



FP6-IST-507219

PROSYD

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Fast simulation of Property-Based Specifications

(Deliverable 3.2/17)

Due date of deliverable: June 30, 2006

Actual Delivery date: July 2nd, 2006

Start date of project: 01.01.2004

Duration: 3 years

Organisation name of lead contractor for this deliverable:

OneSpin Solutions

Revision: 1.0 (public)

| 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 klaus.winkelmann@onespin-solutions.com.

This document is intended to fulfil the obligations of the PROSYD project concerning deliverable D3.2/8, 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 | April 5, 2006 | Creation | Kw | all |
| 0.8 | May 25, 2006 | Technical approach | SKS | all |
| 0.9 | May 30, 2006 | PSL example | kw | 2, 3 |
| 0.91 | June 23, 2006 | Feedback from Consortium | kw | all |
| 1.0 | Dec 31, 2007 | Public version | kw | 3 |

Authors

Sebastian Skalberg, Klaus Winkelmann

Executive Summary

A method to write properties as an executable specification is presented. Rather than simulating the implementation-level details, the specification can thus be simulated, allowing for a speed-up over standard simulators.

Purpose

The purpose of this document is to report the result of task 3.2/17 “Fast simulation of property-based specifications”.

Intended Audience

This report is intended for project partners familiar with the concepts of formal property checking.

Background

Task 3.2 aims at advancing the state of the art in property checking algorithms and tools. As part of this, OneSpin has developed a concept and prototype for fast property simulation. This deliverable presents this concepts and the achieved results.

Contents

| | |
|------------------------------------|-----|
| Table of Revisions..... | iii |
| Authors..... | iii |
| Executive Summary | iii |
| Purpose..... | iii |
| Intended Audience..... | iii |
| Background..... | iii |
| Glossary..... | v |
| 1 Introduction..... | 1 |
| 1.1 Background | 1 |
| 1.2 Starting-point | 1 |
| 2 Technical approach..... | 3 |
| 2.1 Basic concept | 3 |
| 2.2 Small example | 3 |
| 2.3 Scalability | 4 |
| 2.4 Simulation..... | 5 |
| 3 Annex A: Prototype details | 7 |
| 3.1 PSL | 7 |
| 3.2 Features list | 7 |

Glossary

DUV: Design under Verification. An RT level model of a hardware design.

HDL: Hardware Design Language, such as VHDL, Verilog.

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

SAT: satisfiability, the question whether a given set of Boolean formula has a solution.

SAT solver: algorithm that decides a SAT problem and returns a solution if it exists.

1 Introduction

1.1 Background

Task 3.2 aims at advancing the state of the art in property checking algorithms and tools.

For introducing property-based design into practice, it is essential to allow for fast simulation of a specification given as a property set. While this is a hard problem in general, it becomes tractable if the properties are written in an explicit style. The deliverable includes a concept and prototype for fast property simulation.

1.2 Starting-point

As far as verifying a design with respect to its specification, we need to distinguish between three conceptually different aspects of the verification:

1. *What* is computed
2. *When* is it computed
3. *How* is it computed

The first point is strictly a specification issue, while the third point is strictly an implementation issue; the second point is somewhere in between – some timing issues are relevant for functional correctness, some are not. An external memory interface is normally part of the specification, while the timing of data hazard stalls is normally not.

Simulation of HDL is normally a slow affair – for a complex module such as a processor it can be several orders of magnitude slower than a corresponding (at least in theory) simulator written in a high level language like C. The approach presented here has been motivated mainly by processor simulation; however, as the example presented below will show that it is applicable to a wide range of modules.

One of the problems slowing down HDL simulation is the sheer number of signals in the HDL that have no observable impact on the correctness, but rather either provide redundancy for hardware timing issues or have an impact on timing outside of the specification. Signals pertaining to data hazards, for example, are a good example: A functionally equivalent solution would be to flush the pipeline after each instruction; the extra complexity is “only” introduced to maximize the instruction-level parallelism. If we are only interested in an instruction set simulator, simulating data hazard stalls is literally a waste of time.

