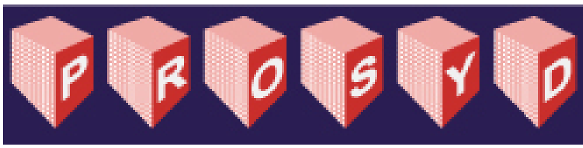

FP6-IST-507219

PROSYD

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Case Studies in Property-Based Verification

(Deliverable 3.4/1)

Due date of deliverable: 31 December 2006
Actual Delivery date: 31 December 2006

Start date of project: 01.01.2004 Duration: 3 years
Organisation name of lead contractor for this deliverable: STMicroelectronics Italy
Revision: 1.1

Project co-funded by the European Commission within the Sixth Framework Programme (2000-2006)

Dissemination Level

PU	**Public**	☒
PP	**Restricted to other programme participants (including the Commission Services)**	☐
RE	**Restricted to a group specified by the consortium (including the Commission Services)**	☐
CO	**Confidential, only for members of the consortium (including the Commission Services)**	☐

Notices

For information, contact andrea.fedeli@st.com.

This document is intended to fulfil the obligations of the PROSYD project concerning deliverable D3.4/1, described in contract number 507219.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

© Copyright PROSYD 2006. All rights reserved.

Table of Revisions

Version	Date	Description and Reason	By	Affected Sections
0.1	**7 November 2006**	**First Draft**	**Andrea Fedeli**	**New**
0.2	**10 November 2006**	**Migration to official template. Minor aesthetic interventions.**	**Andrea Fedeli**	**All**
0.3	**22 November 2006**	**Case studies added, few typo fixing,**	**Andrea Fedeli**	**All**
0.4	**11 December 2006**	**Full Case studies set, few typo fixing, uniformed citation style.**	**Andrea Fedeli**	**All**
0.5	**22 December 2006**	**Addressed comments from Cindy and Anthony; included new OneSpin's Version and IBM's.**	**Andrea Fedeli**	**All**
0.6	**22 December 2006**	**Chapters 7 and 8 completed**	**Andrea Fedeli**	**7, 8**
0.7	**23 December 2006**	**Uniformed Captions**	**Andrea Fedeli**	**All**
0.8	**26 December 2006**	**Fixed ST-F typos revived due to Synchronization issues, Summary Table in chapter 8 slightly reviewed.**	**Andrea Fedeli**	**4, 8**
0.9	**28 December 2006**	**ST-F new version and point review of OneSpin contribution**	**Andrea Fedeli**	**2, 4**
0.10	**30 December 2006**	**ST-F minor changes; manual breaks to improve readability, final round on references.**	**Andrea Fedeli**	**All**
1.0	**31 December 2006**	**approval by project management**	**Cindy Eisner**	**Version Number**
1.1	**9 January 2006**	**Correct errors in relevance column of summary table, rewrite associated text accordingly**	**Cindy Eisner**	**8-2**

Authors

Lyes Benalycherif,

Andrea Fedeli,

Anthony McIsaac,

Mark Moulin,

Ohad Shacham,

Klaus Winkelmann,

Emmanuel Zarpas

Executive Summary

This report describes twelve case studies on a wide range of designs and communication architectures that had been under development in a number of large semiconductor companies during PROSYD life. In the case studies, property-based

verification in PSL are constructed, guided by the methodology documents on property specification D1.1/1 [11], on property reuse D1.1/2 [10], on static/dynamic verification techniques combination D3.1/1 [1], and on using different checking engines D3.1/2 [2]. Property verification activities have been supported by tools described in Workpackage 3. It is shown how the verification activities have been quantitatively and qualitatively affected by PROSYD outcomes: the contribution of the tools is evaluated, and the overall productivity benefits of a property-based verification approach are assessed.

Purpose

The purpose of this document is to describe the work done in case studies on property-based verification using PSL; to assess the contribution made by the PROSYD tools developed in Workpackage 3; to indicate how the methodology framework applies in industrial applications; and to report on the overall productivity benefits.

Intended Audience

This report is intended for individuals who work with or have responsibility for VLSI design projects. It is assumed that readers are familiar with the notions and terms related to VLSI design and verification, and the main features of PSL.

Background

The industrial partners (IBM, Infineon, STMicroelectronics at its 3 sites), and later OneSpin, chose case studies from their current activities so as to cover a wide range of designs and architectures, of sufficient size and complexity to provide challenges for property verification and the tools supporting it.

Previously, properties have been used chiefly as elements in a verification plan, being directed either at complex parts of the design, where coverage of a wider range of scenarios can be achieved through model checking, or at specific assertions which can be checked at little extra cost in the course of simulation. The applications in Workpackage 1 went beyond this, to basing the entire specification of a design on properties; that effort, integrated with the work done to tackle more peculiar practical verification aspects has lent to the material presented in the current document, meant to let partners show how and to which extent verification tools have improved thanks to PROSYD.

Contents

Table of Revisions ... iii
Authors ... iii
Executive Summary ... iii
Purpose ... iv
Intended Audience ... iv
Background ... iv
Table of Figures ... viii
Table of Tables ... viii
Glossary ... ix
1 Introduction ... 1
2 OneSpin's Case Studies ... 2
2.1 Protocol Processor ... 2
Design Characteristics ... 2
Goals ... 3
Tools used ... 3
Work done ... 3
Results ... 4
R&D Value ... 5
Productivity enhancements ... 5
2.2 TriCore2 Processor ... 6
Design Characteristics: ... 6
Goals ... 6
Tools used ... 6
Work done ... 6
Results ... 6
R&D Value ... 7
Productivity enhancements ... 7
2.3 SLIM Core Processor ... 7
3 ST UK's Case Studies ... 8
3.1 Protocol Converter ... 8
Design Characteristics ... 8
Goals ... 8
Tools Used ... 9
Work Done ... 9
Results ... 10
R&D Value ... 11
Productivity Enhancements ... 11
3.2 Transport Front End ... 12
Design Characteristics ... 12
Goals ... 12
Tools Used ... 13
Work Done ... 13
Results ... 14

R&D Value 16
Productivity Enhancements 16
4 ST F's Case Studies 18
4.1 Bus Protocol 18
Design Characteristics 18
Goals 18
Tools Used 18
Work Done 18
Results 19
R&D Value 20
Productivity Enhancements 20
4.2 Memory Interface 20
Design Characteristics 20
Goals 20
Tools Used 20
Work Done 21
Results 24
R&D Value 27
Productivity Enhancements 27
5 ST I's Case studies 29
5.1 Memory Controller 29
Design Characteristics 29
Goals 29
Tools Used 30
Work Done 30
Results 31
R&D Value 31
Productivity Enhancement 31
5.2 Bridge 32
Design Characteristics 32
Goals 32
Tools Used 32
Work Done 32
Results 33
R&D Value 33
Productivity Enhancements 33
5.3 AMBA 34
Design Characteristics 34
Goals 34
Tools Used 34
Work Done 34
Results 35
R&D Value 36
Productivity Enhancements 36
6 IBM's Case Studies 37
6.1 Interconnection Bus 38
Design Characteristics 38
Goal 38
Tools used 39
Work Done 39

Results 39
R&D value 39
Productivity enhancements 41
6.2 Cache Arbiter 42
Design Characteristics 42
Goal 42
Tools used 42
Work Done 43
Results 44
R&D value 44
Productivity enhancements 44
6.3 Scheduler 47
Design Characteristics 47
Goal 47
Tools used 47
Work Done 47
Results 48
R&D value 48
Productivity enhancements 48
7 Assessment of Methodologies and Tools 50
7.1 D3.1/1 Combined Static and Dynamic Verification Methodology 50
7.2 D3.1/2 Effective leverage of Different Static Checking Engines Methodology 50
7.3 D3.2/7 Improved Static Property Checking (IBM) 51
7.4 D3.2/8 Improved Static Checking (OneSpin) 51
7.5 D3.2/9 Improved Static Property Checking (NuSMV) 52
7.6 D3.2/11 Generation of Observers 52
7.7 D3.2/14 Analysis of Simulation Traces 52
7.8 D3.2/15+D3.2/16 Embedded PSL in Verilog (Static+Observers). 53
7.9 D3.2/17 Fast Simulation 53
8 Conclusion 54
8.1 Q&A 54
8.2 Summarized Measurements 56
9 References 59

Table of Figures

Figure 5-1: Result distribution of partial verification for the Memory mapping check of the AMBA case study: x represents the number of steps, y the number of properties. 35
Figure 6-1: The Interconnection Bus....................................38
Figure 6-2: The Cache Arbiter. 42
Figure 6-3: The Scheduler in its context. 47

Table of Tables

Table 3-1: RBPE Engine capacity comparison. 11
Table 3-2: Engine comparison for PID filter....................................15
Table 3-3: Effort on components of Transport Front....................................16
Table 4-1: Rulebase multiengine use measure sample....................................25
Table 4-2: NuSMV use measure sample. 25
Table 4-3: Data transfer RuleBase property checks measure sample. 26
Table 4-4: Total effort (specification effort) spent per hardware component. 28
Table 5-1: Passing, Failing, and Explored depths for the 67 assertions of the Bridge (Opteron 2.8GHz, 8GB, RHEE3.0, timeout = 1h). 33
Table 8-1: Grading of productivity enhancement results: Columns refers to designs relationship, rows to verification techniques (new/old). 56
Table 8-2: Case studies Summary: compared techniques, designs relationship, and relevant outcomes. 57

Glossary

AHB: the Advanced High-performance Bus, a part of the *AMBA* hierarchy of buses specification.

AMBA: the Advanced Microcontroller Bus Architecture, ©ARM Limited 1999.

APB: the Advanced Peripheral Bus, a part of the *AMBA* hierarchy of buses specification.

Assertion: 1.A *property* that the *DUV* has to fulfil. 2. In some contexts (e.g. in dynamic verification), synonym of *Property*.

Assumption: A *property* that the *DUV* verification takes for granted; assumptions are properties used to model input behaviours.

Assertion Based Verification: A verification task using *assertions* as fundamental elements to check *DUV* correctness. Can refer either to *Model Checking* or to Simulation instrumented by assertion checkers.

AXI: AMBA 3 Advanced eXtensible Interface, ©ARM Limited 2004.

BDD: Binary Decision Diagram. A data structure upon which some static checking engines are based.

Behaviour: A succession of states of a design.

Block: A component in a design. A block may contain instances of other blocks.

Boolean satisfiability: The problem of determining whether a Boolean propositional formula has a satisfiable assignment, that is, an assignment of its variables under which the formula evaluates to "true". Also referred to as SAT.

BMC: Bounded Model Checking. A method of model checking in which a limited number of cycles is examined. Typically, a bounded model checker can falsify, but not verify, a design.

Design: A model of a piece of hardware, described in some hardware description language (HDL).

DUV: Design Under Verification. The subject of the verification process (it has to be a model for the set of *properties* used as *assertions*).

Dynamic Verification: *Verification* based on logic *simulation*. A dynamic verification is meant to check whether the design behaves as expected on a single execution trace at a time.

Environment: The part of a system surrounding the *DUV*. Often represented in an abstract form (i.e. describing only what strictly needed to let the verification converge towards a close result, passing or failing, leaving all the other information left unspecified) it is described using the modeling layer of *PSL.*

Flip-flop: A hardware element storing one bit of information.

Formal Verification: *Verification* based on the usage of a formal technique; in PROSYD context this is mainly used a synonym for *model checking.*

GDL: Generalized Description Language, a *PSL* flavor. Originally conceived to describe the environment for *formal verification,* and due to this highly more concise and readable than its HDL-related siblings, it has the drawback of being the only PSL flavor not currently supported by commercial mainstream simulators. Inside PROSYD it has been largely used for specification and verification by model checking.

HDL: Hardware description language.

LTL: Linear-time logic. LTL is a subset of PSL Sugar that allows one to specify properties of a single infinite path.

Model Checking: The automatic or almost automatic verification of a property of a model of a hardware component or in some cases of software.

Property: A collection of logical and temporal relationships between expressions involving design signals, that represents a set of behaviours.

Property Checking: 1) Syn. of Model Checking. 2) (less usual) syn. of ABV.

Protocol: A set of rules governing communication between participants in a system.

PSL: Property Specification Language, the language for specification of designs upon which PROSYD is based. It comprises 4 different layers: Boolean, Temporal, Verification, and Modeling, used to express, respectively, propositions on events at a given time, temporal relationships among propositions related to possibly different times, properties to be proven or to be assumed for a given verification task, and environment behaviours.

SAT: A shorthand for Boolean satisfiability, or an algorithm for deciding Boolean satifiability.

Specification: The process of defining the expected behaviour of a hardware design.

Testbench: The combination of DUV, stimulus generation, expected output estimation, and output comparison. It is usually meant to be used in the context of an HDL simulator.

Verification: The process of falsifying or verifying the functional and performance requirements of a design, be it chip, board or system. Many different kinds of verification tools are in use today, including simulation, formal verification, emulation and rapid prototyping.

Verilog: One of two standardized hardware description languages used to specify the structure and behaviour of electronic systems in textual format.

1 Introduction

In the following chapters, thirteen individual verification case studies are described in detail. In each case, there is a general description of the design, including an indication of its size and complexity, and any features of technical interest from a verification viewpoint. This is followed by an account of the actual work done, including technical information about the sorts of properties that were written, paying specific attention to quantitative information aimed at comparing results obtainable before PROSYD and after PROSYD.

The outcomes of the case studies are described in terms of results (what was achieved in terms of verification), R&D value (novelty of techniques or how they have been applied) and productivity enhancements (either reduction of effort in the verification stage as compared with methods not based on properties, or improvements in overall verification quality or in the parts of the process targeted by specific tool features). Case studies are gathered on a per partner basis, into 5 consecutive chapters, from 2 to 6; chapter 7 is devoted to tools assessments and chapter 8 draws conclusions, summarizing productivity enhancements quantitatively, and addressing a few qualitative questions there were identified during PROSYD review meetings as peculiar verification aspects.

2 OneSpin's Case Studies

2.1 Protocol Processor

Design Characteristics

Infineon developed an application-specific processor dedicated to offloading its protocol drivers from the general purpose processors on the chip and running this peculiar workload separately. After a first version of this processor core was successfully employed, a second version (PPv2) was re-engineered from scratch and, still retaining backward compatibility, extended for wider and more effective usage. This solution, implemented as a reusable IP block supported by IPR, has found wide acceptance throughout the full range from DSL to router chips in the COM wire-line product lines.

The PPv2 architecture is based on a 32-bit RISC pipeline of seven stages, which supports a set of 40 instructions optimized towards bit-level manipulation and data transfer.

The PPv2 supports multithreaded execution with up to four communicating contexts, which can be switched through without any constraints, overhead or response delay. This feature enables up to four "virtual machines" to run on the same architecture and, whenever one of the machines must wait for a response from the periphery and stall, another one may run: the absence of overhead and constraints decouples the software implementation from the underlying switching activity.

Contexts can be arbitrarily switched and/or restarted to any starting point in the program by external events as well as by a dedicated instruction. The context restart can be employed as branch as well, but the architecture features several other kinds of branch instructions for transferring the control flow to any point in the program: these are grouped in "short" branches, which decide whether to branch or not depending on the value of some status flags, and "long" branches, which take their decision after performing a more complex operation such as a decrement and a test of a register value. Every branch instruction incurs intrinsically in a penalty, i.e. a number of cycles in which the processor executes no instructions. This is two cycles to restart a context, three for a short branch and four for a long branch. The PPv2 architecture supports "delay slots", which is a popular technique to insert instructions in place of penalty cycles. Since delay slots increase the processor's performance but tend to bloat the program size, the PPv2 core offers the possibility to turn for every single branch instruction any number of penalties into delay slots, thereby allowing a fine trade-off between performance and code size. The microarchitecture takes care of solving complicated conflicts, which arise from composing those operations under external exceptions and/or in the delay slots of each other.

The complete design amounts to roughly 11000 lines of VHDL code, and 7179 FFs.

The description above should provide an insight in how difficult it can be to reach acceptable test coverage by simulation. Moreover, a flaw in such a programmable IP would have been replicated in all projects reusing it, jeopardizing the ongoing developments not only for the hardware, but also for the firmware as well as for the processor tool-chain which was being developed concurrently to the hardware. Therefore, a solution was mandatory to avoid upfront such a risk and still catch up with the ongoing projects while retaining R&D efforts similar to previous ones.

