

---

Research Report

# Combined Static and Dynamic Verification

Version 2 – Public Version  
March 31st, 2005

Jörg Bormann, Andrea Fedeli, Roy Frank, Klaus Winkelmann

(Deliverable 3.1/1)



## Notices

For information, contact [klaus.winkelmann@infineon.com](mailto:klaus.winkelmann@infineon.com).

The Prosyd organization provides this publication “as is” without warranty of any kind, either express or implied. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore this statement may not apply to you.

This publication may contain technical inaccuracies or typographical errors. While every precaution has been taken in the preparation of this document, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. may make improvements and/ or changes in the product(s) and/or the program(s) described in this publication at any time.

All company, product, trademarks and service names, are trademarks or service marks of their respective owner.

© Copyright Prosyd 2005. All rights reserved.

### Table of Revisions

Issue	Date	Description + Reason for the Modification	Affected Sections
0.1	April 1, 2004	Creation	
0.5	April 30, 2004	Integration of first draft from each partner	all
0.6	May 3 <sup>rd</sup> , 2004	result of phone conference	all
0.7	May 21, 2004	(based on v06_st), elaborated Infineon parts	1
0.8	May 26, 2004	(based on v07_st), elaborated IBM parts	1,2
0.9	May 27, 2004	(based on v08_ibm), rearranged sections, new: section 3.4 (JB)	2, 3, 4
0.10	June 7, 2004	(based on v09) minor typos fixed, changed one period structure, added figure (AF)	All
0.11	June 9, 2004	(based on v10), elaborated section 4.1 (RF)	2, 4
0.12	June 11, 2004	(based on v11) Just re-added 0.10 Description (Picture at the end of Sec. 2.1 still garbled).	-
0.13	June 14, 2004	(based on v12) Replaced picture at the end of section 2.1, erased redundant sentence in section 1.3.	2.1, 1.3
0.14	June 14, 2004	(based on v13) Fixed a reference to picture 2.	2.2.1
1.0	June 16, 2004	small corrections, added abstract	abstract
1.1	June 21, 2004	rephrased 2.2.2, added 3.4.2 (kw)	2.2.2, 3.4.2
2	March 31, 2005	public version, removed a few confidential paragraphs	3.3

## **Abstract**

This document describes important aspects concerning advanced combined static and dynamic verification flows. Important aspects of such a combined flow include modelling the environment, technology impact on properties, content of properties for the various design phases, and interaction of static and dynamic checking. Further we discuss methodology and management aspects as well as tool and language specific requirements.

## **Purpose**

This document serves as a first step towards a combined static and dynamic design flow.

## **Intended Audience**

This guide is intended for individuals interested in advanced hardware verification methods.

# Contents

<b>1</b>	<b>INTRODUCTION AND BACKGROUND .....</b>	<b>7</b>
1.1	Motivation .....	7
1.2	Today's dynamic verification flow .....	7
1.3	Today's static verification flow .....	7
1.4	Terminology .....	8
<b>2</b>	<b>OVERALL METHODOLOGY AND MANAGEMENT ASPECTS .....</b>	<b>8</b>
2.1	Planning the verification project .....	8
2.2	Problem decomposition .....	10
2.2.1	Property driven Verification Plan .....	10
2.2.2	Transaction Driven Verification Plan .....	11
2.3	Techniques for a more proficient verification, for today and tomorrow .....	12
<b>3</b>	<b>ADVANCED FLOW COMBINING STATIC AND DYNAMIC VERIFICATION .....</b>	<b>13</b>
3.1	Modeling the Environment .....	13
3.2	Technology Impact on Properties.....	16
3.2.1	Specification of the Model .....	17
3.2.2	Examination Window Length .....	17
3.3	Content of properties for various phases of design process .....	17
3.3.1	Implicit Assertions .....	17
3.3.2	Designer Assertions.....	18
3.3.3	Protocol properties .....	18
3.3.4	Functional properties.....	19
3.3.5	Integration Verification .....	19
3.4	Interaction of Static and Dynamic Checking .....	20
3.4.1	Use Case: First Simulate Then Prove.....	20
3.4.2	Use Case: Formal Proofs first .....	21
3.4.3	Use Case: Verify Environment Constraints by Simulation.....	21
<b>4</b>	<b>TOOL AND LANGUAGE REQUIREMENTS .....</b>	<b>21</b>

4.1	Ease of use .....	21
4.2	PSL flavor interoperability .....	22
4.3	PSL verification layer interoperability .....	22
4.4	PSL subset support .....	22
4.5	Disambiguation of unlocked property semantics .....	22
4.6	Operators: Reasoning Into the Past .....	22
5	<b>REFERENCES</b> .....	<b>23</b>

# 1 Introduction and Background

## 1.1 Motivation

There's a natural twofold motivation behind combined static and dynamic property verification which is on the one side the need to leverage coverage measures by means of static verification whenever applicable, and on the other side the need to obtain at least some data on those cases which, in spite of the most advanced reduction techniques, appear to be too hard to be attacked just with static methods. In other words, the goal of a combined static and dynamic verification methodology is to combine the "best of two worlds". The advantages of dynamic verification include, among others, the following:

- Amenability to larger designs than those that can be handled using static verification
- Smaller amount of proficient, dedicated knowledge needed
- Well-established tools and methodologies

On the other hand, static verification has several advantages over dynamic verification, including:

- Potential complete coverage
- Shorter setup times, contributing to shorter time-to-market

A methodology that would be able to combine the merits of static verification with those of dynamic verification will enable the design of high quality designs in a cheaper, quicker way.