This is where the specification comes in: By definition, it describes only the minimum information needed to simulate the implementation, disregarding non-specified timing issues. Since our properties will contain the specification, mixed with implementation details, the goal is to represent the specification in a machine-readable form that allows on the one hand verification against it, and on the other hand execution. Thus we would

have a provably correct simulator. We have developed a methodology for writing such specifications, which we will describe in the next chapter.

Before going on, it is important to stress that the argument above works for any granularity of the specification. If we want a cycle accurate simulator for a specific implementation, we can choose to write a “specification” that describes the exact I/O behavior of the given implementation. This will of course slow down the simulation, but we should still get a simulator written at a much higher level of abstraction, and hence with the potential for much faster execution, than the original HDL code.

2 Technical approach

2.1 Basic concept

The main observation is the following: Most, if not all, specifications can be seen as describing a next state function. For processors, this is immediately obvious: For the programmer, one instruction is executed on a given state and updates that entire state in one step. The next instruction executed will work on this new state, produce its own, and so on. To the extent that some architectures have delayed branches that at the first glance change the program counter 2-3 instructions ahead, this can be modelled by not having one program counter, but having three, for example, and every instruction telling how all three change.

Any specification which has a FSM at its heart will also be easily stated as a next state function, as have been the specifications for the FIFOs and the different controllers to which the methodology has been applied.

2.2 Small example

To demonstrate the technique, we will show how a simple counter module can be captured. Although the complexity of the module is quite low, we are still able to write a specification considerably shorter than the implementation (40 lines versus 150 lines). The central part is the `counter_simulator` function:

```
function
  counter_simulator(
    state: unsigned;
    rst, enable, up_else_down, load: bit;
    load_value: unsigned)
  return unsigned is
begin
  if rst = 1 then return initialize(unsigned(0));
  elsif load = 1 then return initialize(load_value);
  elsif enable = 0 then return state;
  elsif up_else_down = 1 then return increment(state);
  else return decrement(state);
  end if;
end;
```

Even without giving the definitions of the auxiliary functions (see Annex A for those), it should be quite clear what the counter module does, giving credit to the function's value

as a specification. The property proving the hardware correct is as simple as the following:

```
vunit counter {
inherit counter_spec;

variable new_state: unsigned(sz_Counter-1 downto 0);
assume always new_state = prev(new_state);

assert
  {new_state = counter_simulator(unsigned(counter_state),
rst,enable,up_else_down,load,unsigned(load_value))
}
|=>
{(counter_state = new_state and
 cnt          = getCounter(counter_state) and
 min_cnt      = getMin(counter_state) and
 max_cnt      = getMax(counter_state) and
 overflow     = getOverflow(counter_state) and
 underflow    = getUnderflow(counter_state))};
}
```

(The min_cnt/max_cnt and overflow/underflow outputs are specified in the auxiliary functions.)

In the property, counter_state is a function describing how the specification state maps to the hardware, the other arguments to the counter_simulator function are inputs to the module. By an induction argument, we now have that the hardware updates the internal state and sets the outputs according to the specification.

The trick is that since the counter_simulator function, by virtue of being a next state function, is completely combinatorial and can hence be executed. The timing issues are encoded into the state (for example, underflow/overflow outputs are set the cycle after the offending operation, while min_cnt/max_cnt is set in the same cycle). Finally, implementation details are completely hidden under the counter_state function, allowing the specification to be separated from the verification.

2.3 Scalability

The counter example might seem nice and clean (especially at the level presented here), but it might be hard to see how it could scale to real designs, let alone a full processor two orders of magnitude bigger.

First, a real example will not be able to prove the correctness of the hardware in a single property, even if it is theoretically possible. In the processor example we split the verification in 40 properties, one for each possible kind of instruction issuing, one for reset and one for stalling. Together, the properties check all possible states of the processor, and hence prove the entire design correct.

Second, the state function, describing the mapping of the specification state to the implementation hides a lot of complexity – exactly the complexity that is invisible to the specification and that we want to get rid of when simulating! The same goes for reachability invariants that may be needed for proving the properties.