Goals

The goals of this case study were

- measure productivity enhancements in an industrial context
- demonstrate fast simulation concept

The technical challenges included:

- Ensure the correct pipelined processing of multiple instructions, guaranteeing no undesired interferences between instructions; and ensure the correct operation of permissible – but unpredictable – behaviours such as traps and interrupts.
- Guarantee transparency of pipelining together with related forwarding and stalling behaviour to the software programmer.
- Comprehensively verify data paths with complex bit-manipulations.
- Meet hard real-time requirements such as trap- and interrupt-execution within permissible latencies.
- Ensure independent execution of multiple threads under all possible combinations of instructions, thread switches and permissible, but unpredictable, behaviours.
- Precisely describe the integration requirements of any hardware/software environment into which the PPv2 would have to be integrated.

Tools used

- D3.1/2 [2] Static Checking Engines Methodology.
- D3.2/8 [13] Improved Static Checking Tool.
- D3.2/17 [16] Fast Simulation.

Work done

A total of 14 person months was spent on this case study (specification 2, design 2, verification 5, evaluation and reporting 5).

The property set developed for the processor was verified with OneSpin's property checker. This checker internally combines several SAT- as well as BDD-based engines. According to the findings documented in D3.1/2, no way is known to predict the performance of particular algorithms on a given design. The Protocol Processor experience confirms the observation that with SAT algorithms, failures are usually detected really fast, while successful proofs often take longer – e.g. a typical property of the protocol processor is proven in 10 minutes, while a false version of the same property is reported to fail in less than one minute.

An important question that was addressed with this case study is:

What bugs (type and quantity) are found using this formal flow?

These turned out to be:

- complex control issues,
- interacting state machines,
- corner cases,
- overflow handling,

An example of such a complex behaviour is that of a branch instruction, for which a static parameter configures execution of up to four delay slots. Upon receiving an interrupt, these slots must be stored, and then executed after the interrupt. This sequence must behave correctly in a wide range of configurations.

The verification detected a configuration in which the slots were only partially executed:

> This is caused by an interaction of a "long" branch and a context switch triggered by an interrupt (IRQ). The "long" branch has 4 delay slots, and as the result of a bug, in case of context switch during the branch operation, slots 3 and 4 are forgotten, i.e. never executed. It is obvious that without deep knowledge of what to look for, such a bug would be almost certain to escape a simulation test-bench.

Using the techniques presented in D3.2/14 [7], properties were also evaluated on a set of simulation traces generated by a standard directed test-bench. As the properties were proven, no failure was to be expected from this. However, in fact all proofs were carried out using certain assumptions about the environment, and the role of checking the traces was to validate those assumptions. In the case of the protocol processor, environment assumption relate to assembler restriction, i.e. certain instruction sequences are specified as illegal. Indeed the analysis of simulation traces revealed one such restriction which was violated in certain cases, and helped to further improve the design.

Technically, this analysis was carried out by generating VHDL monitors from the assumptions, and running those along with the design, with a standard VHDL simulator. Note that the development work for automatically generating these monitors was done outside the PROSYD project, while its application to the case study is part of PROSYD.

Results

As the main result, we developed a property set capturing the full processor functionality. In addition, these properties form an executable specification, which has been simulated using the approach described in D3.2/17[16] (Fast simulation).

OneSpin's Module Verifier provided the PPv2 development with a verification solution resolving all of the project's verification objectives and enabling "true functional sign-off":

- The MV solution ensured error-free functional operation of the entire PPv2 including all possible configurations. Thus correct PPv2 operation could be guaranteed for all hardware/software environments that meet three well-defined integration requirements.
- This was achieved with five engineer-months of effort. The total verification effort was 40% less than that of PPv1.

- Moreover, the verification infrastructure comprised only one standard workstation and the property checker. The entire property suite ran easily overnight on this workstation, enabling immediate verification of code changes.

Verification was based upon:

- 11k lines of unsimulated, newly developed VHDL.
- An informal specification of 130 pages exhibiting a certain amount of ambiguities/unspecified situations.

Results of the verification were:

- A list of 10 serious errors and 17 issues leading to completion of spec, redesign of the context switch logic, better understanding of architectural particularities and sharpened integration requirements.
- After elimination of errors and open issues a suite of 38 proven properties described unambiguously the PPv2's cycle accurate behaviour; on less than 40 well-readable pages.
- Proofs secured that VDHL correctly implements spec and no functionality has been overlooked.
- The entire property suite runs less than 3 hours (excluding completeness check) on a standard workstation allowing for immediate verification of code changes.
- A formal, executable programmer's view resulted as a by-product; this transaction-level model of just 10 pages is provably sequentially equivalent to the VHDL.

R&D Value

For the first time, an industrial processor design relied completely on formal verification. This case study offered a very good situation for comparing productivity, as a previous version was verified by simulation, so that comparison data are available (see below).

Further, both the monitor generation and the fast property simulation were for the first time validated on a realistic design.

Productivity enhancements

The overall effort for specification and verification by the new, property-based approach was 7 person months, vs. 12 which were needed with a simulation-based approach. This amounts to a ~40% reduction overall.

What is more important, the number of remaining bugs in integration test was zero, as confirmed by a large set of system integration tests, while in the previous version 20 bugs had escaped.

2.2 TriCore2 Processor

Design Characteristics:

The TriCore™ is the first unified, single-core, 32-bit microcontroller-DSP architecture optimized for real-time embedded systems. The TriCore™ Instruction Set Architecture (ISA) combines the real-time capability of a microcontroller, the computational power of a DSP, and the high performance/price features of a RISC load/store architecture, in a compact re-programmable core.

The case study addresses the load/store unit (LSU) of TriCore2.

The LSU is the interface to the Memory Management Unit. Its function is to set control signals, control data width, and handle alignment issues.

The LSU processes 2 instructions, each in 13 addressing modes. Its size is 6KLOC, 1969 FFs.

Goals

The main goals for this were

- To test the reuse methodology for properties as outlined in report D1.1/2 [2] (Reuse-Aware property-driven specification). Detailed results are documented there.
- Explore the influence of various specification styles on tool efficiency.

Tools used

- D3.1/2 [2] Static Checking Engines Methodology
- D3.2/8 [13] OneSpin Improved Static Checking Tool, as a target for the specification

Work done

A total of 8 person months was spent on this case study (specification 2, verification 2, evaluation and reporting 4).

A total of 26 properties were written, totalling approx. 900 lines of code.

The property run-time was measured, and used as part of a test bench to compare various tool versions of D3.2/8 [13] (Static Checking).

This was used to tune the proof engines and other algorithms involved.

In addition, making use of results reported in D3.1/2 [2], we studied how a property can be re-formulated for a bounded checker.

Results

For productivity enhancements achieved with the improved static checking tool, see the separate section below.

For the issue of bounded checking, several answers resulted:

- Pragmatically, properties can be broken down into smaller, bounded properties.
- Using induction, some unbounded properties can be proven completely, but when assumptions are used this creates some delicate semantics issues.

As a result of exploring these issues, the “finite-window” subset of PSL was defined that is adapted to the capabilities of bounded checkers. For this subset, it was shown that an induction-based proof is safe. Further, the subset was demonstrated to be practical, as the Tricore properties were easily written in this subset.

R&D Value

The R&D contribution is twofold:

- The properties served as a useful benchmark to tune OneSpin's static checking engine.
- The case study stimulated theoretical work on the capabilities and limitations of bounded checking, and led to defining an appropriate “finite-window” subset.

Productivity enhancements

For this design, no effort figures from a previous approach are available.

The productivity enhancements in writing the specification are reported in the specification case study report.

During the PROSYD project, the overall run-time of this test bench was reduced from 35 minutes to 12 minutes, i.e. by a factor of 3 – this is mainly due to general improvements in the proof engines.

2.3 SLIM Core Processor

The SLIM Core is a processor that has already been implemented in silicon in STMicroelectronics products. It was chosen as a case study for collaboration between STMicroelectronics and OneSpin because the design was well understood, and the work could focus on the details of the specification, and the use of the tools.

All results of interest for this design were reported for the specification case studies (D1.4/1 [15]). As part of WP3, the improved OneSpin static checker was applied, and the report D1.4/1 [15] explains how this tool was used as a target for the specification, and discusses the relative merits of different specification styles for different tool.

Thus we do not repeat these results here.

3 ST UK's Case Studies

3.1 Protocol Converter

Design Characteristics

The STBus Generic Protocol Converter is a highly configurable design that can be interposed between different variants of the interface of a proprietary system bus in STMicroelectronics.

It performs conversion between different widths of bus interfaces; different clock frequencies; and different features of the protocol such as support for out-of-order responses to transactions.

The generic converter was a case study for both specification and verification. The specification aspects are reported in D1.4/1 [15].

From the point of view of verification, interesting features of the design are

- The design can be instantiated with different values of a number of generic parameters. In particular, the size of the design can be varied. This meant that the same property could be checked for different sizes of the design (for example, different depths of fifos), and the performance of different engines could be compared as the size of the design increased.
- The design contains fifos for the request and response packets passing through it. Experience has shown that fifos often lead to computational complexity for static property checking tools, and their presence provides a good challenge for the verification engines.

The RTL model of the generic converter has 11K lines of code. The instances of the design used for the verification case study were of sizes ranging from 488 to 847 flip-flops.

Goals

- To demonstrate the effectiveness of static property checking in providing complete verification of a critical component in an industrial interconnect system.
- To compare the performance of different static checking engines.
- To assess the increase in capacity of the improved engines in the IBM static property checking tool, D3.2/7 [5].

Tools Used

All the verification was done with the IBM static property checking tool, D 3.2/7 [5].

The research reports on combined static and dynamic verification, D3.1/1 [1], and on leverage of different static checking engines, D3.1/2 [2], were used as references.

Work Done

The property-based verification of the generic converter started after the RTL design had been developed, and had been tested in simulation for a range of configurations. The reason for requiring an additional property-based verification environment was to enable verification to be re-run quickly for any configuration, when it was needed in any new system project.

The research report on combined static and dynamic verification, D3.1/1 [1], gives rules of thumb for choosing which blocks to check statically in a verification project (section 2.1). Blocks that make good candidates for static checking typically have lots of control and little data; are hard to cover through simulation; and are buried inside the design. The generic converter has two of these three features: it is predominantly a control block, and any instance of it appears inside the system interconnect, far from the top-level interfaces.

The recommendations in section 2.1 of D3.1/1 [1] are made in the context of the one-off execution of a large verification project, with a choice between dynamic and static verification being made for each block. The reason for applying both static and dynamic verification to the generic converter is that it is an IP that will be re-used in many large projects. The first pass of the verification is dynamic, and this is followed by a comprehensive property-based specification that can be applied to the verification of different instances.

The property specification of the generic converter was developed as described in the report on case studies in specification, D1.4/1 [15]. A number of configurations of the design were verified by running checks of these properties with the IBM static property checking tool RuleBase PE, D3.2/7 [5]. The first configuration chosen was a very small one, with fifo depths 1 or 2, for the purpose of debugging the properties. One design bug was found for this configuration, for a parameter value that was possible in principle, but smaller than expected in practice.

The next configuration was one that is instantiated in an actual project. Checks of all the properties were run in RuleBase for this configuration. The properties checked in the tool were exactly those as written in the specification. No attempt was made to simplify the verification task by, for example, restricting the ranges of some variables, or breaking up single properties into several simpler ones.

Three proof engines were used:

- Discovery, an engine based on BDD algorithms, which was available in RuleBase before the improvements added under PROSYD
- SAT, an engine based on SAT algorithms, which was available before PROSYD but has been improved under PROSYD
- Interpolator, an engine based on SAT algorithms, which has been newly introduced under PROSYD

The research report on effective leverage of different static checking engines, D3.1/2 [2], was written before Interpolator was available, and recommends using

Discovery for conclusive proof and SAT for finding bugs. SAT, being a bounded model checking algorithm, cannot by itself provide conclusive proofs of properties describing unbounded behaviour. Although Interpolator is based on SAT algorithms, it has additional features enabling it to provide conclusive proofs. All three algorithms can give proofs that a property holds up to a certain depth (e.g. for the first 50 cycles of any possible behaviour of the design), even when the computational complexity of the checks is so great that the available time and memory resources are exhausted before a conclusive proof can be found.

It was soon found that the old Discovery engine was not adequate for verification of the chosen configuration, as it typically reached depths of not more than 15 cycles within 12 hours, and if there were any bugs, it was not expected that they would all show up within 15 cycles. It was also found that the new SAT engine could prove that properties held up to depths of typically 100 cycles or more, and this was sufficient to be completely confident in the design, as there were no large counters or long set-up sequences that could mean that, if there were a bug, it would require more than 100 cycles to show it up. For this design, passes for some bounded number of cycles were considered to count as verification of a property, and the SAT engine was used as a tool for verification as well as falsification of properties. Details of the results of comparison of the engines can be found in the "Results" section below.

For this particular configuration, all except 3 of the 63 properties were either conclusively proved, or proved to a depth considered sufficient for complete confidence. The remaining properties were proved to depths of around 25 cycles, in runs lasting from one to two days. This was considered sufficient to give confidence comparable to that provided by simulation testing.

In addition to this application to a configuration instantiated in a product, further experiments were carried out on configurations obtained from this one by varying some of the parameters. The purpose was to see how large the design could be, while remaining within the capacity of the new improved static checking engines.

Results

One bug was found in a configuration with small values of the parameters.

The configuration that was instantiated in a product had already been verified in simulation, and no further bugs were found in it. For this configuration, static checking of the 63 properties took 2 weeks, on top of the 4 weeks already spent on specification of the properties. All properties were checked for at least enough cycles to give acceptable confidence that they cannot be violated.

A detailed comparison of three engines was carried out for 15 of the properties, including those hardest to prove. All properties were run for a maximum of 12 hours and 2GB of memory. To summarize the results:

Discovery: One very simple property passed in 12 seconds. None of the other 14 properties was conclusively proved: the number of cycles for which it was proved that properties could not be violated ranged between 10 and 18.

SAT: Three properties could be proved up to depths of only 15 cycles. The remaining 12 properties could be proved for depths ranging from 55 to 3045 cycles.

Interpolator: Three properties could be proved only for depths of between 15 and 18 cycles (the same properties as could only be proved for 15 cycles with SAT). The remaining 12 properties could all be conclusively proved within the resource limits set, in times ranging from 6 seconds to 4 hours.

It can be concluded that the new engines, Interpolator and improved SAT, are essential for verification of this design. Further, that each of Interpolator and improved SAT is by itself sufficient for satisfactory verification of the design.

The generic protocol converter was also used for an experiment to assess how far the new algorithms in the IBM static checking tool, in particular the Interpolator engine, increased the size of design that could be statically verified.

In general, there is only a loose connection between the size of a design and the resources (time and memory) required for its static verification: there is also a big dependency on the type of design and the property are also One property on a design with 100 state variables may be beyond the capacity of the tools, while another property on a different design with 1000 state variables can be proved in a few minutes. However, for the parameterized generic converter, it was possible to vary the parameters but to keep the type of design constant, and to check the same properties for instances that differed only in their size.

The Interpolator and Discovery engines were used to check ten properties for three configurations of the design, one with 488 flip-flops, one with 671 and one with 847. Limits of 12 hours CPU time and 2GB of memory were set. The results were as follows:

	# properties completing with Discovery	# properties completing with Interpolator
488 flip-flops	1/10	10/10
671 flip-flops	1/10	10/10
847 flip-flops	1/10	7/10

Table 3-1: RBPE Engine capacity comparison.

A conservative conclusion is that the configuration with 488 flip-flops was beyond the capacity of Discovery, while the configuration with 671 flip-flops is within the capacity of Interpolator. (In fact, more detailed examination of the depths to which properties are checked indicates that the configuration with 847 flip-flops is close to being within the capacity of Interpolator.) It is therefore safe to say that, for this design, the new engines have increased the capacity from less than 500 to more than 700 flip-flops, that is, by at least 40%.

R&D Value

The novel feature of the way properties are used in the design flow is that property-based verification is used for regression and for variants of the generic design, even though a conventional testbench was used for the initial verification.

Productivity Enhancements

The total time for specification and verification of one configuration of the generic converter was 5 weeks (3 weeks specification plus 2 weeks verification). For the testbench that had previously been developed, the total time was also 5 weeks (3 weeks specification plus 2 weeks verification).

An experiment that varied the design size by changing some parameter values showed that the new engines increased the size of design that could be verified by at least 40% (from 500 to 700 flip-flops).

The property-based approach therefore required very similar effort to a testbench for a single instance of the generic converter. The objective in this application was

not so much to reduce verification time for a single instance, as to develop a property-based specification that can easily be reused for different configurations.