## 1.2 Today's dynamic verification flow

The verification flows currently used by the three partners IBM, Infineon (IFX), ST Microelectronics (STM) are sufficiently similar to be presented here together. Aspects or tools which apply only to one partner will be labelled with the partners acronym.

Dynamic verification is today performed with a mix of C/C++ (SystemC keeping some pace), Specman *e*, a very little usage of Vera (STM, IBM) and vastly still using HDL language (STM, IFX). Testbench writing is an activity mainly performed directly by design teams.

Emulation and hardware acceleration are both quite popular as late RTL stage verification tools; their usage is typically offered as a service by dedicated central verification teams and tools.

Coverage is estimated using commercial tools, both by design teams and by verification teams. It mainly uses covermeter and VN-Cover (STM, IFX), Specman and Vera (IBM) or in-house tools, such as FoCs (IBM).

## 1.3 Today's static verification flow

In each of the three partners, static (formal) verification has become a de-facto standard for equivalence checking at RT and Gate levels, and is now widely used by design teams.

Property checking (including model checking as well as bounded model checking) is in the design flow just for specific (portions of) projects. In most cases, a dedicated team carries out the

model checking activities. At some projects at IBM and IFX, property checking is also carried out by the project's verification team.

At IBM, in addition to model checking, semi-formal techniques are also being used. Like model checking, semi-formal techniques are applied in specific projects and only for selected designs. Semi-formal activities are also carried out both by dedicated teams and by the verification teams in the relevant projects.

The role of coverage within the static verification differs between the partners: while it is generally perceived as a key issue for dissemination, the actual usage is specific to each partner:

- At IBM, there are known metrics for coverage in the dynamic verification environment, and tools are being used to collect coverage data. However, in static verification this issue is hardly addressed, as the units that go through it are also verified using dynamic verification, at least in a partial way, so the issue is addressed there.
- Infineon does not apply structural coverage measures to the formal verification, but uses a methodology aiming at functional completeness, based on writing properties sets whose assumption parts form essentially complete case splits.
- In STM, coverage has been seldom an issue in verification activities based on model checking so far, essentially due to lack of popularity of this technique. Coverage metrics have to be identified, and there must be a sort of general consensus on their significance.

## 1.4 Terminology

Some terms are defined here, for further terms such as **Property**, **Assertion**, **Constraint**, and **Coverage**: see PSL Language Reference Manual.

**Enabling / fulfilling condition**: Many assertions form implications. We will call the LHS of the assertion the enabling condition, and the RHS the fulfilling condition.

**Boolean assertion**: An assertion requiring a state predicate to hold forever.

**ABV**: assertion-based verification.

**DUT, DUV**: (used with same meaning) design under test, design under verification.

## 2 Overall Methodology and Management Aspects

### 2.1 Planning the verification project

What will be described here is a suggestion for the way a verification project should be managed, based on the experience of IBM design centers that have been using both static and dynamic verification techniques very effectively over a long period of time.

- Even prior to the verification stage, when the architect has finished the high-level architecture of the design and it should be divided into blocks, some planning is made and static verification vs. dynamic verification considerations are being made. Blocks are designed such that most of them will be applicable for static verification. This is achieved by dividing the design into small blocks, with well-defined interfaces between them, etc.
- When a new piece of hardware is transferred to the verification team, the verification lead plans which parts of it will be verified with dynamic verification techniques, and which will be verified with static verification techniques.

- ✍ Some rules of thumb that are used in this planning stage:
  - 1) 40% of design blocks will be formally verified
  - 2) Formally verifying an average module
    - takes about a month
    - takes about 20% of designer's time for support
      - a designer with 4-5 modules going through Formal verification is “blocked”
    - requires allocating 2-3 processors for running the formal verification tool
  
- ✍ Typical modules that make good candidates for formal verification (static checking)
  - Lots of control, little data
  - Coverage is hard to reach through simulation (e.g. arbiters)
  - Modules that are buried inside the design, far from the simulation interfaces
  
- ✍ Typical modules where checkers are automatically produced from PSL code (dynamic checking)
  - Protocol checking of chip interfaces and partition interfaces
  - Protocol checking of interfaces between designers
  - Reuse of assumptions and rules from modules verification
  - Coverage is measured for all properties that are manufactured in this way
  
- ✍ Where automatically produced checkers are not considered
  - Abstract data checking (packet structure, data comparison)

Figure 1 below describes how PSL code is translated automatically into checkers that go into the simulation flow.