Thirdly, and crucially, having properties that only reason from one cycle to the next might seem an unattainable requirement for real hardware with stalls, external interfaces

and other operations of arbitrary length. However, such operations conceptually do not *do* anything apart from move information from one place to another – the specification state does not change.

That this is so may seem strange at first. After all, during data stalls, for example, things certainly seem to happen, indeed, the hardware implements them exactly so that something can happen. Take a code example as

```
ADD R1, R2, #4 /* R1 := R2 + 4 */
ADD R2, R1, R2 /* R2 := R1 + R2 */
```

Let's assume that the second instruction has to stall to wait for the first to compute the sum $R2+4$. Now, in the hardware, it is true, the state changes from having the value of $R2$ (6, say) and 4 as inputs to the ALU, to the value 10 at the ALU output. However, from the user's point of view $6+4$ and 10 are equivalent, the hardware has just represented the same information in a different form.

This all boils down to the way the state function is written, that is, how the hardware state is interpreted. The property we are looking for is that the state is updated immediately at discrete events, between which nothing changes (data is moved around). In the processor example, the relevant events are instruction issue, the states in between are the stalls. The state function is written so as to make hardware states representing the same point in program execution equivalent, regardless of the exact state of the pipeline stages and the registers, which depend on where stalls and other “un-interesting” events happened in the past.

To give some idea of the scalability, we report the sizes of relevant parts of the verification of the counter example compared to the processor example:

| <i>Verification</i> | <i>Properties</i> | <i>State function(LOC)</i> | <i>Specification (LOC)</i> | <i>Implementation (LOC)</i> |
|---------------------|-------------------|----------------------------|----------------------------|-----------------------------|
| Counter | 1 | 1 | 40 | 150 |
| Processor | 40 | 1400 | 2000 | 10300 |

As one can see, the methodology does no magic, we need a number of properties to cope with the complexity issues, and we still need to write 3400 lines of code to verify the processor. However, the 70-80% reduction in size from implementation to specification should give hopes for fast simulation, and provably correct fast simulation at that.

2.4 Simulation

As the properties according to this methodology have the form

```
always state = next_state_fn(prev(state), inputs)
```

it is obvious that this can be converted to a simulator, namely by the scheme as in the following pseudo-code:

```
state := initial_state;
loop forever
  read inputs;
```

```
    state = next_state_fn(state, inputs)
end loop;
```

This has been done in a prototype implementation for our processor example. Implementation details such as controlling and embedding the simulator are not discussed here.

Preliminary tests, using a naïve translation of the specification of the counter module into C++, suggest a speedup factor of between 5 and 10 compared to simulating the RTL directly. We expect the speedup to be more dramatic for modules with more abstraction, such as the processor example. Also, we expect performance gains by optimizing the C++ translator.

3 Annex A: Prototype details

3.1 PSL

(available from OneSpin on Request)

3.2 Features list

Below we give the feature list for this deliverable according to the Prosyd DoW, together

| | Mandatory | Desirable | Nice to have | Present | Ref. |
|--|-----------|-----------|--------------|------------|-----------|
| Standard features | | | | | |
| Pointers to algorithms used | x | | | yes | Chap. 2 |
| List of target operating systems | x | | | yes | below |
| Explanation of coding standards | x | | | yes | below |
| Discussion of license issues | x | | | yes | below |
| User documentation, including documentation of user interface (command line switches) and imported/exported file formats | x | | | yes | below |
| Test suite | x | | | yes | below |
| Standard input language – PSL | x | | | yes | Chap 3.1 |
| Support for Verilog flavour | x | | | no | |
| Support for Other flavours | | | x | no | |
| Features unique to this tool | | | | | |
| Simulator for a property-based processor specification | x | | | yes | Chap. 2.4 |
| User documentation enabling re-use on further property suites | | x | | to be done | |

Target operating systems: Linux, Solaris

Coding standards: The tools are implemented in C and C++, according to the coding standards of IBM Haifa Research Laboratory.

License issues: A FlexLM license mechanism is included for the tool.

User documentation: The user interface and file formats are documented as part of the OneSpin product documentation. The method to specify properties is documented in chapter 2.

Test Suite: Approach was tested on 3 different designs.