The improved engines were essential in making property-based verification of this design possible.

3.2 Transport Front End

The Transport Front End is a substantial IP consisting of several independently developed components.

The IP receives input streams from several sources, and has to extract valid packets from these streams and place them in buffers in system memory. There are several stages of processing of the packets, in a mainly linear flow. Temporary storage in RAM inside the IP is used to buffer the streams after they have been merged, and a DMA engine extracts the packets from this RAM to transmit them to main memory on the system bus.

There are five main components. Three of these were developed specifically for this IP (two components for processing the contents of the input streams, and the DMA engine), and two general-purpose components (an arbiter and a memory interface) were reused from other designs, with some modification.

Property-based specification and verification were both applied to this design. In this document we report on the process and results of the verification activity. An account of the specification activity, and more information about the specification and organization of the design, can be found in the report on case studies in property-based specification, D1.4/1 [15].

Design Characteristics

The whole design has 7000 flip-flops and 15K lines of code in an RTL model.

From the verification point of view, the technical features of interest are

- The design has all the features of a typical moderate-sized IP (several interfaces, interaction with a host CPU, multiple clocks, etc.). Success in complete property-based verification of this IP will give confidence that the approach can be successful for many other IPs.
- The DMA component in particular is especially complex, in terms of its interaction with other components, use of internal resources to store both control information and data at different times, behaviour when errors are detected, etc. It is necessary to devise strategies for how to use the tools to prove the properties.

Goals

The goals of this case study were

- To verify a typical industrial design of moderate size, using only static property checking.
- To assess the performance and capacity of different engines in the IBM static property checking tool.

Tools Used

The IBM static property checking tool, D3.2/7 [5], was used for the verification. The research report D3.1/2 [2] on effective leverage of different static checking engines was used as a reference in choosing options for running the tool.

Work Done

The components of the Transport Front End were verified separately. Reasons for this were that verification of properties on the whole IP would be too complex for the tools, and that verification of the components could be done much earlier than verification of the IP as a whole.

For the components that were developed from scratch for this IP, the verification was carried out in parallel with the implementation of each component. For the two imported components, verification was done after they had been adapted for the needs of the Transport Front End.

The fact that property-based verification was in progress while the RTL implementation was being written allowed the designer to spend less time on simulating his development version, and reduced the design effort, at the expense of some increase in verification effort, as the number of simple bugs found in verification was greater than usual. One unexpected point was the distribution of the detection of different sorts of bugs during the verification process. Some complex bugs involving special cases were found at an early stage; this was a bonus, as tests that would reveal such bugs are often not available until a late stage in conventional verification processes. On the other hand, some simple bugs which would prevent the IP from functioning at all were not discovered until a late stage, simply because the relevant properties were among the last to be checked. In order to reduce problems arising from design models having showstopper bugs, additional properties were written stating that specific packets do get through from one end of the design to the other, and these were checked using the SAT engine, which can be expected to find simple bugs in a short time, even on a large design, while the main verification was done using the Discovery engine, which is aiming to prove properties conclusively on the subcomponents of the design.

The way the set of properties was developed is described in the report on case studies in property-based specification, D1.4/1 [15]. The properties were each checked on the current version of the implementation as soon as they had been written; this contributed to the debugging of the properties, as well as the early detection of design bugs. For many of the properties, direct checking of them exactly as written was too complex for the tool, and a number of tactics were used to reduce the complexity while maintaining confidence in the coverage of the verification. These included

- Checking properties only for a representative set of values of configurable parameters such as packet length.
- Checking only a single bit of data values, where it was known that the design operated on each bit in the same way.
- Explicitly allowing the certain registers to take any values at any time, where it was believed that the actual values in these registers did not affect the truth of the property. This is a “safe” tactic to adopt, since if a property passes when the registers can take any values, it will certainly pass when they only take the values that occur in an actual behaviour of the design.

Such tactics were necessary for the DMA engine in particular, and considerable ingenuity was applied in explicitly making a simplified model of the behaviour of

two wide internal buffers that were used to store data at some times and control information at others. The other components required only straightforward use of tactics for reducing complexity. Indeed, for the imported memory interface, the properties were run in the tool exactly as specified, with no need for further human effort in completing the verification.

The Discovery engine was used for the majority of the proofs. According to the report on effective leverage of static engines, D3.1/2 [2], the engines that can provide conclusive proofs, as were required for this application, are Discovery, Classical, and SmartLoc. It was found from experience on a few properties that Discovery gave the quickest results for this design, and it was used as the engine of first choice. The SAT engine is recommended in D3.1/2 [2] as suitable for finding bugs quickly on larger components, and it was used in this way for sanity checks on packets passing through the design, as described above.

Experiments were also done to assess whether use of the new Interpolator engine would speed up the checks, or make it possible to check properties on larger components than could be done with Discovery. The results are reported below: in fact, Interpolator did not improve on Discovery for this design.

For one block (PID Filter), that performs filtering on the packet identifiers for packets in each of six input streams, and arbitrates for access to look-up tables in main memory, a small number of worst-case performance properties were specified. It had already been possible to establish, by consideration of the microarchitecture, that the average-case requirements were comfortably satisfied; the worst-case performance was not strictly a specification requirement of the design, but rather a feature that it was interesting to know. In checking these properties, the actual values (greatest possible latency, etc.) were not known in advance, but the properties were run with different values until they passed. For the groups concerned, this was a novel way of using "verification" to provide definite worst-case performance measurements.

Results

In all, 969 PSL properties were checked on the Transport Front End. For the final release of the design, all passed except two. These were recognized and documented as acceptable failures, as it was known that, for the context the design was to be used in, the bugs would not cause any problems.

The verification found 90 implementation bugs. These ranged from simple typos and unconnected signals, to incorrect wrapping at the top of buffers, to corner case errors in state machines. Approximately 25 of the bugs could be deemed significant. The number of bugs was larger than normal for a design of this size, even though the designers were experienced, because the verification had started when the design was very immature. In addition to the implementation fixes, there were half a dozen places where the specification was changed following a property failure that indicated a mismatch with the implementation, and one place where a condition was added to the specification, stipulating a need to flush buffers before performing certain operations, that was needed in order to avoid a property failure.

The properties covered the whole functionality of the IP once it was set up and running in normal mode. Property checking was used for the whole verification of this functionality, not just as a supplement to other verification. Things such as test mode, and enabling and disabling streams, were tested separately in simulation without reference to PSL properties. It would have been possible to cover these aspects with properties, but this would have involved opening a new area of specification, and since the features could easily be tested in simulation, these tests were considered sufficient.

When the IP was subsequently integrated into a larger system and simulated, it emerged that two bugs had been missed owing to errors in two of the properties; in both cases, the PSL specification had wrongly interpreted the details of the informal functional specification, in exactly the same way as the RTL implementation had done. There is no way of completely eliminating this sort of error, although more time devoted to review can reduce the risk.

The complete PSL specification was carried out over a period of 9 months, and took 4 person months of effort.

Two experiments were done to compare the performance and capacity of the Discovery and Interpolator engines. The first was carried out on the PID filter. A sample of 14 properties was taken, including two simple protocol properties, a major property about the correctness of the data value returned, and a performance property. The last two properties could each be applied to any of the six input streams, and the performance property in particular involved the interaction of requests from all the input streams. All properties passed on the final version of the design.

The results were:

Properties	Times to pass with Discovery	Times to pass with Interpolator
Protocol (2 properties)	3 – 27 sec	140 – 650 sec
Data (6 properties)	4 – 50 sec	595 – 16146 sec
Performance (6 properties)	56 – 70209 sec	All out of memory

Table 3-2: Engine comparison for PID filter.

For this design, Discovery was uniformly better.

The second experiment was done on a large component (Input Block) that performs the initial processing of an input stream. There are four stages of this processing, and properties were specified and verified for the sub-blocks corresponding to each of these stages, because of the computational complexity of static verification at the level of the whole Input Block. If it were possible to prove properties efficiently for the whole Input Block, this would greatly reduce the effort needed for both specification and verification of the block.

One property about a packet passing through the whole Input Block was checked using the Discovery, Interpolator and SAT engines. SAT found a failure after 280 cycles, in 14 hours of run time, by which time Discovery had checked the property up to a depth of 175 cycles, and Interpolator up to a depth of 42 cycles. In this case, the error was in the property rather than the design. When the correct version of the property was checked, neither Interpolator nor Discovery was able to prove it before running out of memory. So, although the benefits of the improved SAT engine for finding bugs were demonstrated, the new engines could not be used to raise the level at which the Input Block was verified. These results simply indicate that the particular design was not well suited to Interpolator, and there is no

improvement in this case. For the Transport Front End, Interpolator brought no benefit, but nor did it bring extra cost.

R&D Value

This was the first time for the groups concerned that the whole verification of a significant IP had relied on a property-based approach.

Productivity Enhancements

The total effort needed for verification was 4 person months, in addition to 8 months spent on specification. As a comparison with traditional specification methods, the effort to develop a reference model and a verification environment for a previous design of similar size and complexity, performing a similar function, was 7 months, and the verification effort following this was 3 months. It should be noted that the verification effort depends heavily on the number of bugs found, and this was large for the property-based verification: the time spent on verification did in fact reduce the time spent on design. It is also hard to make a clear separation between the efforts on specification and verification; the total efforts of 12 and 10 months for the two approaches are comparable.

The 10 months' effort was for a testbench at the level of the whole IP. Since the components were specified and verified separately in the property-based approach, a closer comparison would use efforts for testbench verification of each component. There are no such figures directly available: in order to compare efforts for these, an expert in testbench development was shown the informal functional specifications, and asked how long it would take a competent engineer to develop a testbench for each component and run tests. The estimates for specification have already been reported in D1.4/1 [15]. Adding the verification efforts to that table we have

Component	Number of properties, flip-flops, LOC	Person-weeks' effort for PSL spec + verify	Estimated effort for testbench spec+ verif	Productivity Gain
Input block	240, 570, 3000	13 + 4	16 + 2	6%
DMA	270, 1200, 2500	15 + 8	12 + 2	-64%
Arbiter	227, 110, 4000	2 + 2	3 + 3	33%
PID filter	59, 290, 450	2 + 2	3 + 2	20%
SRAM interface	52, 500, 1500	1 + 0	2 + 2	75%

Table 3-3: Effort on components of Transport Front

These results show how the effort on the DMA verification was significantly higher than for other blocks, and constitutes a large proportion of the overall verification effort. The results indicate that, for the larger blocks, the property-

based approach did not reduce the effort compared with a testbench approach. It should, though, be noted that the testbench approach is better established, and there is more scope for reducing effort in the property-based approach as a result of experience.

The arbiter, PID filter and SRAM interface are smaller, though still of significant complexity, and the property-based approach has clear advantages here. The way to specify properties is well understood, and the properties can be verified as they stand without too much effort from the user. For the PID filter, the verification time includes that for the worst-case performance properties, which it would be impossible to check in a testbench.

4 ST F's Case Studies

4.1 Bus Protocol

Design Characteristics

The object of this verification case study is the sole ST proprietary bus protocol also dealt within a specification case study which also comprises AXI the latest generation of the AMBA protocols [18]. The ST proprietary protocol is a flexible and structured communication framework which comprises three incremental variants with differing performance and complexity cost (named type 1, 2 and 3). For more details about the ST proprietary bus protocol, refer to section 2.7 of the report on case studies in property-based specification D1.4/1 [15].

Goals

The main goal of this case study was the port of the existing PSL properties covering the protocol from GDL to Verilog flavor for both formal verification and simulation. Another goal was to study the effects on the formal verification of the property specification improvements presented in section 2.7 of the report on case studies in property-based specification, D1.4/1 [15].

Tools Used

This case study was mainly conducted by referring to methodology document on managing the trade-offs between robustness and coverage in a combined static and dynamic verification, D3.1/1 [1].

Work Done

The PSL protocol properties code was ported from GDL to Verilog flavor and packaged according to a different infrastructure following section 2.7 of the report on case studies in property-based specification, D1.4/1 [15]. The original infrastructure was based on the vunit / vprop / vmode PSL containers and each property was the object of a vunit. The alternative infrastructure is based on read only monitor Verilog modules with embedded PSL code. All the safety properties are gathered in a single read only monitor Verilog module while each liveness property has a dedicated read only monitor Verilog module.

All the read only monitor Verilog modules taking all together liveness and safety properties can be enabled in simulation. On the other hand, one read only monitor Verilog module is only used at a given time in formal verification both to avoid collision between assumptions and fairnesses when any and to minimize the property checks complexity.

Most of the properties (actually all type 1 and type 2 as well as 87% of type 3 properties) have the same code reusable in formal verification and simulation. 17% of the type 3 properties require two different code versions. The hardware designs with a type 3 bus interface actually use control signals that vehicle information such as the source communicating entity label or the transaction identifier. In order to take into account, for the properties that involve these signals, all the possible values for these signals, we need to keep track in an appropriate array data structure of what is needed. This results in a modeling layer code overhead that may be too large to be handled by formal verification. Under the protocol viewpoint the concerned designs, do not depend on specific values taken by these signals and handle each value the same way. Because of this design specificity, the formal verification code version focuses on distinguished values for these signals in order to minimize the modeling layer code overhead and render the corresponding property checks tractable by static engines. On the other hand, the simulation code version is extensive and covers the complete value range of these signals for the purpose of functional coverage.

The refinement of the property specification with the sub-profiles taken into consideration was expected to lead to ad-hoc specialized sets of properties also easier to be formally verified. In order to assess the impact on verification of this refinement, we launched a protocol compliance static checking campaign on the hardware components for which it was relevant, using both the original property set and the sub-profile sets.

Results

The porting work done on this case study has mainly resulted in changes of the use model of the ST bus PSL properties within our flow. We now dispose for the ST bus of two property infrastructures in question coexist in our flow for the time being. The static protocol property checking, which is performed on each hardware component with an ST bus interface, is mainly done using the original infrastructure while the alternative is mainly used for simulation. The deliverable D3.1/1 [1] methodology document on managing the trade-offs between robustness and coverage in a combined static and dynamic verification environment, lists the various approaches and discussed their modalities. The retained reasons to define our approach for the protocol case are as follows. First**,** the protocol properties are double checked by formal verification and simulation, to catch by simulation the issues that might have escaped the formal verification for complexity reasons (proofs achieved under strong assumptions or restrictions, bounded checks). Another good reason for this reuse is the functional coverage feed-back supplied the PSL properties for free when using the native PSL support of the simulator. This feed back is particularly welcome when the environment simulation does not have proper means to collect this information. Such is the case for the testbenches build around SystemC models. The PSL properties permit in that case to ensure as much as simulation permits it, that the behaviour of the SystemC is correct w.r.t. the protocol.

As for the impact on verification of the specification improvements, only one design with a ST bus interface (actually type 3 target) was checked so far using the original set of property and the one corresponding to the sub-profile. The size of the design in question was 517 flip-flops, 10854 gates and 158 inputs. Using Rulebase on Linux bi-processor machines with 6GB RAM and 32 bits 3GHz CPU, it took 25mn35s to prove the procotol compliance of the design with the original property set involving 26 properties. With the sub-profile property set, the number of considered properties was reduced to 19 and resulted in a verification time of 19mn40s.

R&D Value

This case study addresses the use of both the static and dynamic techniques of PSL property based hardware protocol checking, including the interoperability aspects between.

Productivity Enhancements

The contributions of this case study are mainly of qualitative nature. The sole item which can be quantified is the impact on the formal verification efficiency of the property specification improvement. The design for which it was possible to check the ST bus compliance using both the original set of property and the sub-profile one, pointed a reduction of the total formal verification runtime of 23,3%. This observed gain which was expected, should be confirmed on other designs given the purpose of the property specification refinement into sub-profiles. The purpose was twofold, taking into account specific implementation choices not captured by the generality of the original set of property and simplifying the corresponding property checks.

4.2 Memory Interface

Design Characteristics

This case study is a joint specification and verification one. A brief description of it will follow. For more details, refer to section 2.8 of the report on case studies in property-based specification, D1.4/1 [15].

The IP carries out the data transfers between the ST proprietary bus and any of these external memory cards e.g. nand flash, compact flash, smart media card. It is organized around a 32 bits RISC CPU to support many operating modes. All the hardware components (the parallel interface, the buffer, the two bus interfaces) for which a property specification was developed as reported in section 2.8 of the report on case studies in property-based specification, D1.4/1 [15] were the object of the verification reported hereafter.

The whole IP distribution contains 50880 lines of code if we exclude all embedded memories and register files, which represents 7319 flip-fops and 231547 gates according to RuleBase Parallel Edition. The top level module has 276 inputs among which 2 clocks, one governing the sole bus interface type 1 and the other clock cadencing all the rest of the IP.