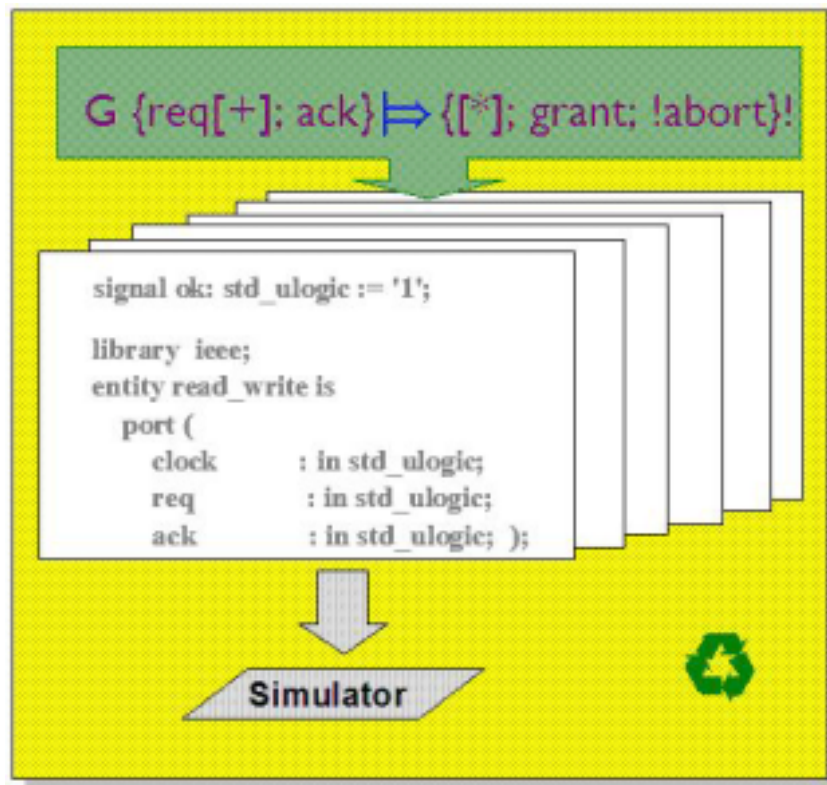


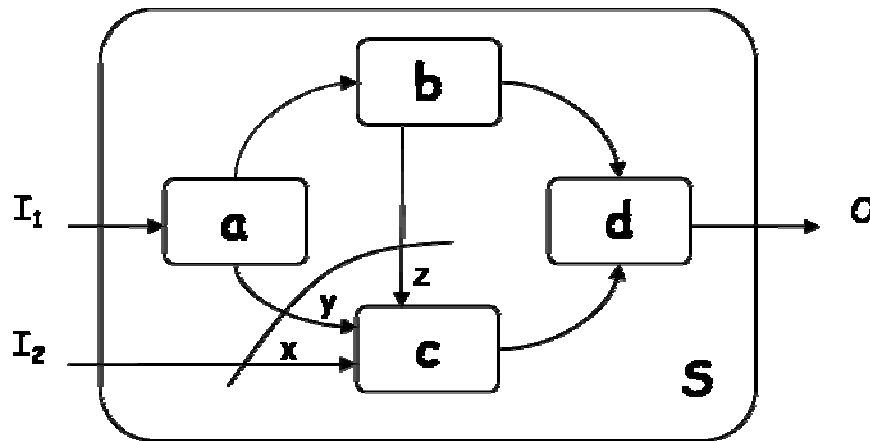
Figure 1: PSL code is translated automatically into checkers that go into the simulation flow

## 2.2 Problem decomposition

The making of a verification plan that is aimed at maximizing reuse between static and dynamic worlds has to be orchestrated so that the size of single elements into which the original verification problem is decomposed can be handled by the most restrictive technique. This implies that, in principle, the problem decomposition should be driven by the tractability limits proper of the verification engines utilized on the Static side, as those are in general the most suffering of problem size. Static engines try to shrink the size of the problem to be dealt with applying various reduction techniques, which, on their turn, try to exploit information derived either by environment constraints in action (constant propagation, constant simulation, input Boolean correlations, etc.) or by the nature of the properties the system should fulfil (e.g. cone of influence).

### 2.2.1 Property driven Verification Plan

Properties are, in fact, a fundamental term to be taken into account when decomposing a problem, as, in absence of advantageous reductions, if a compositional approach is adopted (cf. deliverable D1.1/2 Chap. 6) all the block input signals a property makes some correlating assumption about must be outputs of the same block (ref. Figure 2).



**Figure 2: Atomicity of information sources for assertions in a compositional context. The c assumption  $x \wedge y \wedge z$  cannot be proved as a property of neither a nor b.**

The decomposition of the original problem can be expected to be addressable, to a certain extent, by the verification tool automatically, even though it is not difficult to imagine cases where the tool cannot find a good decomposition by itself and the verification engineer has to provide hints on choices to make.

A big amount of properties, possibly spread around the design (i.e. pertaining to different aspects of the design behavior) could help a tool in finding effective system partitioning to allow compositional verification; to reach this goal the combination of designer's properties, more aimed at specificities of single blocks, and of verification engineer's properties, usually targeting larger checks than those of single block, can provide an effective framework for decomposition heuristics; this require, though, that just properties expected to pass have to be present in designer's property set, which is not necessarily true due to the usage designers could decide to make of properties. Alternatively, tools should offer an easy to use feature to allow masking of unwanted properties, and –possibly– some analysis capabilities to suggest the verifier which properties are likely to be avoided. The mix of Verification engineer's properties, presumably derived by some documentation about what the design should be doing from an input/output point of view, (that is the so-called *black-box verification*), and of those details about what's inside each block (the so-called *white-box verification*) is in fact a combination of a top-down technique, in which problems are partitioned according to their global definition, and of a bottom-up approach, in which smaller, more treatable problems are combined to form a bigger picture.

### 2.2.2 Transaction Driven Verification Plan

Having properties as the basic element of the verification plan is good for static/dynamic reuse. But from other perspectives is it often neither practical nor desirable, for several reasons:

- First the right decomposition is unknown at the beginning of the design phase, as system portions have still to be built and there will be nothing against which top level properties can be checked.
- Further, refinement of design is often checked more against a set of behaviour expressed in operational terms rather than in denotational form. I.e. it uses an *executable specification* (i.e. a *prototype*, something that can be simulated and played with) .

- Finally, the decomposition information which emerges along the system design process needs to be kept consistent. In absence of an automatic verification assistant, this task could become prohibitively complex for a human being.

This all is a consequence of the dualism between model-based design and property-based design. Currently (2004) the first approach is largely preferred to the second, while the ProSyD project aims at giving a boost to the second term of the dualism.

