iss_out` ports connected with the devices. Such a binding allows to assign the memory address of the HW device register to its `iss_in` and `iss_out` ports. Since the processor accesses to a HW controller interface through some memory register (control register, data register, command register, etc.), the SystemC interface of the HW controller must be designed with the special ports, connected to any HW register. Moreover, to handle the communication, the SystemC kernel has to resolve the address of the hardware register to the special ports of the hardware device involved. The link between the SystemC port and the associated memory address is performed by using a binding table stored in the SystemC kernel.

To support multi ISS simulation, the SystemC simulator has modified to manage the messages incoming from all ISS. First of all, the special SystemC `iss_port` (`iss_in/iss_out/iss_inout`) has been extended in order to know which ISS send/receive data on it. In such way, `iss_port` has to be defined including the ISS identification:

```
iss_port *reg1 = iss_port(0x12340000, ISS1)
```

Corresponding to this declaration, the following record is inserted into the binding table:

```
<reg1, 0x12340000, ISS1>
```

Moreover, the SystemC kernel has to create the communication channels to establish the connection with multi ISS; therefore an IPC channel is created for each ISS to be connected, as shown in Figure 13.

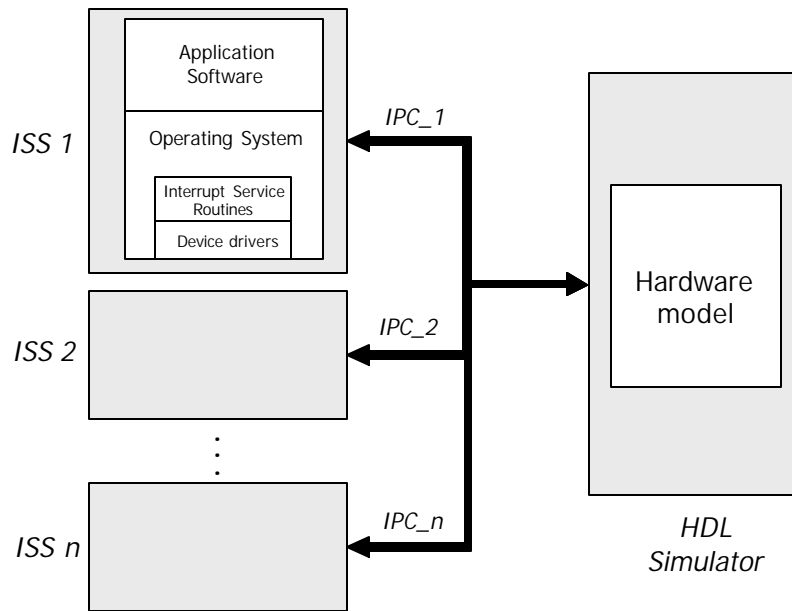


Figure 13: Multi ISS General Architecture.

Furthermore, it is essential to inform the HDL simulator about which ISS communicates on a particular IPC channel. Thus, during the setup phase the ISS sends a particular message to the SystemC simulator; the message encapsulates the ISS identification useful to the HDL simulator to update the binding table involving the port/memory address information. Up to this the binding is scanned and the related records to the ISS identification are modified as depicted in Figure 14.

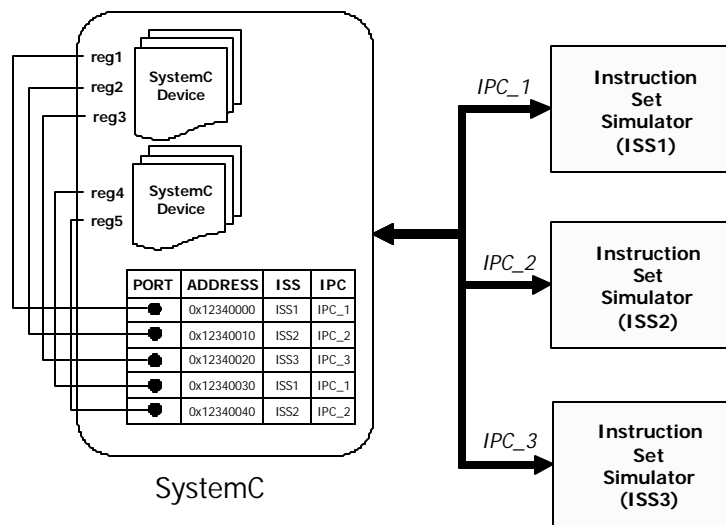


Figure 14: The communication between multi ISS and SystemC.