Goals

The main goal of this case study was to assess the major PROSYD contributions in terms of verification (both on methodological and tools sides) on the hardware components of the IP that were freshly designed or modified.

Tools Used

The methodology document on managing the trade-offs between robustness and coverage in a combined static and dynamic verification, D3.1/1 [1] and the methodology document on effective leverage of various static checking engines, D3.1/2 [2] were used to define the PSL property verification approach.

The verification was carried out using:

- a version of RuleBase Parallel Edition aligned with the manual for improved static property checking tool, D3.2/7 [5] (based on algorithmic framework reported in D3.2/1 [3], D3.2/2 [4]).
- a version of NuSMV aligned with the manual for improved static property checking tool, D3.2/9 [14] (based on algorithmic framework reported in D3.2/1 [3], D3.2/2 [4]).
- a version of FoCs aligned with the manual for property-based automatic generation of simulation monitors for digital designs and D3.2/11 [9] (based on algorithmic framework reported in D3.2/5 [8]) and with the report on the support for embedded PSL in verilog flavour – Observers, D3.2/16 [19].
- a version of RuleBase Parallel Edition aligned with the report on the support for embedded PSL in Verilog Flavor - Static Checking D3.2/15 [17].

Work Done

For each of the four hardware components of the memory interface (the parallel interface, the buffer, the two bus interfaces), the RTL implementation was statically checked against the PSL property specification. RuleBase Parallel Edition and precisely the version implementing the content of the manual for improved static property checking tool, D3.2/7 [5] (based on algorithmic framework reported in D3.2/1 [3], D3.2/2 [4]), was extensively used. The enhanced version of NuSMV corresponding to the manual for improved static property checking tool, D3.2/9 [14] (based on algorithmic framework reported in D3.2/1 [3] and D3.2/2 [4]), was also envisaged on the sole bus interface type 3 block. As we had to assess the engines enhancements brought by PROSYD, we systematically considered for each PSL property of the case study, the four major engines of RuleBase Discovery, Interpolator, SmartLoc and SAT each given the same upper bound time limit. Note that the enhancements described by the methodology document on effective leverage of various static checking engines, D3.1/2 [2] concerned Interpolator, SmartLoc and SAT. In normal use conditions, the property checking strategy using RuleBase consists to take as much benefit as possible from the multi-engine feature, which permits to run several engines in parallel on a network of processors with the quickest job killing the other ones. The underlying concepts and motivations of this feature are detailed in the methodology document on effective leverage of various static checking engines, D3.1/2 [2]. This feature use confirms that it constitute the best approach to overcome to the difficulty to completely characterize the complexity of a property check (the size consideration is only one element of that characterization) in order to determine which engine is the most appropriate.

Moreover, we studied the impact of the property implementation choices on the static checking efficiency and we focused on the pass through properties and in particular on the high level model based framework already mentioned in section 2.8 of of the report on case studies in property-based specification, D1.4/1 [15]. In this framework, the property statement comes together with a score board coded using the modeling layer. The score board was first implemented through a two dimension memory array memory keeping a copy of each data put on the input side and used it as a basis for a comparison at output side as shown in section 2.8 of the report on case studies in property-based specification, D1.4/1 [15]. All the objects that traverse the hardware component are systematically stored in this rough score board and the property states that the transmission though the component is correct

for all the objects. Another score boarding approach can consist to focus on a representative object, only keep track of what is needed for it and prove that the transmission of this particular object through the component is correct but let the object in question to be arbitrary. The code of this implementation choice is also presented hereafter in the case of the buffer hardware component for its SW load mode.

```
vprop partial_scoreboad_data_transfer_SW_load_mode {


reg    any_occurrence = 0;

always @(posedge clk)
  begin
    if (!rst_n)
      begin
        any_occurrence <= 8'd0;
      end
    else
      begin
        if (!any_occurence)
          any_occurence <= nondet(0,1);
      end
  end

reg[7:0]    objects_ahead  =0;

always @(posedge clk or negedge rst_n)
  begin
    if (!rst_n)
      objects_ahead<= 8'd0;
    else begin
      if (any_occurence) begin
        if in && !our && (objects_ahead < 12)
          objects_ahead <= objects_ahead + 8'd1;
        else if (out && !in && (objects_ahead > 0)
          objects_ahead <= objects_ahead - 8'd1;
      end
      else if (out && (objects_ahead > 0) :
        objects_ahead <= objects_ahead - 8'd1;
    end
  end


assert forall d[15..0] boolean :

  always (
    (in
     && data_in == d
     && rose(any_occurence))
    -> next_event
       (out && (objects_ahead == 0))
       (next[2](data_out == d))));

}
```

We made measurements of the static property checks corresponding to these two score boarding implementations using RuleBase. In addition, we considered for each of this implementation choice a coding variant of which we measured the property checks. This coding variant consisted in converting all the sequential behaviour of the modeling layer into a large set of assume statements.

As already mentioned in the section 2.8 of the report on case studies in property-based specification, D1.4/1 [15]**,** the purpose of the memory interface development project, of which this PROSYD work constituted a part, consisted to enhance the IP to different communication and operating requirements. The project started not only with an existing RTL distribution but also with some existing verification materials actually a Specman™ testbench This particular context led us to keep this testbench and reuse it after the needed adaptations according to a functional verification strategy combining simulation and formal verification as follows :

- simulation at top level using the Specman™ testbench
- PSL property static checking at entity level on each hardware component freshly coded from scratch or re-designed

The PSL properties developed for the memory interface played a central role in this strategy.

In addition to being statically checked as indicated above, the properties were simulated within the Specman™ environment using the Cadence Incisive Unified Simulator and its native PSL support capabilities.

The methodology document on managing the trade-offs between robustness and coverage in a combined static and dynamic verification, D3.1/1 [1] lists the various approaches and discussed their modalities. Within the context of this case study, the followed approach, consisted to first carry out the property static checking and then handle their simulation focusing on a special property subset, the assumptions. Simulation constitutes a concrete automatic means to validate the assumptions which supplements the code review process to which all the properties are submitted in our PSL property use model.

In addition, a particular attention was paid to the simulation of the ST proprietary protocol properties. The idea was to use the PSL code of the properties, to collect the coverage information w.r.t. the simulated ST proprietary bus traffic. Note that in our simulation environment this information was available supplied through Specman™ by dedicated data structures expressed in e language.

An alternative property simulation path to the native PSL support of the simulator was also envisaged using FoCs. The considered FoCs version was aligned with the manual for property-based automatic generation of simulation monitors for digital designs and D3.2/11 [9] (based on algorithmic framework reported in D3.2/5 [8]) and with the report on the support for embedded PSL in Verilog flavour – Observers, D3.2/16 [19]. This was only applied to the ST bus proprietary properties. We considered both formats in which these properties were available:

- vunits
- embedded in Verilog read-only monitors following the framework described in section 2.7 of D1.4/1 [15].

Under practical viewpoint and in a first time, all the developed properties were put in separate files using the PSL vprop, vmode, vunit and checked that way. When the version of RuleBase supporting the embedded PSL in Verilog flavor became available, the code of distinguished properties was transferred from the vunits to the Verilog RTL source files. The concerned properties were local implementation

ones. As they strongly depend on the RTL internal details implementation, the embedded approach makes more sense for them. This was done for the sole buffer hardware of which RTL is pure Verilog. Hence the PSL Verilog flavour feature of Rulebase was only used on these local implementation properties.

Results

In addition the nine defects found by the property specification process, a total of eleven bugs were uncovered at the verification stage by the property static checking:

- Ten involved the bus interface and the buffer blocks. Eight were revealed as violations of the ST proprietary bus protocol. The root cause of them was the misinterpretation of the ST proprietary bus protocol specification as the designer was not yet familiar with the protocol and was exposed for first time to related design. The two other bugs also resulted in the register property violations but as the effect of a bad apprehension of two control conditions.
- One bug involved the parallel interface hardware component. It was found by the data transfer property ruling the non vectorial path of the input mode. The bug resulted in broken data integrity for certain values of the parameters ruling the pulse duration of the control signals within the interface the GPIO.

In terms of use of the multi-engine feature of RuleBase let us distinguish between the ability to falsify and the one to prove.

Taking all the hardware components together 60% of the failures were found by Discovery and 40% of them found by the enhanced version of SAT.

As for the proof power, the overall results are uniform for both the bus interfaces and the buffer with 100% of the properties all proved by Discovery. The distribution is more balanced between the engines for the parallel interface hardware component with 53% of the properties proved by Discovery, 25% also proved by Interpolator only and the rest checked by the enhanced version SAT up to a given explored depth. The following table gives the measures for a representative sample of the property checks performed on the parallel interface. The overall size of this hardware component is 469 flip-flops, 17025 gates and 170 inputs. P1 is a register property, P2 is a local implementation about the internal FIFO, P3 is a data transfer property for the input mode and P4 a data transfer property for the output mode.

All the measures reported in the sequel, were made on Linux bi-processor machines with 6GB RAM and 32 bits 3GHz CPU. In the second sub column per engine which gives the time in hh:mn:ss, “to” stands for “timed out” i.e. non completion after 12h00 of run times. “oom” stands for “out of memory”. In the first sub column, P stands for the property is completely proved otherwise the number of explored clock cycles is supplied.

P#. # flip-flops / # env variables / # satellites / # inputs / # gates	Discovery		Smart Loc		Interpolator		SAT	
P1. 34/85/3/48/1473	-	To	-	to	P	0:01:34	-	to
P2 124/60/6/68/5482	2509	To	-	to	P	3:01:57	460	to
P3. 401/221/0/149/18141	P	7:58:13	-	to	31	oom	40	to
P4. 500/226/0/150/21099	P	10:16:67	-	to	37	oom	35	to

Table 4-1: Rulebase multiengine use measure sample.

All our measures beyond Table 4-1 figures, show how relevant is the multi-engine feature of RuleBase. The falsification time is reduced thanks to the supplementarity provided of the enhanced SAT w.r.t. Discovery. On the proving side, the supplementarity of Interpolator w.r.t. Discovery does not only reduce the run time but also increase the verification scope in the sense that properties that were not tractable by Discovery are solvable by Interpolator.

The use of NuSMV was undertaken on the type 3 interface block after the completion of its RuleBase campaign. One can regret the lack of an integrated front-end for VHDL and Verilog or a well defined compilation path, which is harmful for the flexible use of the NuSMV.

We first reused a custom Verilog compilation path based on Synopsys Design Compiler and Perl scripts to convert the RTL into a SMV model required by NuSMV. Given the difficulties we faced to get it absorbed, we decided to rely on the SMV model built by RuleBase. The good side of the coin was that a lightweight Perl script was needed to render the RuleBase SMV model tractable by NuSMV also paving the route of plugging NuSMV as an additional external set of engines to RuleBase. The other side of the coin was that this prevented us from assessing the capabilities of NuSMV of processing the original models as the RuleBase SMV model is already consolidated especially the cone of influence of reduction.

Table 4-2 gives the measures for a representative sample of the property checks performed on the type 3 interface block. The overall size of this block is 517 flip-flops, 10854 gates and 158 inputs. C1, C2, C3, C4 were all ST proprietary bus protocol properties checked by Discovery in at most few minutes for the largest one.

C#. # flip-flops / # env variables /# satellites / # inputs / # gates	bdd		sbmc – MiniSAT	
C1. 342/207/3/213/11102	-	oom	100	1122s5
C2. 52/192/0/205/2988	-	oom	100	979s5
C3 35/159/3/173/1992	P	8s34	100	125s8
C4 20/140/0/145/1380	P	2s43	100	8s4

Table 4-2: NuSMV use measure sample.

If the bounded model checking mode showed an interesting behaviour as illustrated by the fifth column figures above, the picture remains incomplete for the

unbounded model checking mode as reflected by the second column figures for C1 and C2. In fact, NuSMV was unable to digest the SMV models starting from a certain size. This compromised our assessment of the tool performance and capacity to prove properties.

Table 4-3 presents the measures for a representative sample of the property checks related to the various property organizations envisaged for the pass-through behaviour as well as the differing implementation score boarding choices and coding styles. The table content refers to the buffer of which overall size is 929 ff, 34117 gates, and 102 inputs including two RAM arrays.

S/H#. # flip-flops / # env variables / # satellites / # inputs / # gates	Discovery		Smart Loc		Interpolator		SAT	
S1. 247/67/9/55/28061	P	0:01:56	-	to	13	to	15	to
S1'. 247/295/0/183/30969	-	to	-	to	19	oom	30	to
S2. 247/59/9/39/27925	P	0:01:47	-	to	15	to	15	to
S2'. 247/292/92/9/63/28542	P	0:02:44	-	to	15	oom	15	to
H1. 247/59/9/39/27925	P	3:22:51	-	to	na	na	na	na
H2. 247/115/0/55/28694	P	0:01:30	-	to	16	to	15	to
H3. 247/295/0/183/30969	P	0:01:38	-	to	16	to	15	to

Table 4-3: Data transfer RuleBase property checks measure sample.

S1 is a rough score board based data transfer property and S1' the variant coding of S1 only with declarative statements. S2 is a partial score board based data transfer property and S2' the variant coding of S2 with only declarative statements. H1, H2 and H3 are the high level view property covering the pass through behaviour without implementation consideration as presented in section 2.8 of the report on case studies in property-based specification, D1.4/1 [15]. H1 states that any input data is eventually output, H2 that any output data has been previously input and H3 that order is preserved from input to output.

All our measures show that verification cost of the score boarding approaches is clearly lower than the high level view framework as reflected by the figures for S1, S2, H1, H2, and H3. Therefore the score boarding approaches are to be favored under the verification efficiency viewpoint. In addition, all our measures of which the figures for S'1, S'2, S1, and S2 are an illustration, showing that the pure declarative coding style leads to the less efficient verification. Therefore the more the coding style is imperative, the better verification efficiency will be. As for the two score boarding approaches, it comes from the figures for S1 and S2, that the partial score board is better. Actually, from the other similar measures, we have rather comparable efficiency as if the additional non determinism of the partial score board lead to an overall complexity which is comparable to the one of the extensive array.

The main observation following the use of FoCs when simulating the ST bus proprietary protocol properties within the Specman™ environment is that the simulation overhead caused by 30 monitors generated by FoCs, was in average less than 2%. This was the best result achieved taking into account together all the simulation means of PSL. In addition, our use of FoCs led us to meet several issues

and formulate few enhancement requests. Seven items were logged in IBM tools bug tracking system to that respect.

As for the use of the PSL Verilog flavor capabilities of RuleBase both embedded and in vunits, it was permitted by a tight interaction with IBM. A total of twenty two defects and enhancement requests were logged in IBM tools bug tracking system at our initiative and about 80% of them have been fixed so far.

R&D Value

The added value of this case study is twofold. First, a wide range of the PROSYD verification tools for digital designs were exercised on it, actually all but the ones corresponding to:

- the manual for Improved static property checking tool, based on algorithmic framework reported in D3.2/1 [3], D3.2/2 [4], D3.2/3 [20], D3.2/8 [13]
- the manual for Property-based analysis of simulation traces based on algorithmic framework reported D3.2/14 [7]
- the report on fast simulation of property-based specifications D3.2/17 [16]

On the other hand, the case study brought a thorough exploration of the factors that can have an impact on the verification efficiency. The focus was made on the engines, the property organization, the property implementation choices and the property coding style. The object of this study was a typical and common design behaviour, present in a majority of hardware designs, the pass through one.

Productivity Enhancements

In the case of RuleBase, which is the major tool used for this case study, the impact of the novel engine and the enhanced engines on the verification scope is particularly positive especially if we consider the parallel interface component. About half of the inconclusive properties with Discovery and SmartLoc, were proved thanks to Interpolator. In other words, the PROSYD contributions on the engines increased the proportion of the proved properties by about a half bringing the percentage from 53% up to 78%. In addition, the enhanced version of SAT, achieved a better explored depth for the other half of the inconclusive properties (with Discovery and SmartLoc) which represents a status improvement.

As for the overall productivity consideration, we dispose for each considered hardware component of the measures of the total effort spent in the property based approach together with just dedicated to the specification. An estimation made by the designer is also available for the potential efforts required by a Specman™ based testbench approach. This approach, which was actually not applied at hardware component level of the case study, is the high-quality testing of a design, including corner-case features, giving the closest possible comparison with the quality of a property based approach.