Nevertheless, there is space to apply formal techniques even in the context of model based description as follows:

- represent the communication between system elements and their surroundings (whether internal or not) through *language* formalization, i.e. by recognizing the structure of the communication among the parts (finding out the *tokens* of the communication and their relation in terms of a grammar) and
- keep checking containment relations between languages along system evolution and corresponding language refinements.
- Furthermore, the grammar productions and token refinements can be utilized to drive design refinement in a natural top down development flow.

That is the Transaction Level Reasoning (TLR), which goes in pair with Transaction Level Modeling (TLM), and (roughly yet) depicts a framework for reasoning on transactions even before they get implemented into a timed model, just exploiting the pre-order relation fragments that each transaction description brings with itself.

### **2.3 Techniques for a more proficient verification, for today and tomorrow**

Assertions have already been available in HDLs for quite a while; temporal assertions represent a novelty, but not of a radical kind; usage of Boolean assertions is often left as an option to the designer, and are quite rarely used. More than a matter of introducing a new language, assertion-based verification is more about a way to organize the design and verification activities in conjunction. Designer's aim is, and must remain, to build a design; that being his/her main goal, it comes to no surprise that documentation, usually produced after the design is complete and meant for those that will have to use the design as a description of *what* the design is supposed to do and, to some degree, *how*. As that *what* is exactly the intended subject of properties to be written, an extra effort should be made to let that properties emerge, if not before, at least during the design process. This requires the ability to look at the same problem from two different viewpoints. When the design effort is not trivial (e.g.: a new kind of functionality is to be built, or a strong optimization constraint have been imposed by the committer) there is enough in the design activity per se to not ask the designer the extra mental shift needed to continuously swap between problem view angles; rather, the very simple solution which has practically shown to be really effective is to let designers work in pairs with one of them dedicated to the verification activity whilst the other keeps designing with the two roles swapped on a regular basis. This is essentially the idea of *Extreme Programming* (XP), which has gained some popularity in the field of software production in the last few years [Beck00]. The ideal scenario of a couple of designers writing temporal assertion and designing in turn is a bit of a visionary due to the restricted familiarity that most designers have with temporal logic to date; hence, instead of pointing directly to *Extreme Design*, there could be space for a temporary elbow-to-elbow collaboration between a single designer and a verification expert, trained in usage of Assertion Based Verification systems that should

- Read the available specification
- Interview the designer

- Collect extra specs
- Translate everything relevant into properties
- Run the verification as soon and frequent as possible

the really important aspect is that this collaboration happens from early stages of the design activity and doesn't stop until the design is frozen. The experience has shown that exposing designers to this type of collaboration for the lasting of even a single project is enough to have the most of them be convinced of the validity of the approach and to be eager to repeat the experience for the design coming next. Such an Extreme *Verification*, where the substitution scheme (design/test) of XP does not apply, as the designer and verification roles are not interchangeable, is to be used until the knowledge on how to apply assertion based verification has been transfer to a significant designers basis. As a reference for pros and cons of extreme design and extreme verification (with the exclusion of everything linked to swap for the latter) [Be00] can be used as a starting reference; other aspects linked to this approach are still to be investigated.

### **3 Advanced Flow combining static and dynamic verification**

Assertion-based verification is said to bridge between static and dynamic verification. Limitations to this claim will be discussed in the sequel (3.1, 3.2). Keeping these limitations in mind, appropriate methodology will be presented. This methodology differs between the various application domains of assertions. These domains will be discussed in section 3.3.

#### **3.1 Modeling the Environment**

The environment modeling activities are quite different in the two verification worlds (cf. section 2.3.1), mainly due to design size and secondly due to nature of the intent. The place where reuse appears quite natural is at block level verification, where the modelling of the environment is to be made explicitly both for dynamic and for static checks. In spite of structural similarities, the current situation is such as depicted in Figure 3, where the environments written for dynamic verification and for static verification are seen as two independent entities.