Figure 15 shows the pseudo-code of the SystemC simulator to support multi ISS simulation. Line 1-4 implement the setup phase previously described. After this phase, each ISS is able to send/receive data to/from the HDL simulator. In spite of the data exchanging is implemented as in the HW/SW co-simulation methodology described previously by using the time

information for the time-accurate co-simulation, in the multi ISS case the HDL kernel could receive co-simulation messages from different ISS. Therefore each IPC channel has to be monitored to verify the presence of an ISS request (Line 6 and 7). Corresponding to each ISS request, the SystemC kernel has to reply to the proper ISS selected by `iss_id` variable. Finally, the simulator extracts the event from the queue to schedule it. In the multi ISS case, SystemC has to find the ISS able to receive the response (Line 17).

```
1  for (iss_id=0; iss_id < ISS_NUM; iss_id++) {
2      create_IPC_channel(iss_id);
3      update_binding_table(iss_id);
4  }
5  do {
6      for (iss_id=0; iss_id < ISS_NUM; iss_id++) {
7          if ( !channel[iss_id]->isEmpty() ) {
8              receive(msg);
9              if (SystemC_Time < msg.ISS_Time)
10                 add_timed_event(<operation,
11                               msg.ISS_Time, ISS_Port, iss_id>);
12             else
13                 send_response_to_ISS(iss_id);
14         }
15     }
16     event = extract_event_from_queue();
17     if (event == "TIMED_EVENT")
18         iss_response = find_iss_by_event(event);
19         send_response_to_ISS(iss_response);
20     else
21         // NORMAL SystemC Kernel code
22 } while(...SystemC events...);
```

Figure 15: SystemC procedure to support multi ISS co-simulation.

Finally, the HW/SW co-simulation tool implements the interrupt mechanism as the original framework; obviously, the SystemC kernel has been modified in order to create the interrupt channel for each ISS during the co-simulation start phase previously described.

## 4.2 Automatic Generation of Device Drivers

---

This section describes rules for automatic generation of device drivers.

A device driver is a set of procedures (i.e., system calls) that allow higher-level user applications to interact with hardware devices. In particular, a device driver allows the user applications to access the hardware by ignoring all the low-level details specific to the device as, for example, communication protocols, memory mapping of the device addresses, etc...

A typical configuration of a HW/SW communication system, starting from the SW application to the HW device is shown in Figure 16.

The user application accesses the HW device by using the function calls provided by the device driver interface. Several functionalities are available to the application as, for example, auto-configuration calls (i.e., checking of which devices are available), I/O operations on the devices (i.e., open, read, write, close), interrupt handling, etc. The device driver implements the communication protocol by defining all the rules the software has to follow in order to access and synchronize with the hardware. How a device driver can be automatically generated is the aim of this work, and it is briefly described in the next subsections.

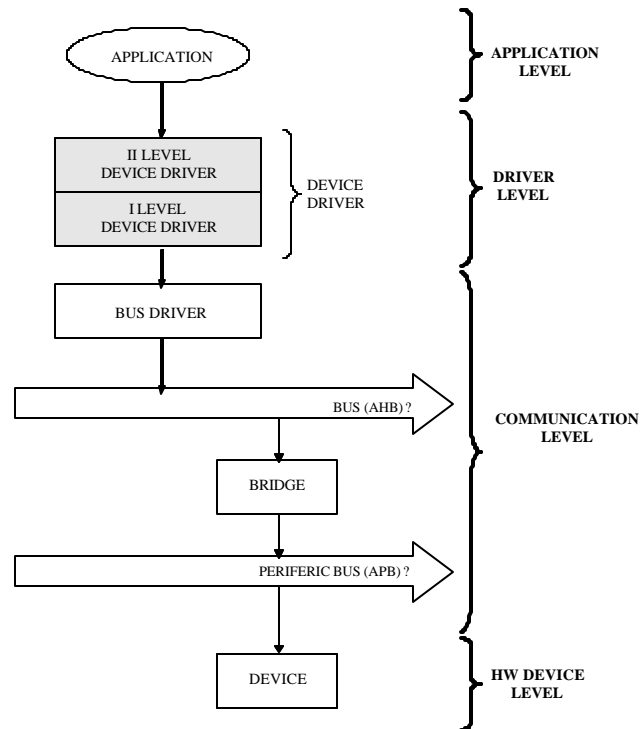


Figure 16: HW/SW communication levels

## 4.2.1 Multilevel Device Driver

We propose to split the driver implementation in two different levels (Figure 16) to make feasible and easier its automatic generation. Each level is composed of the following system calls:

- *Basic function calls.* The first level device driver consists of atomic and standard system calls that have direct access to the hardware (e.g., write(), read(), ioctl(), etc.) and they are called by the protocol specific functions implemented at the second level. Basic function calls generally rely on a standard structure and on a specific communication mechanism (i.e., memory mapped I/O), thus, they are typically independent of the device except for a reduced number of parameters related to the device memory mapping and that have to be set by the user.
- *Protocol-based function calls.* The second level device driver consists of system calls that implement the communication protocol of the device. They are called by the user application and they call a sequence of basic function calls depending on the specific device protocol to communicate with the device (i.e., handshaking sequences, data init, data write/read, etc.)

Device drivers can be automatically generated by defining a library of basic function calls and implementing protocol-based function calls as explained in the next subsections .

## 4.2.2 Generation of 1° level device driver

An example of basic function calls of an I/O device is shown in the follows:

```
/******  
 * sc_dev_read  
 * Read access on character device  
 * Example : cat /dev/scmem_dev  
*****/  
ssize_t sc_dev_read (struct file *filp, char *buf, size_t count, loff_t *pos)  
{  
    int data_mem;  
    printk(KERN_INFO "reading from scmem[%d]... \n",r_index);  
    (*address) = r_index;  
    (*command) = 0; // read command  
    r_index++;  
    wait_event_interruptible(wq, flag != 0);  
    flag = 0;  
    data_mem = (*data);  
    /* function to copy kernel space buffer to user space*/  
    if ( copy_to_user(buf,&data_mem,sizeof(data_mem)) != 0 )  
        printk( "Kernel -> userspace copy failed!\n" );  
    return 0; /* EOF */  
}  
/******
```

The function implements the *read* operation to a general device. The function is not specific to any particular device, as it can be called to access (in reading) to different devices just specifying target address, data type and size. Thus, the function can be customized for a specific device by properly setting the parameters specific to the device in a preliminary manual step. In particular, information regarding the *inode* in which the device is linked and the *memory addresses* in which the device is mapped have to be set by the user.

A library of this kind of *standard* basic function calls is implemented once for all. Depending on the device type, only the useful function calls are reuse and customized for the driver generation.

Once the preliminary step has been accomplished, the first level of device driver is generated, and a set of atomic function calls can be used in the body of the protocol-based function calls. In particular, they are called in a sequence that depends on the communication protocol, and that can be automatically extracted by the RTL implementation of the device.

## 4.2.3 Generation of 2° level device driver

We assume that the RTL testbench is available together with the RTL implementation of the device. The RTL testbench actually sends testvectors to and receives results from the device in order to test the different device functionalities. Thus, the RTL testbench performs an

ordered sequence of *write* and *read* operations in compliance with the device communication protocol and depending on the tested functionality. The methodology to extract the 2° level driver consists of three steps:

1. The formal model of the communication protocol (i.e., Extended Finite State Machine) is automatically extracted from the RTL code of the testbench. Each state of the EFSM represents a basic operation (i.e., a basic function call) to the device, while a path of transitions represents the sequence of operations corresponding to the protocol.
2. The EFSM subset representing different device functionalities (e.g., *write* and *read*) are identified by tagging the initial and final states visited during an access for sending data to or receiving data from the RTL device (see Figure 17). This provides the necessary support to extract information of the RTL protocol encapsulated in the testbench. Operations on the device functionalities are generally performed through the RTL interface by distinct EFSMs. Figure 17 shows an example of the tagging process. The initial and final state of EFSM performing *write* operations are tagged with *BEGIN WRITE* and *END WRITE*. Similarly, *BEGIN READ* and *END READ* tags are used to identify the EFSM performing *read* operations.