	Size in number of flip-flops	Specman™ based testbench	Property based approach	Productivity Gain
bus interface type 1	113	2 m.wk (of which setup 0.4 m.wk)	1 m.wk (of which spec 0.4 m.wk)	50%
bus interface type 3	517	3 m.wk (of which setup 0.4 m.wk)	2 m.wk (of which spec 0.4 m.wk)	33%
Buffer	929	2 m.wk (of which setup 1 m.wk)	2 m.wk (of which spec 1 m.wk)	0%
parallel interface	469	10 m.wk (of which setup 5 m.wk)	12 m.wk (of which spec 5 m.wk)	-20%

Table 4-4: Total effort (specification effort) spent per hardware component.

The results differ in function of the hardware component size and type.

The best productivity gain (50%) corresponds the smallest component (in terms of number of flip-flops), the bus interface type 1 which is also the simpler one. For the larger components, we can observe that the productivity gain is not proportional to the number of flip-flops. It actually also depends on an additional factor which is difficult to apprehend, the intrinsic complexity of the hardware component. The parallel interface despite its lowest size gave the worst productivity measurement (-20%) because of its internal complexity which led to overrun for the static property checks. A significant amount of time was spent in vain trying to reach a complete proof for 22% of the properties which remained finally just explored up to a bounded depth. If we look at the nature of the considered hardware components, they are pass-through designs combined with control of which density is function of size and complexity. We can make the observation that up to a certain level of control density, the property based approach is completely under control and is the most appropriate functional means.

5 ST I's Case studies

ST I's case study are all conjoint specification and verification case studies; to avoid useless repetitions only brief summaries have been reported in each design characteristics sections, focusing just on verification aspects; for more information on each case study and to relative contexts, please refer to sections 2.9 – 2.11 of D1.4/1 [15], respectively.

5.1 Memory Controller

Design Characteristics

The MultiPort Memory Controller (MPMC) manages memory access requests coming from 7 different AMBA High performance Bus (AHB) masters, directed to a pool of Double Data Rate (DDR) memory banks. The MPMC is a highly configurable block, with 100 user-addressable registers mapping 254 parameters related to 112 different aspects of the functioning of the MPMC.

The MPMC RTL model is composed of 270 Verilog files, for a total of 59655 LOC that translates into 5618 FF.

Although not novel as a kind of component, the MPMC represents a verification challenge for essentially three aspects:

- The number of parameters available at the user's disposal is extremely high.
- The number of Masters in relation with the adopted functioning mode.
- Access on each AHB port can be asynchronous.

Goals

The intent of the present case study has been to compare the usage of Assertion Based Verification exploiting PSL in comparison with more traditional verification approaches. To reach this goal we have verified the correct functioning of the MPMC in connection with a memory banks in a realistic configuration specified by our internal customer; the intent was to complement the verification of the component that came with a set of directed and partially randomized tests, written using a combination of ad-hoc HDL and SystemC-based Testbench Automation layer. This case study was also used to compare Formal verification and dynamic verification on the protocol checking activities.

Tools Used

D1.1/1 [11] and D3.1/1 [1] have been used as driving methodologies whilst identifying the set of properties that were expected to be significant for protocol verification and data integrity tests. A portion of the case study (the DDR interface) was to be addressed with NuSMV (D3.2/9 [14]), and the whole MPMC, without memory banks, has been checked for protocol compliance using RuleBase PE (D3.2/7 [5]), also taking into account D3.1/2 [2] content. Property simulation (3.2/14 [7]) and FoCs (D3.2/11 [9]) have been used to complete the case study work. The PSLText tool (D1.2/6 [12]) has been extensively used during property adaptation phase (cf. section "Work Done" below). Although we originally planned to address the verification of a MPMC portion using OneSpin Improved Static Checking Tool (D3.2/8 [13]), an unexpected lack of resources forced us to review our original programs, dropping that opportunity.

Work Done

Simulation

Our interest for this case study was to compare setup times and verification efforts for traditional verification and ABV on the same design; as the aim of the verification from customer's perspective was to increase the degree of confidence related to the real final working configuration, and data integrity, which was an essential part of the verification, was out of scope for Model Checking, we concentrated most of our efforts in putting in place a comprehensive test by means of a composition of Specman eVC (*e* Verification Component) AHB components utilized for Master AHB traffic generation, AHB arbitration, AHB decoding, for AHB protocol compliance checks, and we developed ad-hoc components and procedures for data integrity tests. Due to inherent complexity (the AHB eVC is composed by 134 files for a total of more than 24KLOC, has a 190 Pages user's guide that is good for first customization steps, but has to be integrated by direct source code inspection to get a good understanding of components dependencies and control fields usage) and lack of previous experience with that eVC, it took us 4 person months to put the basic verification environment in place, for a quite condensed total of 51 files and 5695 LOC.

As the Specman eVC comes with a set of monitors aimed at checking the AHB protocol compliance of the DUV, a portion of the work done for this case study has been related to check the effect of replacing the original monitors by means of PSL code to be used in simulation; to maximize reuse of the set of properties, following D1.1/2 [10] directives, we rewrote the properties aimed at checking protocol compliance by embedding all the properties in an HDL module; by aptly instantiating this HDL module in a slightly changed version of the testbench we were able to replace the original Specman monitors set. The embedding of properties into HDL has required a rewriting of our property set, originally thought for an external usage in unbound vunits. The PSL text tool has been valuable during this stage, even though a less straightforward approach has been possible in this case w.r.t. that of the specification case study: being now the PSL code embedded in comments, it has been necessary to swap Emacs major mode back and forth between Verilog and PSL to have the correct syntax highlighting of PSL keywords. As a further comparison term, we have used FoCs to generate checkers from the original properties, and a secondly changed version of the Testbench has allowed the instantiation of the corresponding set of checkers. As the PSL supported by FoCs is based on unbound vunits, which do not allow parameter specification in current PSL version, we limited the translation to one single port just to have a comparison term.

Model Checking

We applied RuleBase PE to the MPMC for protocol compliance. To reduce the environment overhead we checked one MPMC AHB data port at a time, assuming all other data ports, but that used to configure the MPMC, to be fully free (i.e. not driven by AHB compliant-talking masters). The most annoying step was about defining the working condition by forcing an override of the configuration registers; whilst in simulation the real initialization sequence was used plainly, with RBPE, for sake of state space containment, we directly forced the value of constant registers.

We also tried NuSMV on the DDR interface of the MPMC, by synthesizing the MPMC on a pseudo-library of basic combinational and sequential components and post processing that with a perl script to get a SMV description of the MPMC, but we didn't get to the end of the model preparation stage, i.e. the set of steps NuSMV performs to build an internal representation of the DUV (described as "go" in D3.2/9 [14]).

As last step, we showed the effectiveness of using PSL post-mortem analysis by using the property simulation tool to show the customer how easy the test of a new property can be on a given simulation trace.

Results

The simulation time of the original testbench with no assertions and no monitors was increased by around 10% in presence of Specman monitors, and by 17% in presence of PSL properties. Our trials with embedded PSL and with FoCs-generated monitors didn't show significant differences.

RBPE was able to check the MPMC protocol implementation correctness on slave side in 34 minutes on each single port. No property failed.

R&D Value

The comparison of verification times by ad-hoc written monitors, natively supported PSL, and translated from PSL into DUV language.

Productivity Enhancement

For the part that the two verification environments, the one based on Specman, and the one based on PSL, have in common, i.e. the protocol checking, the productivity enhancement is essentially represented by the information coming from the RBPE application; with the chosen configuration, protocol monitors could have been disabled in simulation, This, depending on how much extra computation is needed for accessory data structures handling (mainly scoreboards) turned into a gain of around 10% of simulation time. For what pertains setup time, if the work had to be redone today, after having collected the necessary experience so to fairly compare the two activities starting from a comparable background experience basis, we would estimate a setup time for Specman around the double of that for pure PSL, mainly due to the coupling interface activity that the Specman approaches requires. It must be said, though, that the Specman environment would bring with itself all the necessary to allow other kinds of intervention, like traffic shaping in simulation that PSL would not allow to address directly just form inside itself.

5.2 Bridge

Design Characteristics

The bridge is a protocol translator from the AHB to STBus **Type 2** (T2). Its main objectives are:

- To correctly handle all the different kind of AHB requests (INCR, WRAP) by translating them into a corresponding STBus format.
- To adapt data format from AHB to STBusT2 (Requests) and from STBusT2 to AHB (Responses).
- To handle the split communication of the STBus.

The bridge connects one AHB master port to one STBus T2 Initiator port and one STBus T2 Target Port.

The block description is composed by 9 Verilog files for a total of 4218 LOC, featuring 166 Inputs, 124 Outputs, 35601 RB-Gates (~14000 after reduction) and 629 FF (~370 after reduction).

Goals

The two main goals we wanted to address with this case study were:

1. Checking effectiveness of new engines in RBPE.
2. Checking the applicability of NuSMV to a reasonably small example.

Tools Used

The Case study was verified using RBPE (D3.2/7 [5]), taking into account D3.1/2 [2] content, and NuSMV (D3.2/9 [14]); FoCs (D3.2/11 [9]) was used to simplify the property adaptation (cf. section "Work Done" below), and the property simulation tool was used during the development of a SystemVerilog AHB traffic generator aiming at trying to check the feasibility of a combined static/dynamic approach D3.1/1 [1]. PSL Text tool (D1.2/6 [15]) has been used for all the property editing work. D3.2/14 [7] was used for ABV simulation post-mortem applications.

Work Done

RBPE was used to check 67 assertions under 25 assumptions, for a total of 1115 PSL LOC, aimed 1at checking both protocol compliance for AHB and STBus sides, and pass-through data integrity. We did the same verification twice: once with an old version of RBPE (v1.22) dated back to October 2004, the second time with most recent RBPE version (v1.32.1, September 2006). Both verifications were run on the same hardware.

To apply NuSMV on the same description, as the PSL style supported by NuSMV is not the same of that of RBPE, looking for a way to avoid property rewriting, FoCs was applied to translate each of the 67 assertions into a corresponding Verilog checker to be instantiated into a top level composing both the original DUV and the specific checker. This was also meant to allow us testing how good the COI reduction of NuSMV is. This extended DUV was then synthesized using an HDL compiler, mapped onto a pseudo-library of elementary cells, and converted into SMV by means of a Perl script. On an Opteron server running RH EE 3.0, NuSMV 2.4.0 compiled with gcc 3.2.3 causes (twice on two trials) a system freeze when the "go" step was performed. Trying the same example on

Athlon XP (32 bit) and on P4 – based GNU/Linux releases (SUSE 10.0 and Simply Mepis 6.0, respectively) the system just run out of 1.5 GB memory after a few minutes of computation. To be sure there were no issue in the translation flow put in place; we explored the behaviour of NuSMV on a trivial shift register design, made parametric in the register length. We saw that a 60 stages shift register was handled pretty easily: on an Athlon XP 2800+ it took 2 seconds to perform the "go" step, and 4 second for reporting the failure of the trivial input-output property

```
check_pslspec -p "always (D -> Q);"
```

that requires to analyze the whole register's chain (`D` is the input of the first stage and `Q` is the output of the last stage). A 600 stages shift register took 50s to be parsed and after 1 hour the same input-output relationship reported above was not falsified yet. We haven't investigated on this any further, but as a matter of fact we haven't been able to apply NuSMV to the Bridge design.

We finally applied the property simulation tool to traces generated from the original DUV by means of an AHB traffic generator written in SystemVerilog, mainly to demonstrate the property simulation tool to ST designers.

Results

The results obtained on the Bridge are summarized in Table 5-1; many of the properties that with the old version for RBPE were reported simply as explored to a certain depth, are now classified either as passing or failing. the improvements reported on both those categories are basically due to the introduction of the Interpolator engine (all the passing and 3 of the failing properties were found using it) and of the Incremental version of the SAT engine, which allowed the identification of 2 further failures and the increasing of the mean explored depth. Reasonably, no truth value change (from passing to failing or the other way around) happened.

	RBPE 1.22	RBPE 1.32.1
Passing properties	13	28
Failing properties	4	9
Explored (mean, dev.)	50 (25.4, 12.1)	30 (30.2 13.1)

Table 5-1: Passing, Failing, and Explored depths for the 67 assertions of the Bridge (Opteron 2.8GHz, 8GB, RHEE3.0, timeout = 1h).

R&D Value

The case study has offered a way to check the effectiveness of the improvement introduced in RBPE during PROSYD life.

Productivity Enhancements

RBPE 1.32.1 has shown an improvement of 115% on the number of passing, of 125 % on failing properties, and 19% increase of the mean explored depth for terminated checks, in a time span of 2 years. Most of the property verification status changing (from terminated to either passed of failed) were actually requiring only a few extra exploration steps, around 10 in most cases, but those 10 steps were

not in the grasp of RBPE yet after almost a year that PROSYD had started. The results of this case study, therefore, witness a major practical improvement happened *during* PROSYD lifetime.

5.3 AMBA

Design Characteristics

The AMBA infrastructure is used for the connection of

- 10 AHB masters
- 27 AHB slaves
- 18 APB slaves

With a programmable memory-mapped connection matrix and a programmable arbitration scheme (highest priority first, non preemptive, with a priority scaling for pending requests); the design features 17152 inputs, 4052 outputs, 9417 FF, 497145 Gates, and 523714 nets, in 231709 lines of code.

The most relevant aspect of this design is its size: the high number of masters and slaves has represented a challenge in terms of environment weight, and verification effort.

Goals

- To check connectivity correctness potentially affected by manual intervention of automatically generated topology, in presence of a quite high number of master-slave pairs, showing the effectiveness of the property-based approach.
- To compare performances of RBPE (D3.2/7 [5]) against a non-PROSYD state of the art commercial model checking tool.

Tools Used

This Case study involved the PSL Text tool (D1.2/6 [12]) and RBPE (D3.2/7 [5]). RBPE application was made taking into account D3.1/2 [2] content; although originally planned to be used for a conjoined application of formal verification and simulation, to check practical property portability between formal and simulation worlds, finally the work was done only statically, and therefore we dropped the application of D3.1./1 [1].

Work Done

The activity was performed according to the following steps:

Memory mapping

Given a table of Address-Slave correspondences for each meaningful master-slave pair described in the specification, a script has been used to generate the pertaining set of assertions and assumptions. Those were aimed at checking both correctness of accesses to slaves, as well as data integrity, by constraining only the traffic originated from the specific master of the pair under verification, and leaving all the other masters almost free. Every time a property was reported to fail, a more detailed environment was used to generate a realistic counterexample. The final set was composed by 2470 verification tasks.

Arbitration scheme

No written specification was available about the arbitration scheme adopted, hence, after a customer interview, simulation has been used to figure out the expected behaviour and to identify a reference basis for following result discussions with our customer. A couple of properties have then been written to generalize the expected behaviour, instantiated multiple times (one for each meaningful masters pair) and finally verified, using a script-based mechanism similar to that of the memory mapping check. The final set was composed by 150 verification tasks.

The two verification sessions were more meant for bug hunting rather than for certification, even though, by the end of the verification activities more than a third of properties were fully proven. As the verification task was known to be hard and hardware availability was putting a strong constraint, we organized the verification in *waves*, initially using very short timeout to skim the property set so to quickly find out which properties were passing (or failing) easily, start collecting information for feedback to customer, and rerunning those property verifications that were killed by timeout, using progressively increased timeouts.

Results

9 Bugs were found, in a total of 3 design respins. On the last version, out of the 2470 Master-Slave pairs, 899 were fully proven, 4 failed, 1309 were proven for just 3 steps, and all the others were distributed from 2 to 131 steps, as reported in Figure 5-1.