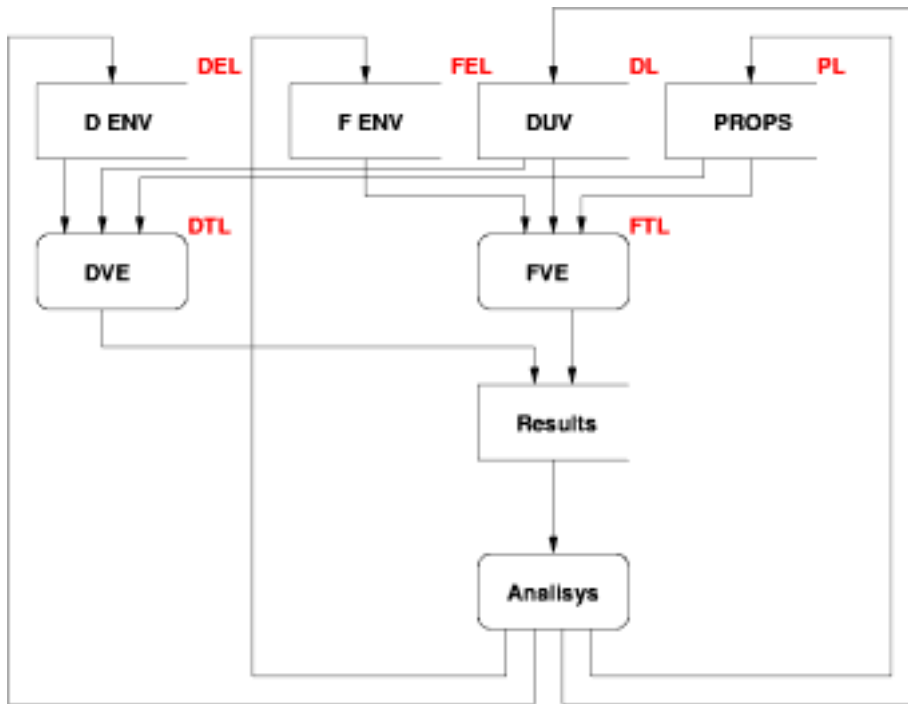


Figure 3: The Current scenario for block verification. In it

**DEL** : Dynamic Environment description Language

**FEL** : Formal Environment description Language

**DL** : Design description Language (HDL)

**PL** : Property description Language (PSL)

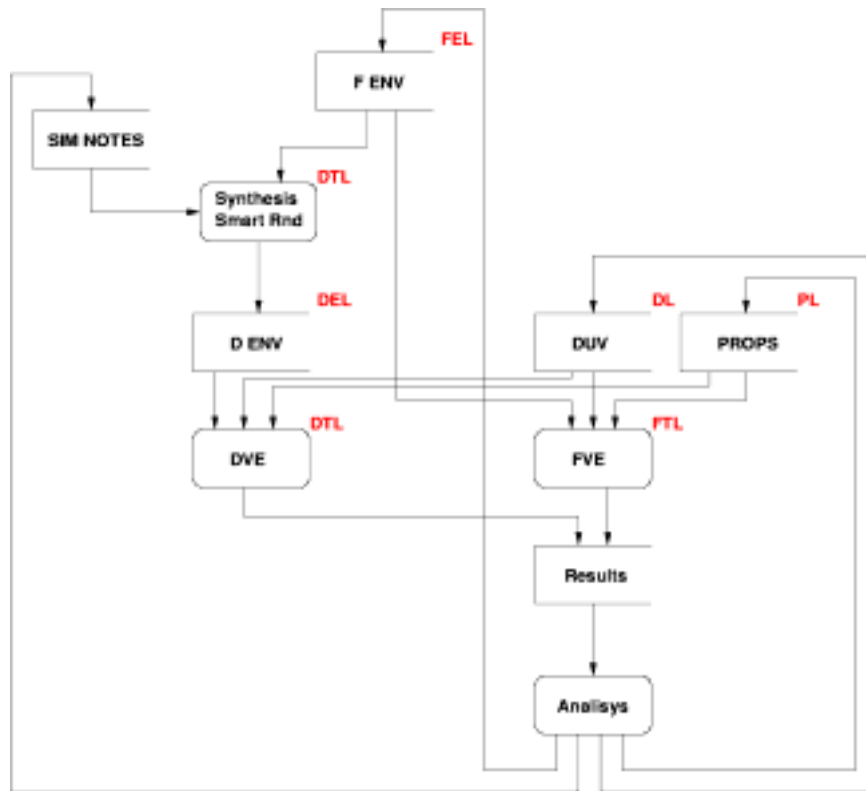
**DTL** : Dynamic verification Tool Language (Simulator control language)

**FTL** : Formal verification Tool Language (Model checker control language)

**DVE** : Dynamic Verification Engine

**FVE** : Formal Verification Engine

Having to write two different models to represent the same object (the environment) exposes the verification process to potential inconsistencies and requires to double the modeling effort: to avoid such a repetition it is foreseeable (ref. Figure 4) to adopt a single description environment approach, augmented where in case to deal with extra simulation-oriented information, like value range random distribution biasing, which do not have to be considered in formal verification,. Not all the different implications of such a single-entry flow have been identified hitherto; whether such an approach can be turned into reality is still an open question, but we perceive it as a central term in the widespread adoption of formal methods for block verification.



**Figure 4: A single-entry environment block-level verification scenario**

The flow of Figure 4 brings with it a computational complexity issue to be addressed possibly in ProSyD (cf. task 3.2/11): describing the environment by means of assertions has, in general [KV97] an impact on model checking as it results to have an exponential complexity in the size of formula representing the conjunction of the assumptions; as the same complexity results holds also for the construction of the maximal model identified by the assumption, which could be ideally used as a generator of sequences (satisfying by construction the given assumptions) for the dynamic verification task, the right decisions have to be taken on how to solve the sequence generation problem starting from a set of assertions, which is indeed necessary if we want to guarantee the usability of a single environment description. In practice this could mean to impose a restriction on the possible assertion shapes for those assertions to be used as assumptions to keep the computation load under control.

A totally different scenario is to be considered for system level verification, typical of those cases in which property checking is applied at a later stage of conception. In that case, there is no more need for an environment building on the dynamic verification side.