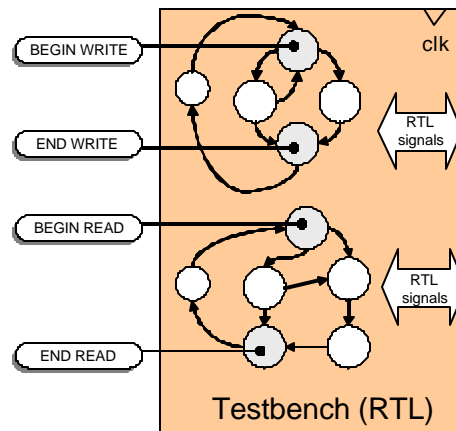


Figure 17: EFSM tagging

3. Each sub-graph of EFSM representing an operation on the device is translated into C code. It is composed with the protocol depending sequence of calls to the basic functions of the l level driver. An example is shown in the follows:

```

/*****
 * sc_dev_ioctl
 * ioctl access on character device
 *****/
int sc_dev_ioctl(struct inode *inode, /* include/linux/fs.h */
                struct file *file, /* ditto */
                unsigned int ioctl_num, /* number and param for ioctl
*/
                unsigned long ioctl_param)
{
    int addr;

```

```
int *temp_addr;
struct access_sc *temp_data_sc;
struct access_sc data_sc;
/*
 * Switch according to the ioctl called
 */
switch (ioctl_num) {
  case IOCTL_WRITE:
    for (w = 0; w < num_words; w++) {
      copy_from_user(&_sclink_data,
                    &(_k_memory_page->words[w]),
                    sizeof(uint16_t));
      // We build the data value
      _sclink_data |= (_page_size << 16);
      // We perform a write operation on sclink
      sc_dev_write(0, _sclink_data);
      break;
    }
  case IOCTL_READ:
    /*
     * Receive a pointer to a message (in user space) and set
that
     * to be the device's message. Get the parameter given
to
     * ioctl by the process.
     */
    ((struct ecc_memory_page*) arg)->ecc = sc_dev_read(0);
    break;

  return 0;
}
```

## 5. Mixed Static/Dynamic Verification (T4.4)

---

In this section it is described an example showing how Formal Verification has been used to supplement the classical simulation approach and to provide verification closure for the Alternate Function check at SoC level. In addition, automatic extraction of properties has been implemented for this very sensitive check in the ST automotive product family. Moreover, a mixed static/dynamic verification approach is presented which uses model checking to aid, or guide, the simulation process in certain situations in order to boost coverage.

### 5.1 Mixed Verification of System Integration Unit Lite (SIUL) module

---

The System Integration Unit Lite (SIUL) is the module used for the management of the pads and their configuration. It controls the multiplexing of the *Alternate Functions* used on all pads as well as being responsible for the management of the external interrupts to the device. The alternate function mechanism provides the needed flexibility of the final product - through different mapping sets over the pads - depending on the package and its pin configuration.

Given the high number of peripherals integrated in the automotive product, the management of alternate function becomes very complex and requires a specific verification.

Traditionally the *Alternate Functions* are checked by simulation at top level.

All of the *Alternate Functions* are implicitly checked during the peripheral stimulation in the SoC context, as shown in figure 14.

Each IP is stimulated by specific test in the SoC integration context. Each test generally runs separately from each others, as each test requires a specific Alternate Function setting.

In this way we can exercise a certain number of IPs in parallel, that do not share the same pads at the same time, and perform functional verification in the SoC context. Although this gives us a certain degree of confidence that the SIUL has been correctly implemented, we need to be sure that the setting of an Alternate Function affects only the selected IP and no other ones - that are not exercised at the time of the functional test and that are physically addressable by the pads in use.

Formal Verification provides the necessary support for that purpose.

In order to explain the performed check, it is necessary to introduce the DUT (Design Under Test) for this specific process.

The multiplexing logic implemented by the SIUL greatly depends on the physical characteristics of the PADS - driving capability, electrical protection, transition speed etc. - but, for the purpose of the physical connection, it may be described as a set of registers that store the current selection for driving/reading the PAD - i.e. the what pin of what IP is presently connected - and the value of the signal itself. In particular the SIUL may store the signal value and keep it available after the device that performed the write has been disconnected.