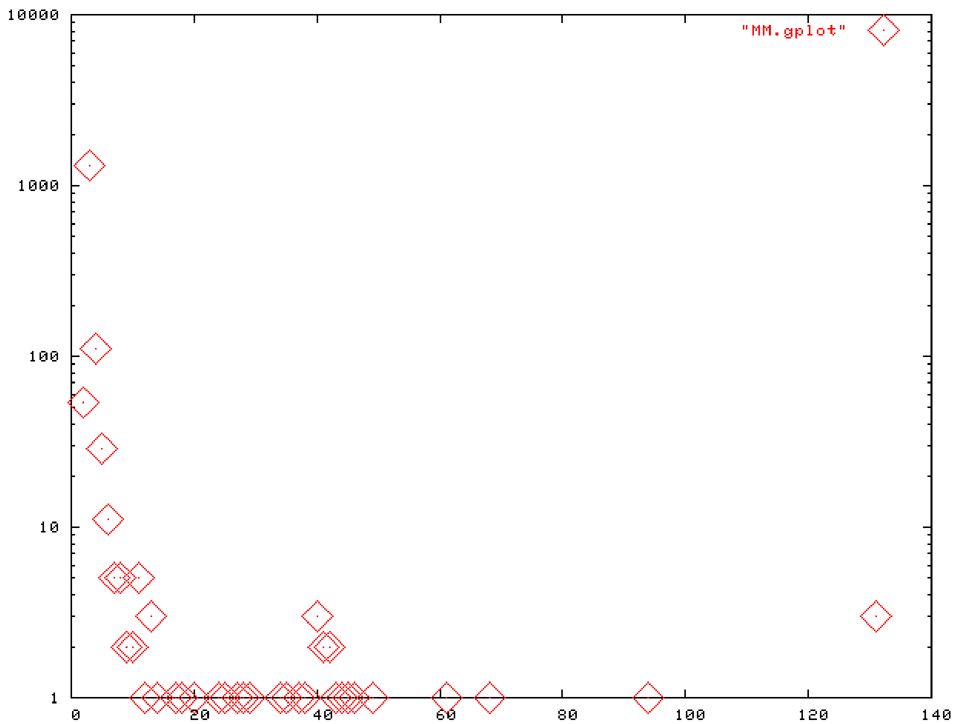

Figure 5-1: Result distribution of partial verification for the Memory mapping check of the AMBA case study: x represents the number of steps, y the number of properties.

Out of the 150 Master-Master pairs, 71 were fully proven, 42 checked for 7 steps, 4 for 8 steps, 5 for 9, 3 for 10, and 1 for 18 steps.

R&D Value

Comparing a RBPE with another, non-PROSYD, commercial model checker.

Productivity Enhancements

On the two configurations taken as comparison terms, one for the memory mapping, and one for the arbitration, RBPE showed to work slightly below 20% better (19.3 % meanly faster on passing and failing properties, 18.7% meanly deeper in explored paths) than the other commercial solution; in particular the Interpolator engine appeared to be effective both in proving properties faster, and in exploring the state space deeper than the other model checker.

6 IBM's Case Studies

Each year RBPE platform is used in a half dozen of IBM projects. For the PROSYD case studies we selected the most representative units from the IBM confidential core hardware. The IBM activity in PROSYD case studies in property based verification is comprised of verifications of a bus interconnection subunit, a cache arbiter subunit, and a CPU scheduler subunit.

The HRL IBM staff used RuleBase PE platform according to deliverables "Combined Static and Dynamic Verification" D3.1/1 [1], "Effective Leverage of Different Static Checking Engines" D3.1/2 [2], "Improved Decision Heuristics for High Performance SAT Based Static Property Checking" D3.2/1 [3], "Research Report on Improved Symbolic Search Strategies and Model Reduction for Static Property Checking" D3.2/2 [4], "Improved Static Property Checking Tool" D3.2/7 [5] and "Porting of IBM tools to support Accellera-standard version of PSL" D3.3/2 [6]. We used property based analysis of simulation traces tool according to deliverable "Property-based analysis of simulation traces" D3.2/14 [7], and FoCs monitors for dynamic property checking according to deliverables "Optimized algorithms for dynamic verification" D3.2/5 [8] and "Manual for Property-based automatic generation of simulation monitors for digital designs, based on algorithmic framework reported in 3.2/5" D3.2/11 [9].

These verification works provided a notable impact on quality of certain IBM hardware. Dozens of bugs were found, and the production cycles were shortened. The PROSYD tools and methodologies helped to achieve acceptable FV results. The reliability of tools and methodologies provide a good impression on IBM design teams, and give good prospects on future dissemination through EU hardware design centers.

The verification case studies proved again that the use of formal verification should be a part of the entire verification strategy. The verification lead should closely review the bugs found by formal methods on a regular basis. He must assess the properties checked, and approve the necessary model restrictions if needed. Formal verification has to support simulation through property based monitors. Also, a property based analysis of simulation traces should be used for debugging and analysis of verification results.

During this work each case study was managed according to methodology that was derived in IBM during the last 5-8 years, and sharpened in the Prosyd project course D3.1/1 [1].

The verification plans were drafted on the stage when the architect has finished the high-level architecture of the design, and designer was in the middle of its implementation. Together with architect and design teams we choose the appropriate formal verification blocks, and made some planning on application of static vs. dynamic verification, and possible reuse of assumptions and rules from modules verification. The blocks (subunits) had a medium size of 1.5-8 thousands of flip-flops, and were supplied with well-defined interfaces. We assumed that formally verifying an average subunit will take 3-4 months allocating 2-3 processors for running the formal verification tool. According to criteria presented in methodology document D3.1/1 [1], these good candidates for formal verification (static checking) have a lot of control functionality and little data processing (interconnection bus); their coverage was hard to reach through simulation (arbiter, scheduler). A data comparison logic in scheduler was chosen as a good candidate for dynamic checking, where checkers are automatically produced from PSL code.

Below we give descriptions of each case study, and provide general conclusions on productivity enhancements, and methodology developments.

6.1 Interconnection Bus

Design Characteristics

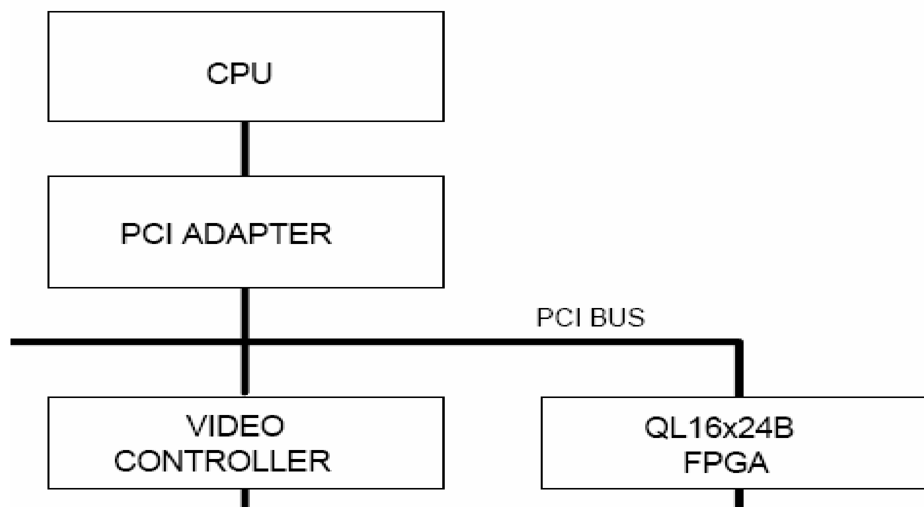

Figure 6-1: The Interconnection Bus.

The interconnection bus provides a shared data path between the CPU and peripheral controllers. The case study deals with a critical block in the interconnection bus. The original implementation has 1400 flip-flops, 1600 inputs, and 42000 gates. It is a medium size design with a lot of inputs. The final set of properties includes 300 properties. A typical property for each output required the reduced model of 60-150 flip-flops, 250-300 environment variables and 30000 gates.

Goal

To increase verification productivity through applying PROSYD enhanced verification methodologies and tools.

Tools used

We used RuleBase PE [1-6] for formal verification of a complex and critical block according to deliverables “Combined Static and Dynamic Verification” D3.1/1 [1], “Effective Leverage of Different Static Checking Engines” D3.1/2 [2], and “Improved Static Property Checking Tool” D3.2/7 [5].

Work Done

4 person months were spent (3.5 PM planned). At the June 2006 review it was stated that 3 PM were spent on this case study, which was completed in March 2006. However, since then the development of the interconnection bus has continued, and additional effort, mostly in regression testing, was spent.

We wrote about 300 properties, in about 1500 line of code. Here some example of properties:

PSL assertion:

assert always (al_tx_queue -> next ! al_tx_queue);

Explanation: al_tx_queue output signal is a pulse.

PSL assertion:

```
%for i in 0..3 do

assert always  ((tar_arb_val & tar_arb_vc(i) &
tar_arb_type(0..2)=010b) ->


next[2] (al_tx_queue & (al_tx_size(0..10)=3)) ;

%end
```

Explanation: if tar_arb_val request is sent, and tar_arb_vc(i) indicates that it comes from vc_i, and tar_arb_type(0..2)=010b indicates that the request is on non-posted type, then two cycles later al_tx_queue should be high and al_tx_size(0..10)= should be equal to 3.

Results

We wrote and checked about 300 properties, 90-95 % of them passed, the rest passed partially (e. g. were checked with bounded model checking up to some bound). Typical properties passed in a few minutes. However the properties which passed only partially have quite long run-times (in fact as long as the time-out set).

We found 25 bugs with formal verification, and induced and checked important debugging in the design.

R&D value

The R&D value of this case study is exploitation of novel Interpolator engine D3.2/7 [5], and refinement of environment modeling and PSL properties derivation processes.

Most of the models to be checked were outside the scope of BDD based Discovery search engine D3.1/2 [2]. The Interpolator search engine allows us to completely check most of the properties without using bounded model checking engines. Our work went in parallel with Interpolator (beta version) development, and we provided a lot of inputs on improving reliability and performance of the tool. For instance, we open several issues on the development database concerning RuleBase bugs (issues #2126, 2451). We also provided feedback on the reduction analyzer options, such as the issue #2437 on different variable visualization filters, and issue #2033 on better matching of reduction analyzer with compilation engines.

For creation of environment we model input signals behaviour using the PSL modeling layer D3.3/2 [6]. A safe approach involves starting with an environment as general and abstract as possible. A property that specific signal sequence does not exist proved with general environment will still hold for the constrained environment. If false negatives appear, the environment can be refined during the verification process. In addition, abstract environments tend to be simpler and easier to write. As a result, it is usually better to start with a general environment and refine it when needed. Indeed, starting with a very precise, very detailed environment will take a long time to write, debug and tune. By refining a general environment, you very well could never have to reach such a level of detail, and even so, it is likely to be at a late stage of the verification project. Bottom up approach is more risky: it is very easy to lose considerable time in tuning in a precise way complex behaviours for some inputs signals with no significant gains. Very often some abstracted behaviour would have done as well.

The same considerations are true for writing properties in RuleBase property language PSL. PSL is simple to learn, yet the way it is used to write properties can have a significant impact on a formal verification project. Simple properties are easy to write, easy to understand, and easy to maintain, while complex property are difficult to tune. It makes sense to start writing the simplest properties you can imagine for your model. This will allow you to assess your model and determine if it represents the design, its complexity, whether it is suitable for formal methods, or whether it should be made smaller by some restrictions. Many very important properties can be expressed in a relatively simple manner. We found that checking even trivial properties uncovered bugs. For example we found bugs by checking that a signal was actually a pulse. When you want to write a complex rule, there is often a simpler version, or a simpler rule (either stronger or weaker) that will hint the same bugs. There are a lot of bugs that can be uncovered by very simple properties. For example a specific signal signal_1, that is supposed to be a pulse, could hold for several cycles in some condition, which could imply very complex high level bugs in a signal signal_2. The bug could be hinted by checking a very simple property using only signal_1:

```
always (signal_1 -> next !signal_1)
```

with model size of 100 state variables, instead of property:

```
always (signal_1 -> next[3] signal_2)
```

with model size of 200 state variables.

It makes sense to first seek out the simpler rule. You may not be able to avoid writing and checking complex properties, however it is a safe policy to write them during a second iteration.

Productivity enhancements

Three RuleBase PE features have brought major productivity enhancement in this case study: engine dispatcher D3.1/2 [2], Interpolator engine D3.2/7 [5] and reduction analyzer D3.2/2 [4]. We first discuss the productivity enhancement of each feature individually, then the overall productivity enhancement of the PROSYD-enhanced tools.

We had to work on a not very mature design, so we had to re-check properties against new version of the DUV quite often, it would not have been possible without the engine dispatcher functionality which allowed us to distribute properties verification on several nodes and therefore achieve reasonable time for complete regression check. We estimate that automation of the regression checking saved us at least a half an hour of manual effort per day, which is 6% of the daily verification time.

The Interpolator search engine D3.2/7 [5] was responsible for the proof of 60% of the complex properties of this project. Thus, approximately 60% of the productivity enhancements noted below can be attributed to this engine.

Reduction analyzer played a less critical role, however it was extremely useful in order to be able to reduce the model size. The new reduction algorithms D3.2/2 [4] give additional reduction of the model till 40% compare to the previous RuleBase version (400 state variables against 300 state variables for similar design, where state variables are the flip-flops and environment variables after reductions). This formal verification project could not have been carried out successfully with the previous RuleBase version, due to the size of the current design.

We can estimate an improvement on 40% in the size of logic, compared with what was in the range of formal tools available before, and an improvement on 20% in verification run time, compared with how long it would have taken if only pre-PROSYD algorithms were utilized. The verification time was decreased up to 15-18% for finding fails (Discovery, SAT) compared to the previous RuleBase version, and to 12-16% for finding proofs (Discovery, Interpolator). The engine dispatcher saves 6% of daily verification time.

It is difficult to quantify quality improvement brought by RuleBase PE. The main reason for this being that pre-PROSYD RuleBase would have failed to verify the design without severe restriction in the behaviour of the inputs. However we think that addition of extra restrictions which would have been needed in order to perform the verification with RuleBase 1.0 would have cause to miss between 15 and 30% of the bugs. Taking into account an improvement on 20% in verification time, we estimate that the rate of bug finding (i.e. number of bugs per total verification time) has improved by $k = 45 - 80\%$ according to the following relation

$$\frac{Founded_Bugs_{RuleBase_1.0}}{Verification_Time_{Rulebase_1.0}} k = \frac{Founded_Bugs_{RuleBasePE}}{Verification_Time_{RuleBasePE}}, \textbf{ or}$$

$$\frac{(0.7 \div 0.85) Founded_Bugs_{RuleBasePE}}{Verification_Time_{Rulebase_1.0}} k = \frac{Founded_Bugs_{RuleBasePE}}{0.8 Verification_Time_{Rulebase_1.0}},$$

and assuming

$Founded_Bugs_{RuleBasePE} = 1,$ **and** $Verification_Time_{Rulebase_1.0} = 1,$

$k = 1.47 \div 1.79$

6.2 Cache Arbiter

Design Characteristics

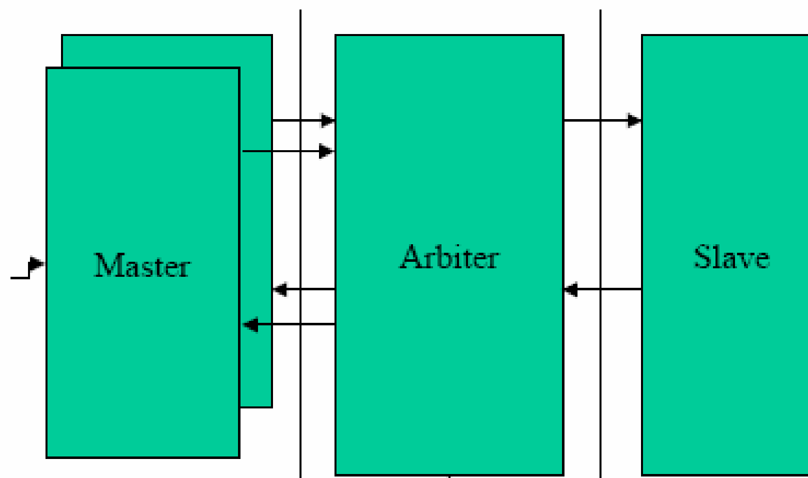

Figure 6-2: The Cache Arbiter.

A cache is a small fast memory holding recently accessed data. A cache arbiter designed to speed up subsequent prioritized access to the data. The verified DUT is an arbiter that is an integral part of an I/O chip. The arbiter VHDL code has 8452 flip flops, 5917 inputs, and 554867 gates.

For most of the checked assertions, the size of the reduced model was 245 flips flops, 188 environment variables, 312 inputs, and 88593 gates. For some of the assertion we needed to constrain the model in order to achieve proves, and the size of the reduced models for these assertions were around 71 flip flops, 60 environment variables, 60 inputs, and 83000 gates.

Goal

To increase verification productivity through applying PROSYD enhanced verification methodologies and tools.

Tools used

RuleBase PE [1-6] was used for formal verification of arbiter according to deliverables “Combined Static and Dynamic Verification” D3.1/1 [1], “Effective Leverage of Different Static Checking Engines” D3.1/2 [2], and “Improved Static Property Checking Tool” D3.2/7 [5]. The novel property based analysis of

simulation traces tool D3.2/14 [7] facilitates trace analysis with PSL assertions giving integration between formal and simulation verification.