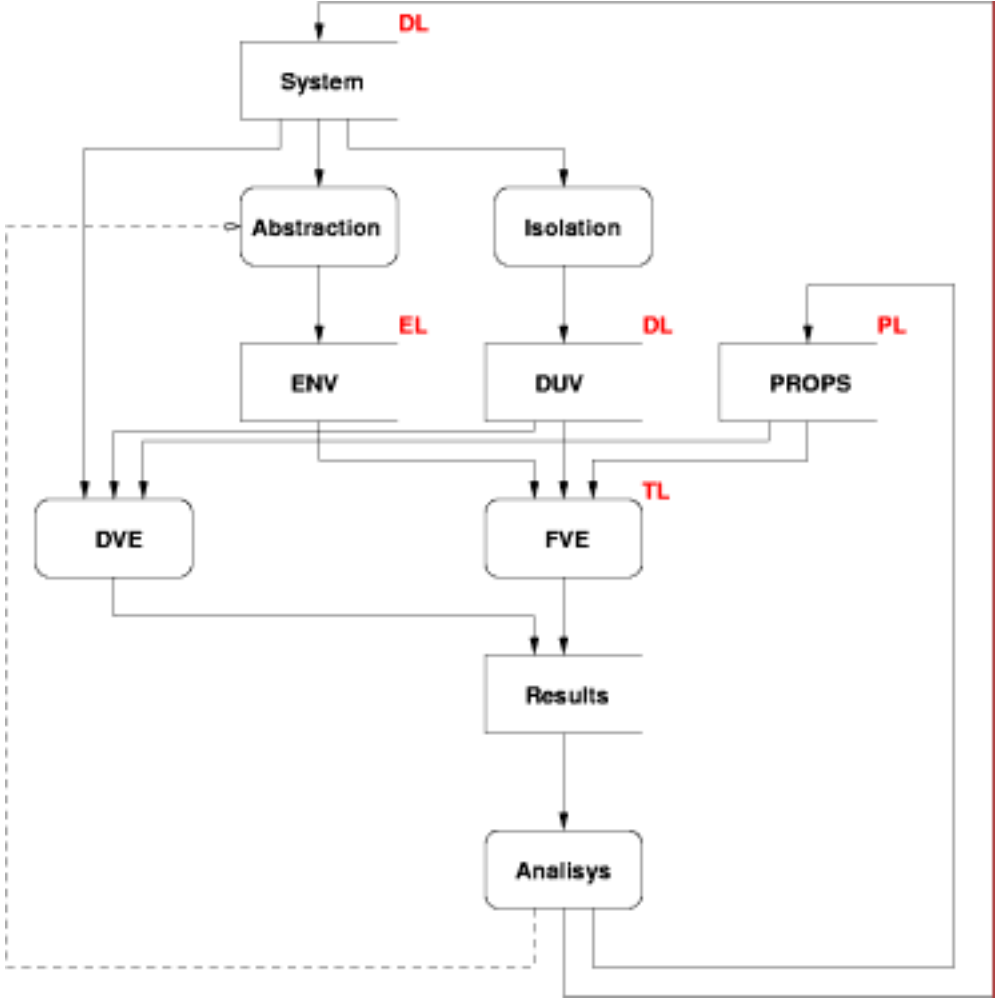


Figure 5: The System Verification Scenario

In Figure 5 a couple of processes, abstraction and isolation, are used to apply formal verification on those portions of the system which are, likely, more critical from a correctness point of view (for safety, security, or other reasons); as the abstraction and isolation processes will have to be performed on a case by case basis, the only methodological recommendation that can be given is to define a mechanism to easily identify block level properties that designers put in their works with the intent to capture relevant design aspect from the system perspective. The analysis of conjoint dynamic and static verifications results will enable the identification of a more suitable abstraction of the environment to be used in formal verification sessions.

**3.2 Technology Impact on Properties**

Properties can be examined by simulation, iterative property checking, and bounded property checking. Each technology imposes its own requirements on the property specifications. It should be examined to which extent these requirements contradict each other.

### **3.2.1 Specification of the Model**

Some of the environment constraints described in the section above depend on the configuration of the top level that is being used for formal verification. If the top level contains a module that is responsible for a certain input behaviour, there is no need for an input constraint that describes this input behaviour. In turn, the complexity of a proof task may be severely reduced by black boxing (i.e., excluding some modules from the formal examination and treat their interface signals as primary inputs and outputs). It is therefore necessary to back up assertions with information about how a formal proof should be executed.

For simulation, all these issues do not exist, since the module is executed as it would be executed in a system environment.

### **3.2.2 Examination Window Length**

The examination window of a property starts at the first point of time where the property has a nontrivial state predicate and ends at the last such point.

For bounded property checkers the examination window size is always bounded. For simulation and for iterative property checkers the examination window may have unlimited length. Consequently, bounded property checking can only be executed on a suitable subset of PSL. This subset does not include liveness properties, the [\*] repetition operator in sequences and many other constructs.

## **3.3 Content of properties for various phases of design process**

The content of properties for the various phases of the design process are described in this section.

### **3.3.1 Implicit Assertions**

Implicit assertions are specified through the use of constructs in the RTL that may render the code inconsistent (see Fig. 7 for some examples). Some of these assertions are checked by simulators, some others are not, but workarounds exist to make simulators check them. A third class of such assertions cannot be checked by a single simulation, but only after the examination of several simulation stimuli. Formal verification of these assertions can be done with an Infineon tool called GateCheck. This delivers a proof, a counter example from reset, or an indecisive situation, where both an incomplete proof of the assertion on some number of clock cycles after the reset state, and a possibly unrealistic counter example are returned.

## Implicit Assertions: Simulation & GateCheck

(*) must be satisfied whenever the case statement gets executed	Example Code	Implicit Assertion	Support by	
			Sim	Gate Check
array access	<code>reg_array(adr)</code>	<code>reg_array'left &lt;= adr and adr &lt;= reg_array'right (*)</code>	yes	yes
assignment to subtype	<code>y &lt;= v</code> , where type of y is subtype of v	y contains subtype element (*)	yes	yes
full case pragma	<code>case (s) // synopsys full_case // synopsys parallel_case a: .... b: .... c: .... endcase;</code>	<code>s == a    s == b    s == c (*)</code>	no	yes
parallel case pragma	<code>onehot_or_zero( {s == a, s == b, s == c} ) (*)</code>		no	yes
bus contention	logic signal sig with multiple drivers	<code>sig /= 'X'</code>	no	yes
floating bus		<code>sig /= 'Z'</code>	no	yes
dead code	<code>if (a) begin ... end;</code>	assertion violated, if <code>a == 0 (*)</code> holds.	no	yes
constant signal	Declaration of a signal sig.	assertion violated, if <code>sig == prev(sig)</code> holds.	no	yes