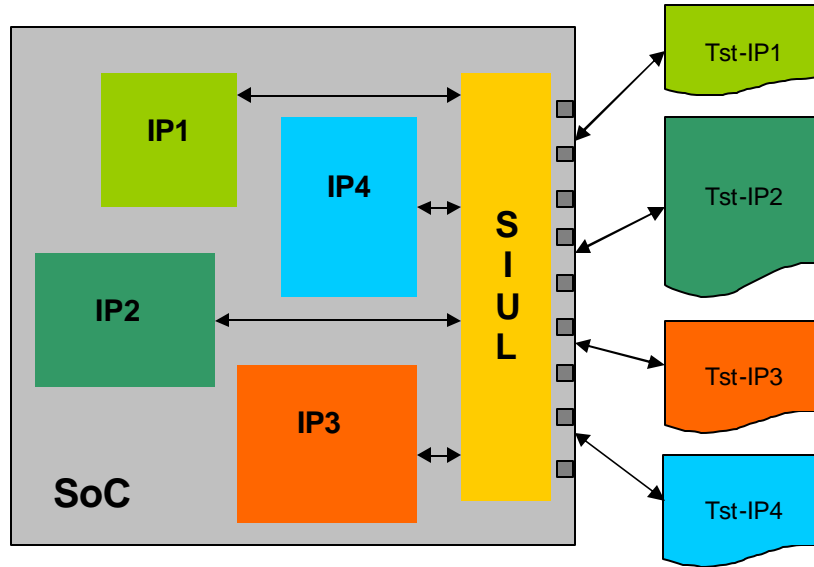


Figure 18: Testing IPs through Alternate Functions

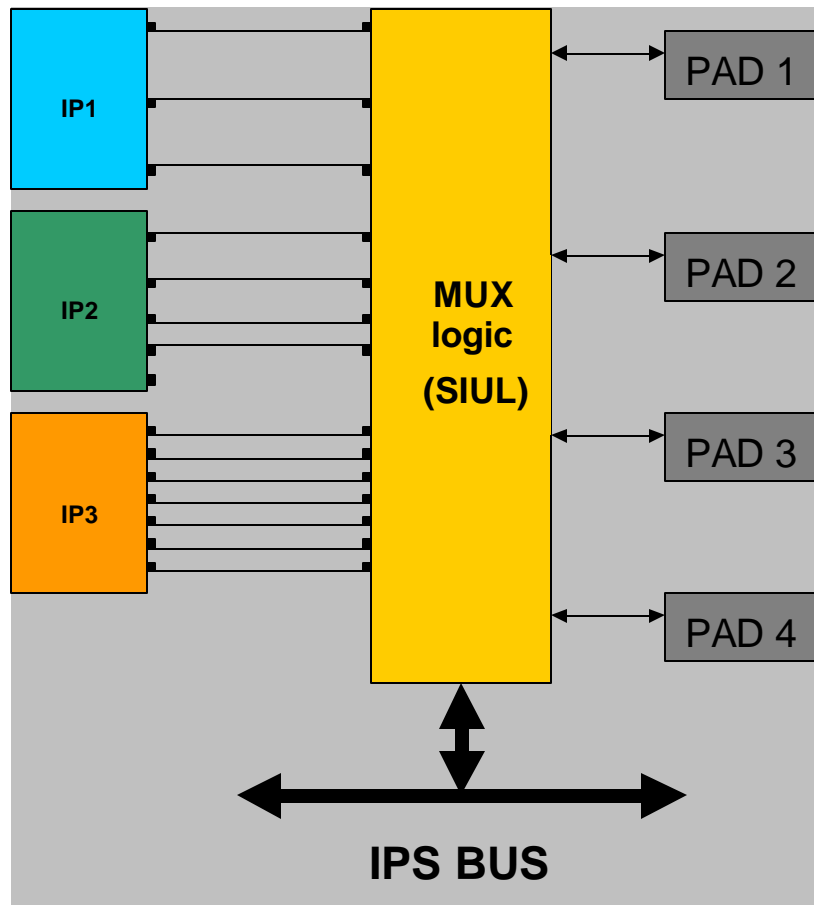


Figure 19: Loading the configuration in SIUL through the IPS bus

The SIUL configuration, i.e. the Alternate Functions currently set, is loaded via the peripheral bus – IPS BUS. The configuration registers themselves are mapped to a physical address, and the content that is set identifies a certain Alternate Function.

The formal proof mimics the configuration write on the bus and checks the consistency of the values between the IP pin and the PAD – thus providing the verification of the actual integration. For each connection a certain number of properties has to be set, as the involved logic includes registers for both the control and the data lines.

Therefore the real challenge becomes the generation of a huge number of properties needed for this check.