Work Done

3 person months were spent (3 PM planned).

We wrote more than 100 properties. Most of the assertions were automatically duplicated using RBPE's m42 preprocessor. These were doable because of a good naming convention inside the design and to significantly ease the verification process.

Almost 10% of the assertions were liveness assertions and checked starvation. The corresponding PSL assertions were:

```
assert always (req -> eventually! ack);
```

The rest of the assertions were safety assertions and used for checking the routing of signals through the arbiters and for checking the functionality behaviour of the arbiters. In most of the cases, few auxiliary variables were allocated in each rule. Many kinds of assertion were used, including sequences and syntactic sugar:

PSL assertion:

assert never{write_req%{mm} | read_req%{mm}) & (start_write%{om} | start_read%{om})); (write_req%{mm} | read_req%{mm})[*]; start_write%{om} |start_read%{om}}!;

Explanation: Master mm cannot win arbitration twice while another master (om) is waiting for arbitration.

PSL assertion:

```
assert never (start_write%{mm} & ! write_req%{mm});
```

Explanation: start_write%{mm} cannot be asserted without write_req%{mm}:

PSL assertion:

```
assert always {slave_req & start_write%{mm}} |=>
{(!slave_wack & !master_wack%{mm})[*]; {slave_wack;
slave_wack & master_wack%{mm}}[*]; slave_wack;
slave_wack & master_wack%{mm} & done & last_wack%{mm}};
```

Explanation: If slave_req=1 and master mm won arbitration then there are zero or more cycles without slave_wack, zero or more cycles with slave_wack (only the even data ack should be rerouted to master mm), and one cycles with slave_wack and done. The slave_wack and done should be rerouted as well at the last cycle. The assertion does not demand that done will eventually occur (in order to keep it safety property) but done is enforced using the environment.

In the case study we did not use fairness constraint even while checking liveness, due to the fact that the designer provided finite numbers that denote the slave delay

when ack is received, and additional confidential fields. This helps in modeling the environment using NFAs instead of using fairness constraints. Moreover, we did not use assume statements in our environment.

Results

All the assertions were eventually verified using 32 bits machines (process memory limit was 2GB). Some were verified using an under-approximated model but most of them used the original environment. Eight bugs were found, including liveness properties. All the safety bugs were found in less than a minute runs using our SAT based BMC engine D3.1/2 [2]. The liveness bugs were found using Discovery engine after a few hours D3.1/2 [2]. Eventually, the run time of all the assertions took a few days. Many regressions were done due to timing and functional changes in the design.

R&D value

The work provided inspiration regarding development and explorations of SAT based techniques that enlarge the capabilities of RuleBase PE. We plan to exploit all the SAT based technologies which were enhanced and developed during the PROSYD project such as the decision heuristics developed at "Improved Decision Heuristics for High Performance SAT Based Property Checking" D3.2/1 [3] and the improvements from "Improved Static Property Checking Tool" D3.2/7 [5]. The importance of expanding SAT capabilities in verification of large models beyond trivial safety properties was learned. We have route our future research and development goals towards this direction.

In addition, working on the arbiter shows that the best way to analyze properties is by using a very easy environment. This helps in checking whether the PSL assertion encodes what we wanted to. In many cases bugs can be found even on a constrained model.

Productivity enhancements

Load balancer, SAT and interpolator engines, and property based analysis of simulation traces tool brought major productivity enhancement in this case study. We achieved an improvement on 20% in verification time, compared with how long it would have taken if only pre-PROSYD algorithms were utilized (15 minutes run per rule against 18-20 minutes in average). We found eight bugs. We estimate an improvement on 43% in the size of a cache arbiter, compared with what was in the range of formal tools available before (430 state variables against 300 state variables for similar design). Many assertions during the verification could not be verified using Discovery, SmartLoc D3.1/2 [2], and even the Interpolator. The addition, of incremental SAT capabilities to Interpolator engine adds significant strength to the engine which solves each one of the problematic assertions in less than an hour. This of course not only saves days of reducing the model size in order to verify these assertion, but also increase the verification quality by providing proofs for the checked assertions using less constrained model.

The arbiter logic is a deep design with a large amount of nondeterminism. Very strong verification engines were needed in order to perform the verification task together with strong CPUs. The SAT and Interpolator engines that were highly enhanced during Prosyd were very fruitful and provided the required capabilities.

In most of the cases under-approximation was not needed due to the strength of these engines. In addition, RBPE's ease of use and the load balancer support significantly increase the productivity D3.1/2 [2]. Note, that while Interpolator engine significantly increases the size of the verified blocks and reduces the time of the verification, the Discovery engine can verify few assertions at the same time while the Interpolator engine can verify only one assertion in each execution. Therefore, RBPE's load balancer support increases the productivity of the tool by a factor of 10 by enabling parallel execution of the Interpolator engine. Also the use of the property based analysis of simulation traces tool ease the simulation wave analysis D3.2/14 [7].

In many cases the generated trace is large and contains many cycles and checking the correct behaviour along this trace is annoying and time consuming. This is where the property based analysis of simulation traces tool gets into the picture. Many scenarios along the generated trace can be checked in no time.

For example, the following assertion is being used in order to achieve a trace which demonstrates a full transaction:

assert always {done}(false).

The generated trace can be huge and checking the right behaviour of the arbiter can be difficult. Therefore, we wrote assertions such as:

```
assert always {aack} |=> {!wack[*]; wack[*]; wack &
done};

assert never (aack & wack);

assert never {!aack[*];done}!;
```

These assertions can be verified in a few seconds using the tool however, manually analyzing these assertions can consume tens of minutes from the verification engineer time. Moreover, verifying these assertions using RBPE would have taken a long time which would have not been justified in an early stage of the project.

During the case study we have used the trace analyzer to check the behaviour of the switch during a transaction. For this reason, we asked RBPE using assertion 1, reported below, to generate a trace showing a transaction with 20 consecutive write_data_acks. The rest of the assertions were used to check the behaviour of the switch during the transaction:

Assertion 2 checks that address_ack always followed by some data ack.

Assertions 3 – 7 check the correctness of signals routing through the arbiter during the transaction. While assertions 8 and 9 check that every request get acknowledged and that acknowledge cannot occur without a request.

```
1. assert never {(write_data_ack(0..1)!=0 )[*20]};

2. assert always ((address_ack(0..1)!=0) ->
   next[2](write_data_ack(0..1)!=0 |
   read_data_ack(0..1)!=0));
```

```
3. assert always (slave_address_ack(0..1)!=0 ->
   master_ address_ack(0..1)!=0);
4. assert always (master_read_data_ack(0..1)!=0 ->
   slave_read_data_ack(0..1)!=0);
5. assert always (master_write_data_ack(0..1)!=0 ->
   slave_write_data_ack(0..1)!=0);
6. assert never{ slave_write_data_ack(0..1)!=0 ;
   slave_read_data_ack(0..1)!=0}!;
7. assert never{ slave_read_data_ack(0..1)!=0 ;
   slave_write_data_ack(0..1)!=0}!;
8. never (master_addres_ack(0..1)!=0 && !request);
9. assert always ((request!=0) -> (eventually!
   master_addres_ack(0..1)!=0));
```

The trace analyzer usage significantly reduces the time spent for analyzing traces provided by RBPE. Without using the trace analyzer, in order to debug we have to look at the trace manually, this can generally take up to 10-20 minutes to understand the trace and check all the corresponding signals. In case that we wanted to automatically check auxiliary properties on the same trace in order to help understand the bug, we used to have to run these assertions under RBPE using restrict to analyze the same trace as previously. This could take up to several hours and thus was a very large overhead. Using the new trace analyzer, the time to do this is only a few seconds. Thus, the added productivity enhancement of the trace analyzer is twofold: it provides a significant speedup of 1-2 hundreds compared to doing the same thing with RBPE (few seconds against minutes, or even hours), and it provides a qualitative improvement, since due to the very large overhead of using RBPE as previously, many debugging properties were simply not run using the previous method.

In this work, we run these assertions several times during the environment refinement process. Doing this work by using RBPE and restrict construction D3.2/7 [5] is not cost effective and can add a few redundant weeks to the verification flow. Therefore, we believe that the aid of the trace analyzer in this case ease our work and significantly reduce the total verification time.

6.3 Scheduler

Design Characteristics

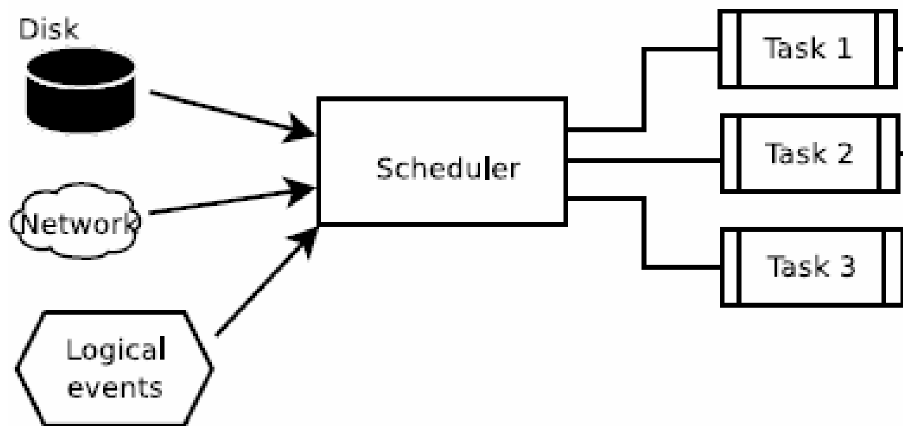

Figure 6-3: The Scheduler in its context.

A scheduler designed to optimize subsequent CPU requests for the tasks. The verified DUT is a scheduler unit from CPU. The scheduler's VHDL includes 3600 flip flops, 700 inputs, and 69000 gates. For most of the checked assertions, the size of the reduced model was 600 flips flops, 70 environment variables and inputs, and 32000 gates. For some of the assertion we constrained the model, and the size of the reduced model was 360 flip flops, 50 environment variables and inputs, and 30000 gates.

Goal

To increase verification productivity through applying PROSYD enhanced verification methodologies and tools.

Tools used

RuleBase PE [1-6] was used for formal verification of scheduler according to deliverables "Combined Static and Dynamic Verification" D3.1/1 [1], "Effective Leverage of Different Static Checking Engines" D3.1/2 [2], and "Improved Static Property Checking Tool" D3.2/7 [5]. The novel property based analysis of simulation traces tool D3.2/14 [7] facilitates trace analysis with PSL assertions giving integration between formal and simulation verification. At unit level wide simulations were done using simulation monitors FoCs D3.2/5, D3.2/11 [8, 9].

Work Done

3.5 person months were spent (3.5 PM planned).

We wrote around 100 properties. Most of the assertions were safety assertions and checked requests priority protocol:

PSL assertion:

```
%for ii in 0..7 do

assert always{dmd_mod_c0&log_req;[*3];
bvtoi(prad_c1(50..60))=ii &
entry_store_%{ii}(0);[*11];var_rej_a3=0}(false);

%end
```

Explanation: If a scheduler in demand mode dmd_mod_c0 has request log_req with an address prad_c1(50:60), and three cycles later this address matches an occupied entry store entry_store_%{ii}(0), then eleven cycles later there should be a reject signal var_rej_a3 associated with that pretest.

Results

All the assertions were verified using an under-approximated model, or the original environment. A few bugs were found using SAT based BMC engine. Constant communications with simulation teams allow using simulation trace analyzer D3.2/14 [7] and FoCs D3.2/5, D3.2/11 [8, 9].

R&D value

The work on the scheduler proves again the effectiveness of methodologies presented in the interconnection bus and cache arbiter case studies. Two bugs were found using trivial properties, such as an output control signal is a pulse, and there are no two acknowledge signals for one request. We also saw how an exploitation of a novel simulation traces analyzer tool helps to analyze wavers, facilitates the verification process, and provides good cooperation with simulation team. At the end of the verification simulation team began regularly provide traces for our analysis. This allows improving a timing protocol of request scheduling, and avoiding unreasonable queues of requests which stack a processing of a whole unit.

According to our requests some issues were resolved by RuleBase PE development team. In particular, they implemented additional features of trace reconstruction mechanism for internal signals.

Productivity enhancements

RuleBase PE, simulation traces analyzer, and simulation monitors (FoCs) demonstrated stability, and ease of use. Simulation traces analyzer show a good performance, still limited number of properties was added through the use of simulation trace analyzer. In any case we can conclude that SAT, property based analysis of simulation traces tool, and FoCs monitors brought major productivity enhancement in this case study. We achieved an improvement on 25% in verification time, compared with how long it would have taken if only pre-PROSYD algorithms were utilized (we compared the performance data collected by RuleBase team two years ago on the logic of similar functionality). We found seven protocol flaws on timing and priority protocol. Using the approach from bus interconnection section, we estimate that the rate of bug finding (i.e. number of bugs per total verification time) has improved by 55-90%.

Using the verification protocol data we estimate an improvement on 36% in the size of scheduler logic, compared with what was in the range of formal tools available before (410 state variables against 300 state variables for similar design).

In this case study we research how to easily use properties in both formal verification and simulation. The salient feature of FoCs is a possibility to reuse rules between formal verification and simulation, and a low-cost setup of simulation test benches D3.2/5, D3.2/11 [8, 9]. Usually, a manual writing and maintenance of checkers is high-cost (especially for complex, temporal properties, e.g. overlapping transactions). Conciseness and expressiveness of PSL, a standard language, insures significantly fewer errors in checkers. The writing assertion both for verification and checker saves time effort. Also your verification methodology remains generally the same in both cases. As in FV, test results are compared against a formal specification, not a human interpretation of a specification document. Unlike FV, which proves the correctness of an implementation vs. a formal specification, specification based verification using FoCs uses conventional testing methods where the program under test is repeatedly stimulated and outputs and/or other values are observed and compared against expected values.

Most of the time properties derived for formal verification were directly used by FoCs. Here we worked according to methodology derived in Prosyd deliverable D1.1/2 [10]. Some properties of formal verification were specially reformulated for FoCs. A liveness property on scheduler data transfer protocol said that after a data transfer eventually another transfer is arbitrated and processed. The formal verification of this property encountered a size problem after 40 cycles working on a model of 2000 state variables. Thus in coordination with simulation team we used FoCs monitor. Here the property was reformulated that during the test after each data transfer every new data transfer is being arbitrated and processed.

Most of the time formal verification and simulation teams found that properties on timing issues, duration, and sequence of signals are always reusable. According to property reuse methodology D1.1/2 [10] we separate the liveness assertions into a special file. Next we selected properties on data, and provided them to simulation team. Communicating with simulation team we observed an improvement on 20% in verification time, compared with how long it would have taken if only checker were written. One deep bug was found through the use of FoCs.

7 Assessment of Methodologies and Tools

6 tools, and 2 methodology documents, corresponding to 9 different PROSYD deliverables, have been used in the execution of the 12 case studies considered in this deliverable; those tools and documents are here assessed in general terms, with the exception of the last two, each of which was applied to only one case study.

7.1 D3.1/1 Combined Static and Dynamic Verification Methodology

This deliverable has played an essential role in 9 out of 13 of the reported case studies. Even though not all the nifty ideas presented in D3.1/1 [1] finally found an application in the reported verification case studies, a reference to that deliverable has been undergoing in many of the practical decision taken along the different case study development lines, either because decisions had to be taken about what to prove statically and what to address by simulation, or because there was an interest in using the same information for simulation and for model checking. Some of the presented case studies have specifically shown the effectiveness of the integration of both verification techniques in performing verification of a complex design.

7.2 D3.1/2 Effective leverage of Different Static Checking Engines Methodology

Deliverable D3.1/2 [2] has played a fundamental role in the execution of 11 of the 13 case studies reported in the present document, practically everywhere there had been an issue on which technique would have worked better for the specific property/design combination considered. In particular, and in relation with D3.2/7 [5] whenever possible, and particularly during early stages of verification tasks, the distributive approach has shown to be winning.