Figure 7 - Implicit Assertions: Simulation and GateCheck

### 3.3.2 Designer Assertions

While coding the designer should produce properties that help during verification. Such properties are

- Assertions that form plausibility tests for the code (e.g. about decoders)
- Coverage points that highlight known dangerous situations (e.g., FIFO getting full or empty)
- Assertions about signal relations that a designer regards as important but not obvious (e.g., value of registers in idle state).
- Constraints: Input constraints that are exploited by the circuit that is being designed (e.g. SDRAM interface that relies on the stability of the address to extract row and column addresses at different points in time)

Designer assertions are often Boolean assertions and as such they are easily examined by simulation. Despite their simple shape these assertions can create heavily differing verification efforts. These efforts range from a quick formal verification of the plausibility assertion about a decoder to consequences of system wide invariants. While the plausibility assertions can be checked almost without effort, the consequences of system wide invariants may require a careful choice of lemmas, which are then assembled to prove the original goal. We will present a methodology that keeps track of the differing demands of apparently very similar properties.

### 3.3.3 Protocol properties

Assertion based verification allows a concise and intuitive description of protocols. This description provides assertions and constraints about modules and - if not trivial - about the communication infra structure. The advantage of assertion-based verification over state of the art protocol checks by monitors is

1. The direct use of the protocol assertions in verification. The protocol description can be used without changes for the verification purposes, while the state of the art approach requires its implementation by a monitor.
2. The support of a transition from simulation to formal verification.

Protocol properties should be defined by either the designers whose modeules communicate over a common protocol, or they should be defined by the concept engineers.

### **3.3.4 Functional properties**

Functional properties aim at describing the functionality of a module more closely. The goal is to write so many properties that every bug will be identified shortly after its occurrence. Another goal is to justify difficult designer properties.

Functional properties are described most likely by verification engineers, possibly building on hints given by the designers.

The enabling conditions of functional properties typically form interesting functional coverage points, because they show during simulation, how often the functionality described in some functional assertion is being executed.

One methodology for functional properties can start with the description of the functional coverage points and describe how they are extended to assertions. Another methodology may require to first prove functional properties in a module verification and then to reuse their enabling conditions as functional coverage points for system level verification.

### **3.3.5 Integration Verification**

The definition of the properties above gives rise to constraints, mainly about the inputs. These constraints form requirements that must be met in order to integrate the module with others. This allows for an integration verification before and after the integration (cf. next figures).

If integration verification is done after integration, as usual, this means:

- Environment constraints of one module become assertions of the neighbouring modules
- It is sufficient to integrate DUT1 and DUT2 into one simulation. No additional user interaction is required.
- but there is one drawback: Integration verification after integration requires all modules to be coded

Verifying before integration means:

- Environment constraints of one module are turned into proof goals of the neighbouring modules.
- This requires explicit integration of these inherited proof goals.
- Assertion Management is required to ensure updating the proof
- If DUT1 and DUT2 are completely formally verified, this integration verification suffices.

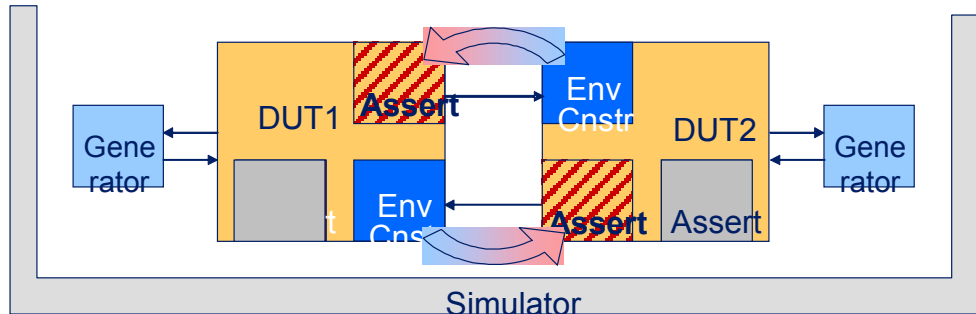


Figure 8 Integration Verification after integration

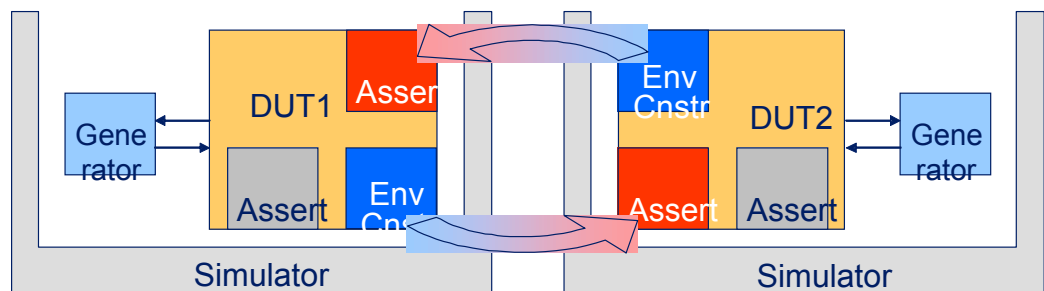


Figure 9 Integration Verification before Integration

### 3.4 Interaction of Static and Dynamic Checking

There are ideas around to integrate static and dynamic checking into one proof engine (e.g. Amplified Simulation from 0-in). Besides these ideas an integration may also take place on a more methodological level, exploiting the technology independency of assertions. We list three potential use cases in the sequel.