### 5.1.1 Automatic property generation

In this *Alternate Function Connection Check* we decided to introduce two main enhancements with respect to the previous setting:

- the use of Formal Verification to achieve 100% coverage
- the *automatic generation of properties starting from the specification*

In figure 16 the RTL generation process for the SIUL is sketched.

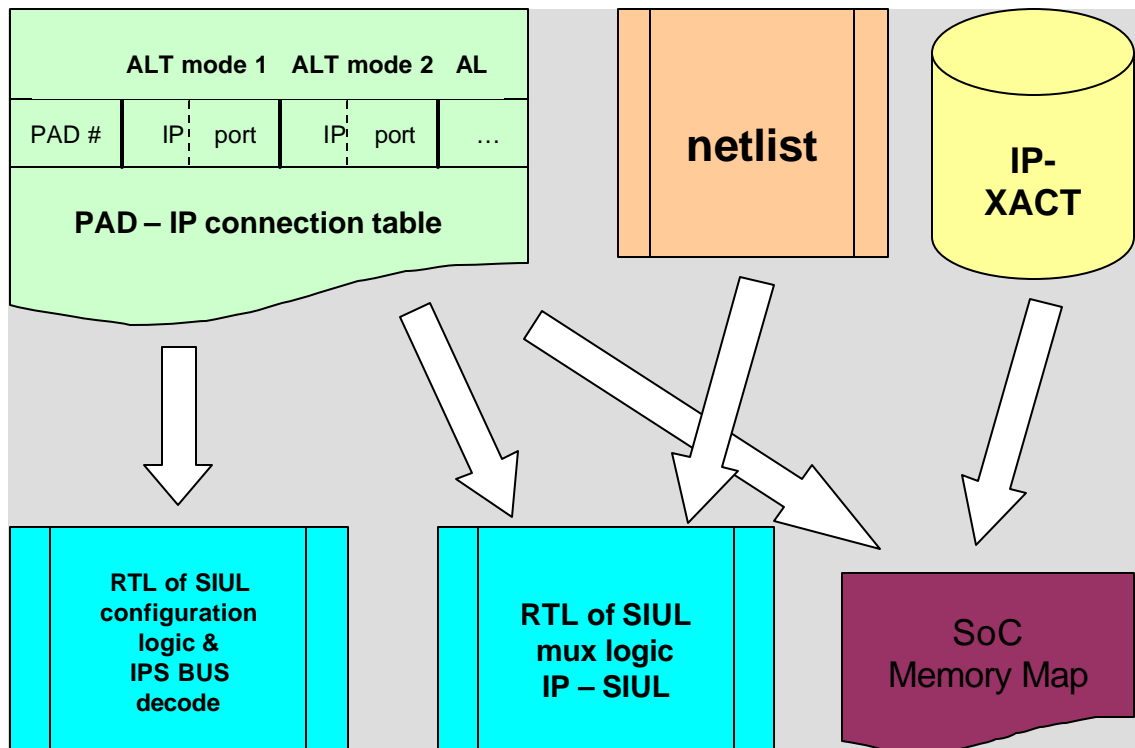


Figure 20: RTL generation process for Alternate Functions

The SoC architect specifies the different alternate functions at high level in the PAD – IP connection table – usually a file in .XLS format. Given the format in that table, it is possible to automatically generate the RTL implementing the registers that control the different PAD functions. The mux logic between the IPs and the SIUL is generated by another automatic process, that takes in input the connection table and the top level netlist connecting the IPs.

To close the process, the address map for each IP is extracted from the IP-XACT data base in order to produce the SoC entire memory map.

The automatic generation of properties, checking Alternate Function, takes in input:

- the PAD – IP connection table
- the SoC Memory Map
- the generated RTL

In this way we have two parallel automatic flows for the generation and the check of Alternate functions, both starting from the same .XLS specification document. In case of a real device some 2800 properties have been automatically generated and successfully proven.

## 5.2 Formal method-aided simulation

---

It is known that dynamic verification based on simulation suffers from the fact that they only examine a small fraction of the state space. Therefore, simulation results cannot be 100% guaranteed. Formal techniques, on the other hand, suffer from state space explosion and might not be practical for huge, complex systems due to memory and time limitations. We have therefore developed a mixed static/dynamic verification approach, which addresses some of the above problems.