7.3 D3.2/7 Improved Static Property Checking (IBM)

The RuleBase PE formal verification platform [4-6] demonstrates an ability to verify complicate hardware logic. An effective reduction and optimization algorithms reduces the DUT of several thousands flip-flops to an optimized model of hundreds flip-flops. RuleBase PE requires different times for verification. In some cases we have to wait for 6-7 hours for outcome of liveness properties verification. The tool shows its maturity and robustness. During the PROSYD project the number of system crashes during the run dropped on 80% till 1 crash on 10000 runs. The algorithmic cores of most engines (Discovery, Classical, Belzeebub, and SAT) have 4-6 years history of development and constant enhancements. During PROSYD we add new Interpolator engine [2], and enhanced a very powerful SAT engine with incremental analysis algorithm [3]. We improved reductions [4] and trace reconstruction. The novel load balancer and engine dispatcher allow running property in parallel way with different engines on various CPUs [5]. Such enhancements improved the verification (analysis) time in average on 22% (20% for interconnection bus, 20% for cache arbiter and 25% for scheduler), compare to performance of RuleBase version. The size of processed logic increased in average on 40% (40% for interconnection bus, 43% for cache arbiter and 36% for scheduler) according to these case studies.

Experience in static verification over many years has already shown that there may be large and unpredictable differences in the relative performance of different engines on different designs; what is needed is to have as many engines available as possible, and to use them efficiently. It was found that RuleBase PE is designed so that several engines can be used in parallel, and further that one engine can easily be switched off if another is proving more effective. So there is no extra cost in either time or number of CPUs, when the extra engines are available.

7.4 D3.2/8 Improved Static Checking (OneSpin)

The case studies contributed to improving OneSpin's static checking engine in three respects:

- software quality (robustness): all properties resulting from the case studies have been included into OneSpin's regression test bench, thus ensuring that software problems resulting from modifications are discovered immediately.
- Algorithmic performance: various improvements in the engines, (reported in [4] and [7]) have been validated in the case studies, demonstrating an overall improvement during the project duration. As ususal in such comparisons, the performance gain is dependent on the design and property . It ranges from a factor 2 to more than 20.

7.5 D3.2/9 Improved Static Property Checking (NuSMV)

NuSMV is a static property checking with both a BDD based unbounded model engine and a SAT based bounded model checking engine. The tool does not present an integrated front-end for VHDL and Verilog or even a well defined compilation path for HDL designs, and this has represented a serious obstacle that limited our assessment. It was only possible to use the BDD based engine only on properties involving very small portions of the state space, whereas we faced no limitation in using the SAT engines and observed good performance achieved thanks to the improved algorithms related to these engines.

7.6 D3.2/11 Generation of Observers

FoCs tool is used for automatic translation of PSL assertions to simulation checkers. It can translate PSL assertions to Verilog, VHDL, C++ languages. It supports various mechanisms for connecting checkers to the different world-wide used simulation environments, and can create synthesizable code, for emulation purposes. Writing temporal assertions in PSL and then translating them to the design language using FoCs is much simpler task versus writing checkers manually. FoCs is a very simple tool. No additional knowledge in different verification techniques is required. It has a simple and friendly user interface. During PROSYD projects IBM made optimization of the algorithms for generating simulation monitors, using initial knowledge about the behaviour of the simulation environment [8, 9]. In the IBM scheduler case study FoCs showed its effectiveness and usability, and helped to find one bug and several protocol flaws. The simulation team recognized an improvement on 20% in verification time, compared with how long it would have taken if only checkers were written.

7.7 D3.2/14 Analysis of Simulation Traces

Trace analysis is a part of the dynamic verification process. Contrary to simulation with monitors ("online checking") trace analysis is a post-simulation process that does not affect simulation time. It works on the simulation output trace, and verifies it against the provided specification. The simulation traces analysis tool was derived in a framework of PROSYD [7]. The issued prototype was proved to be stable, and relatively usable. During case studies more than 30 issues were issued on bugs and features. There were issues on GUI, and request for additional debugging features. Most of the issues were resolved; still some of them will be resolved in future. The future enhancements have to deal with problem that for the serious case study trace analysis process requires a great deal of the storage space to save the potentially huge simulation trace.

The added productivity enhancement of the trace analyzer provides a significant speedup of 1-2 hundreds compared to doing the same with RuleBase PE (few seconds against minutes, or even hours). Previously verification engineers refused to make simulation traces analysis with RuleBase PE due to this very large time overhead.

7.8 D3.2/15+D3.2/16 Embedded PSL in Verilog (Static+Observers)

RuleBase was successfully used to check PSL assertions embedded in Verilog within the framework of the memory interface. During this case study, more than 20 issues were faced. Most of bugs were fixed and the limitations found addressed.

7.9 D3.2/17 Fast Simulation

OneSpin's approach for fast property simulation has been validated on the Protocol Processor case study – the approach is currently supported by a prototypical implementation, and the case study results justify further work in this topic, towards incorporating it into a generally applicable tool.

8 Conclusion

This deliverable has reported the verification experiences collected on twelve case studies, produced from different industrial perspectives and contexts; compared with D1.4/1 [15], the present deliverable appears much more focused on practical and quantitative aspects; this is a direct consequence of both a more concrete nature of verification w.r.t. specification, and of the leveraging of the experience collected along the PROSYD lifespan.

To summarize all case studies outcomes, this conclusive chapter has been organized in two sections: Section 8.1 gathers answers to questions arisen during PROSYD technical meetings, about why, what, and how to improve the usage of formal methods for addressing verification more effectively than it is done today in the industries; Section 8.2 gives a tabular view of the case studies succinctly listing most relevant aspects for each of them.

8.1 Q&A

The following list is by no mean intended to be either complete or highly representative of issues faced during hardware formal verification application; it is simply the subset of the question list that we, as PROSYD partners, came across during our shared discussions in the various technical meetings happened throughout PROSYD lifetime.

Q: What's measurable benefit of assertions in terms of time saved etc. (e.g. writing assertions vs writing checkers)?

A: Communicating with simulation team we observed an improvement on 20% in verification time, compared with how long it would have taken if only checkers were written. According to collected data through the last five years of dissemination and support of FoCs, the reuse of properties saves in average one day in a week from double work of separate writing checker and FV assertion.

Q: What's the measurable effect of improved algorithms?

A: According to our case studies in average we observed an improvement on 40% in the size of verified logic (40% for interconnection bus, 43% for cache arbiter and 36% for scheduler), compared with what was in the range of formal tools available before PROSYD, and an improvement on 22% in verification time (20% for interconnection bus, 20% for cache arbiter and 25% for scheduler), compared with how long it would have taken if only pre-PROSYD algorithms were utilized. The rate of bug finding (i.e. number of bugs per total verification time) for the scheduler logic has improved by 30-40%, and for interconnection bus even by 40-60%.

Novel simulation traces analysis tool gives a significant speedup of 100-200% compared to doing the trace analysis with RuleBase PE (few seconds against minutes, or even hours). Simulation monitors FoCs save 20% in verification time.

For the Protocol Converter, the size of logic in the range of formal tools was increased by at least 40%. For the Transport Front End, there was no effect.

Q: What bugs (type and quantity) are found using this flow (formal or simulation)?

A: Typos, bugs in arbitration protocol, timing inconsistencies, so-called corner and deep bugs. A few bugs were found by random simulation as well, mainly data-related; (e.g.: a bug in a scheduler data based control logic which tracks and depends on transfer data content), in fact, excluding those strongly data-related on long temporal traces (e.g.: checking the correctness of an FFT computation), there was no type of functional bug in the digital part of designs that could not have been easily expressed, and found with a property-based approach. However, the distribution of bugs detected in the course of verification was different from the pattern in testbench-based verification: some complex bugs were found early, while some elementary bugs were not found until late.

Q: How far can there be a tool-independent property specification?

A: As far as just PSL is considered, and there is not interest in structuring the verification plan so that inheritance mechanism can be exploited to keep modeling and properties aside to allow an orthogonal approach, the only way to have a (commercial) tool independent approach to date is to use embedded Verilog flavor PSL. Ideally, the use of GDL would allow a much neater structuring of verification components, but as a matter of fact only RBPE, as of today supports structured PSL in GDL flavor.

8.2 Summarized Measurements

The present document has described a quite variegated set of case studies; in reporting them, emphasis has been placed on productivity improvements, comparing verification tasks performed without PROSYD-related tools and techniques, with verification tasks performed taking PROSYD outcomes into account. Each comparison has involved (at least) two designs, and (at least) two verification tools and techniques; to try to give a global overview of work done, the relevance of each case study can be qualitatively weighted according to the following table, in which comparisons referred to the same design, hence with biggest chance to give comparable terms, are considered more significant than those relating subsequent releases of the same design, assigning the lowest level of relevance to the case in which designs with some similarities are considered as comparison terms. Analogously, more relevance is given to cases in which the comparison is between an old term verified dynamically and a new term verified statically, giving formal vs. formal and dynamic vs. dynamic progressively decreasing levels of relevance. Weighting the Design relationship as more relevant than verification techniques, the final matrix is that of Table 8-1.

	Same Design [=]	Next Version [+]	Similar [~]
Formal/Dyn [F/D]	Maximum	High	Medium
Formal/Formal [F/F]	High	High	Medium
Dyn/Dyn [D/D]	High	High	Low

Table 8-1: Grading of productivity enhancement results: Columns refers to designs relationship, rows to verification techniques (new/old).

With Table 8-1 acting as a legend, the whole set of case studies presented in this deliverable can be summarized as reported in Table 8-2; from there it is possible to rapidly derive that the degree of relevance of the considered case study was always at least high, and maximum in three cases.

Company	Case Study	Size, measured in number of flip-flops	Comparison Relevance	Relevant Data
OneSpin	Protocol Processor	7K	High (F/D +)	42% P/M reduction in going from simulation to static property checking.
	Tricore2 Processor	2K	High (F/F =)	66% reduction in proof time, thanks to improvements in proof engine.
ST UK	Protocol Converter	800	High (F/F =)	40% improvements in DUV size.
			Maximum	0% improvement from simulation to static property checking
	Transport FE	7K	Maximum	From -64% to 75% improvement from simulation to static property checking
ST F	Bus Protocol	500	High (F/F =)	23% reduction of formal verification run time by property refinement.
	Memory Interface	7K	High (F/F =)	~50% improvements in result determination (from explored to passed).
			Maximum	From -20% to 50% improvement in verification time when moving from simulation to static property checking
ST I	Memory Controller	6K	High (D/D =)	~10% improvement in simulation time by means of ahead property checking.
	Bridge	600	High (F/F =)	> 100% improvement in result determination (from explored to passed or failed) comparing early RBPE-PROSYD (v1.22, Oct.2004) with late RBPE-PROSYD (v1.32.1, Sep.2006)
	AMBA	9K	High (F/F =)	RBPE ~20% better than No-PROSYD property checker.
IBM	Interconnection Bus	1.4K	High (F/F =)	40% improvement in DUV Size. 20% improvement in verification time. From 45% to 80% improvement in rate of bug finding comparing RB1.0 with RBPE-PROSYD.
	Cache Arbiter	8K	High (F/F =)	43% improvement in DUV size. 20% better verification time comparing Pre-PROSYD RBPE with RBPE-PROSYD. From 100x to 200x speedup in checking auxiliary properties when complementing RBPE with the Trace Analyzer.
	Scheduler	700	High (F/F =)	36% improvement in the DUV size 25% improvement in verification time comparing Pre-PROSYD FoCs and FoCs-PROSYD. From 55% to 90% improvement in rate of bug finding.
			High (D/D=)	20% improvement in verification time comparing handwritten checkers with FoCs.

Table 8-2: Case studies Summary: compared techniques, designs relationship, and relevant outcomes.

As it is easy to get from Table 8-2, the quantitative information coming from this deliverable are quite variegated and in some cases widely distributed, spanning from worsening by 64% the

execution time when moving from simulation to static property checking in a case study, to a 75% improvement in another. Even with the 64% deterioration for the DMA of the Transport Front End, the average improvement when moving from simulation to property checking was 16% (includes figures for each of the sub-blocks of the Transport FE and the Memory Interface as shown in Table 3-3 and Table 4-4). Excluding the DMA, which was also problematical in the specification case studies, the average becomes 22%. Excluding also the best result (75% for the SRAM interface of the Transport Front End), the average becomes 18%.

An interesting claim about the reduced effectiveness of static property checking on large and/or complex blocks as noted by ST UK and ST France on the Transport FE and Memory Interface respectively, suggests that problem decomposition has to be considered as a challenge to be addressed strongly, both methodologically, and through tool support.

With regards to dynamic property checking, the figures show an average improvement of 15% when moving to the PROSYD property-based approach.

Of the remaining raw data, one of the major recurring factors is the ~40% improvement in DUV size during the course of this project, meaning that the PROSYD methodology is becoming applicable to larger and larger designs. A second theme is the improvement in verification time between pre-PROSYD and PROSYD-enhanced tools, ranging from 10% at the lowest end through to 66% for the Tricore2 Processor. A third important theme is the improvement in result determination (from inconclusive to passed or failed) of 50% to more than 100%. Fourthly, the rate of bug finding as calculated by IBM for the Interconnection Bus and the Scheduler has improved between 45% and 90%. Finally, the Trace Analyzer (D3.2/14) has proved extremely useful when debugging auxiliary properties, giving a speedup of between 100-200x, thus encouraging designers to use properties in a way that previously would have been impossible due to the extremely large overhead.

Overall, the PROSYD property-based methodology has been shown to improve productivity for both dynamic and static property checking on the order of 15-22%. In over half of the cases, the productivity improvements range from 20% to 75%, indicating the potential of this methodology as more experience is gained in the future. Furthermore, during the project the PROSYD tools have been improved in remarkable ways, and there appears to be a great margin for further improvements pursuing better integrations between simulation and formal techniques, and ways to raise tool interoperability striving for a convergence on common PSL subsets from different implementers.

9 References

1. J. Bormann, A. Fedeli, R. Frank, and K. Winkelmann. Combined Static and Dynamic Verification. PROSYD deliverable 3.1/1.
2. G. Auerbach, A. Fedeli, and E. Zarpas. Effective Leverage of Different Static Checking Engines. PROSYD deliverable 3.1/2.
3. E. Zarpas, M. Roveri, A. Cimatti, K. Winkelmann, R. Brinkmann, Y. Novikov, O. Shacham, and M. Farkash. Improved Decision Heuristics for High Performance SAT Based Static Property Checking. PROSYD deliverable 3.2/1.
4. R. Tzoref, R. Brinkmann, and Z. Nevo. Research Report on Improved Symbolic Search Strategies and Model Reduction for Static Property Checking. PROSYD deliverable 3.2/2.
5. Z. Nevo, T. Veksler, and K. Yorav. Improved Static Property Checking Tool. PROSYD deliverable 3.2/7.
6. A. Orni. Porting of IBM tools to support Accellera-standard version of PSL. PROSYD deliverable 3.3/2.
7. D. Pidan, and M. Shamis. Property-based analysis of simulation traces. PROSYD deliverable 3.2/14.
8. D. Pidan, S. Keidar-Barner, D. Fisman, and M. Moulin. Optimized algorithms for dynamic verification. PROSYD deliverable 3.2/5.
9. D. Pidan, and M. Shamis. Manual for Property-based automatic generation of simulation monitors for digital designs, based on algorithmic framework reported in 3.2/5. PROSYD deliverable 3.2/11.
10. M. Farkash, A. Fedeli, L. Gluhovsky, A. McIsaac, A. Maggiore, and V. Preis. Reuse-aware Property Specification. PROSYD deliverable 1.1/2.
11. C. Eisner, A. Fedeli, M. Moulin and S. Ruah. Property-Driven Specification of VLSI Design, May 2005. PROSYD deliverable 1.1/1.
12. E. Bendersky, and A. Orni.Textual Property Based Requirements Specification ToolPSL Text, PROSYD deliverable 1.2/6.
13. R. Brinkmann, and K. Winkelmann. Improved Static Property Checking Tool. PROSYD deliverable 3.2/8.
14. R. Cavada, A. Cimatti, M. Roveri, and S. Semprini. Manual for Improved NuSMV Static Property Checking Tool. PROSYD deliverable 3.2/9.
15. G. Auerbach, L. Benalycherif, A. Fedeli, D. Fisman, A. McIsaac, and K. Winkelmann. Case Studies in Property-Based Requirements Specification, November 2006, PROSYD deliverable 1.4/1.

16. S. Skalberg, and K. Winkelmann. Fast Simulation of Property Based Specification. PROSYD deliverable 3.2/17.

17. Z. Nevo, D. Pidan, and M. Shamis. Support for Embedded PSL in Verilog Flavor – Static Checking. PROSYD deliverable 3.2/15.

18. Amba^{TM} AXI protocol specification (Rev 1.0) IHI022B ARM Limited 2003.

19. D.Pidan, S. Rabinovich. Support for Embedded PSL in Verilog Flavour – Observers. PROSYD deliverable 3.2/16.

20. R. Brinkmann. Exploitation of RT information in static property checking algorithms. PROSYD deliverable 3.2/3.