#### 3.4.1 Use Case: First Simulate Then Prove

Designers will verify their assertions by simulation first. This will provide some confidence that these assertions capture the designers intentions. The formal verification step will then be used to make sure that the assertion will not only hold for the simulated cases, but that it will hold always. In current simulation based verification environments the formal step will require additional information as e.g. environment constraints for the proof to succeed. It can therefore be expected, that first runs of formal tools will lead to counter examples. The analysis of these counter examples is simplified through the confidence that was already gained by the previous simulation steps. It can be expected that the counter example highlight a bug or a missing environment constraint.

### 3.4.2 Use Case: Formal Proofs first

As opposed to the previous use case, formal verification can be applied directly instead of simulation, as the main method for module level correctness. While this requires a deviation from current standard practice, experience indicates that it is very beneficial: design errors are found earlier and the time to market decreases. IBM reports a comparative study of formal vs non-formal verification, which showed that with the formal approach, bug-finding finished earlier. This is consistent with other partners' results.

### 3.4.3 Use Case: Verify Environment Constraints by Simulation

To make sure that the proof of assertions in a module is relevant in a system level context, it must be verified that the environment satisfies the environment constraints of that module. A related formal proof of the environment may not always be feasible or efficient. E.g. if the environment constraints are about the instruction sequence of a processor, the environment that is responsible to satisfy this constraint is the code generator of some compiler, which cannot be treated by formal hardware verification, but which is easily accessible by simulation. In this situation the environment constraints that were used to formally verify the assertions of the module will become assertions for a simulation based chip or system level examination.

## 4 Tool and Language Requirements

### 4.1 Ease of use

One of the main difficulties in the deployment of static verification methods is that it is commonly believed that the application of static verification requires a very high level of proficiency and that those who carry it out must be very experienced experts, having a considerable academic and theoretic background. On the other hand, dynamic verification methods are widely spread and are usually well established and integrated into the design flow of the organizations. In this sense, dynamic verification is considered to be a task that every design/verification team can perform, and does not require any special level of proficiency.

In a combined static and dynamic verification methodology, this gap should be bridged. This problem should be tackled from a couple of angles:

- The language angle

The temporal logic language that is used to describe the properties that need to be verified should be a language that is powerful enough to express very complex properties, but on the other hand simple enough so that an average verification engineer could be capable of learning it and putting it to use after a short time of studying it (two to three weeks).

The layered structure of PSL holds this requirement. While PSL has the capability of expressing very complex assertions, its syntax of sequences is powerful enough to express most of the assertions that a novice verification engineer would need.

- The tool angle

The commonly used algorithms in formal verification, such as SAT-based algorithms, BDD-based algorithms etc., use a considerable amount of parameters, that have a high impact on the performance of the algorithm. Therefore, if a user gives a 'good' set of parameters to the tool, he is a lot more likely to get the results than if a 'bad' set of parameters is used. This fact makes it more difficult to use formal verification tools, as in order to get successful runs of the tool, one would need to be a proficient and experienced user, who knows what is the set of parameters that is likely to be 'good' for the algorithm.

This problem can be relieved by adding the ability to choose a 'good' set of parameters automatically to the tool. According to the design, the specifications and possibly other inputs, the tool's front-end can choose the parameters that are to be given to the formal verification engine, thus releasing the user from this task.

Another way of using the tool to reduce the level of needed proficiency is by manufacturing properties automatically. Some of the properties can be extracted by the tool automatically, such as properties that the designer requires implicitly ('Implied Intent'). For example, the tool can manufacture properties to verify that every output line can hold both '0' and '1', given the appropriate input sequence.

#### **4.2 PSL flavor interoperability**

The PSL standard definition in its current v1.1 draft incarnation, does not impose the equality of the HDL flavor used in properties and of the HDL used for the DUV; to allow reuse of properties it will be an obvious prerequisite that the same decision is taken for dynamic and static implementation of parsers; at the same time, to empower property reuse it would be preferable to have the two language be completely disjointed. To date (May 2004) only the verilog flavour is officially supported in PSL, hence that should be the chosen language, to date.

#### **4.3 PSL verification layer interoperability**

To empower reuse of properties dynamic verification environment should also be able to deal with the object hierarchy proper of the PSL verification layer; as depicted in Figure 4, an extra element, currently outside the scopes of PSL, will have to be forecast to guarantee the handling of those information which are not relevant in static verification and plays instead an essential role for effective verification in the dynamic context, e.g.: random distribution shaping. Tools should be able to get a PSL description, augmented with whichever needed information, and treat it in both dynamic and static worlds with no need for intervention on the property files, whether they hold extra information.

#### **4.4 PSL subset support**

To guarantee property portability, the PSL utilized should be restricted as much as possible to the *simple* PSL subset (cf. [PRM04], sec. 4.4.4)

#### **4.5 Disambiguation of unlocked property semantics**

Currently PSL allows to write properties without referring to a clock, even if no default clock is specified. It is then left to the tool what the semantics of such a property is.

This means that such unlocked properties cannot be shared between tools without the danger of meaning different things. For formally verifying asynchronous designs a solution of this semantics problem is required.

#### **4.6 Operators: Reasoning Into the Past**

Infineon experience indicates the PSL  $\rightarrow$  and  $\Rightarrow$  implication operators of PSL are insufficient because they require the enabling condition to lie temporarily before the fulfilling condition. To our experience it must be possible to temporally mix enabling and fulfilling conditions. This can be achieved with the  $\rightarrow$  sequence implication operator, but this operator is not part of the simple subset of PSL and - as most simulator vendors strive to support only the simple subset - properties containing this operator cannot be checked by simulation.

## 5 References

[Be00] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison Wesley Longman, 2000

[KV97] Orna Kupferman, Moshe Vardi, *Modular Model Checking*, in Proceedings of the COMPOS'97 International Symposium, Lecture Notes in Computer Science, Springer, 1997.