In our approach, formal methods, in particular model checking, are used to aid, or guide, the simulation process in certain situations in order to boost coverage. The invocation frequency of the model checker is dynamically controlled by estimating certain parameters related to the simulation speed of the particular system at hand. These estimations are based on statistical data collected during the validation session, in order to minimise verification time, and at the same time, achieve reasonable coverage.

The objective of our mixed verification technique is to examine if a set of temporal logic properties, called assertions, are satisfied in the model under validation (MUV). As opposed to existing simulation-based methods, our approach is able to handle continuous time (as opposed to discrete clock ticks) both in the model under validation and in the assertions. It is moreover able to automatically generate the “monitors”, which are used to survey the verification process, from assertions expressed in temporal logic.

The developed technique imposes the following three assumptions:

- The MUV is modelled as a transition system, e.g. PRES+.
- Assertions, expressed in temporal logics, e.g. (T)CTL, stating important properties which the MUV must not violate, are provided.
- Assumptions, expressed in temporal logics, e.g. (T)CTL, stating the conditions under which the MUV shall function correctly (according to its assertions), are provided.

The assertions and assumptions described above constrain the behaviour on the interface of the MUV. They do not state anything about the internal state. The assumptions describe the valid behaviour of the environment of the MUV, i.e. constrain the input, while the assertions state what the MUV must guarantee, i.e. constrain the output. As mentioned previously, the objective of the validation is to examine if the MUV indeed satisfies its assertions.

The result of the verification is only valid to the extent expressed by the particular coverage metrics used. Therefore, certain measures are normally taken to improve the quality of the simulation with respect to the coverage metrics. This could involve finding corner cases which only rarely occur under normal conditions. Coverage enhancement is therefore an important part in simulation-based techniques.

The developed strategy consists of two phases : simulation and coverage enhancement. These two phases are iteratively and alternately executed. The simulation phase performs traditional simulation activities, such as transition firing and assertion checking. When a certain stop criterion is reached, the algorithm enters the second phase, coverage enhancement. The coverage enhancement phase identifies a part of the state space that has not yet been visited and guides the system to enter a state in that part of the state space.

In the simulation phase, transitions are repeatedly selected and fired at random, while checking that they do not violate any assertions. This activity goes on until a certain stop criterion is reached. The stop criterion is, in principle, a predetermined number of fired transitions without any coverage improvement.

When the simulation phase has reached the stop criterion, the algorithm goes into the second phase where it tries to further enhance coverage by guiding the simulation into an uncovered part of the state space. An enhancement plan, consisting of a sequence of transitions, is obtained and executed while at each step checking that no assertions are violated. It is at this step, obtaining the coverage enhancement plan, which a model checker is invoked.

The two phases, simulation and coverage enhancement, are iteratively executed until coverage is considered unable to be further enhanced. This occurs when either 100% coverage has been obtained, or when the uncovered aspects, with respect to the coverage metrics in use, have been targeted by the coverage enhancement phase at least once, but failed [7].

## 6. References

---

- [1] S. Cordibella, F. Fummi, G. Perbellini, D. Quaglia, "*HW/SW Co-simulation Framework for MPSoC Design*", Internal report - Università di Verona. Under submission to IEEE HLDVT 2008.
- [2] N. Bombieri, F. Fummi, G. Pravadelli, S. Vinco "*Automatic generation of device driver from SystemC descriptions*", Internal report - Università di Verona. Under submission to IEEE DATE 2009.
- [3] 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.
- [4] Karina Minakova, Uljana Reinsalu, Anton Chepurov, Jaan Raik, Maksim Jenihhin, Raimund Ubar, Peeter Ellervee. High-Level Decision Diagram Manipulations for Code Coverage Analysis, Baltic Electronic Conference, IEEE, 2008.
- [5] D. Karlsson, P. Eles, and Z. Peng, "Formal Verification of Component-based Designs," *Journal of Design Automation for Embedded Systems*, Vol. 11, No. 1, Mar. 2007, pp. 49-90.
- [6] D. Karlsson, P. Eles, and Z. Peng, "Transactor-based Formal Verification of Real-time Embedded Systems," *Proc. Forum on Specification & Design Languages (FDL)*, Barcelona, Spain, Sept. 18-20, 2007, pp. 1-6.
- [7] D. Karlsson, P. Eles, and Z. Peng, "Model Validation for Embedded Systems Using Formal Method-Aided Simulation," *IET journal on Computers & Digital Techniques*, 2008, in press.