Proofs of Hardware Component Descriptions Using Scenarios

Nicolas Ayache

CEA, LIST, Boîte 65, Gif-sur-Yvette, F-91191 France
Nicolas.Ayache@cea.fr

Abstract. In this article, we propose a methodology to formally verify hardware component descriptions. Our approach considers asynchronous descriptions and introduces abstract scenarios as a working representation to reduce the number of behaviours. We aim to use this methodology on the *SystemC* language.

1 Introduction

Hardware Description Languages (HDL) help a lot in circuit synthesis, but also in checking the behaviour of a component from a textual description. Verification aims to minimize the synthesis of erroneous components. To further trust these descriptions, we propose a methodology for their formal verification, besides a dynamic verification by simulation.

To cope with the explosion, in model-checking, of the number of reachable states, we propose a verification methodology based on scenarios and formal proofs, with user-guided abstractions. Within the scenarios, we progressively build Quality-of-Service (QoS) properties of a system from the functional behaviour of its components. Scenarios are sequential and help understanding how a system behaves on a sequence of events.

There are mainly two approaches to develop systems: *top-down* (requirements to implementation) and *bottom-up* (component reuse). Most of the time, a system is built using both approaches. Our methodology focuses on the case where the description of the system is already done but lacks formal verification and QoS properties. The challenge is: what can we use to make the formal verification process less memory and time consuming? We use scenarios to answer this question. They can deal with time and express functional dependencies. Moreover, they are compatible with the top-down (system view [1]) and the bottom-up approaches. In this article, we describe them and explain why they help in formally verifying a system in a bottom-up approach. We illustrate our work on the *SystemC* [2] language. For the *SystemC* semantics, we partially rely on $SystemC^{\mathbb{F}L}$ [3]. On the other hand, scenarios are well known objects as Live Sequence Charts [1], or use cases in UML modelling.

In section 2, we show how we build the semantics of components, systems and scenarios. Section 3 describes the user's point of view and the algorithms attached to the use of scenarios. At last, section 4 concludes.

2 Semantics

SystemC is a *C++* library that defines high-level primitives to describe time-dependant components in which hardware and software aspects intertwine. A system is built from components assembled together through communication channels. *SystemC* comes with a simulator engine to test the behaviour of a synchronous and deterministic execution of the system.

We define a semantics for *SystemC* descriptions, closer to the hardware implementations. Although a *SystemC* description is often far from synthesis, our semantics introduces non-determinism opposed to the rather deterministic *SystemC* scheduler. In this semantics, a local and a posteriori analysis (based on data and control dependencies on SSA forms [4]) can verify the determinism.

2.1 Components and systems

A component in *SystemC* is one or several processes (*C++* methods) with communication interfaces: channels convey streams of values that are accessed by the component through ports. Specific types and instructions are added to the *C++* language to deal with communication and synchronization (e.g. *sc_port*, *sc_mutex*, *notify*, *wait*).

In our semantics, a port is a function from time to values and a process defines relations on ports and shared data (the relations introduce non-determinism). A system is obtained by connecting components, which defines a semantics representing all the interleaving on shared data.

Simulation of a system. A simulation of a system is a particular scheduling of events on ports and shared data. A (potentially infinite) sequence of events is a trace of simulation. An ideal scheduler is defined by giving the hand, at each *moment*, to a component that will execute a part of its process. The latter can be cut out following synchronizing points. The resulting portions of code are called the *actions* of the components. A *moment* is the time to execute an action. The choice of which component should act is open but must respect the following rule: a component in a waiting state or that has no instruction left to execute cannot act. The set of all possible simulations of the system is then an abstraction of the scheduling and the inputs. We use inference rules to derive this set. They express a relation between a system, a scheduler and traces of simulation (sequences of events).

2.2 Scenarios

Given a system, some of its simulations share common properties. A scenario is a description that looks like a program. It acts as a filter on traces of simulation and checks whether:

- the trace of simulation is out of the scenario (an **assume** failed);

– the trace of simulation belongs to the scenario and the system well-behaves (all **assume** and **assert** succeed);
– the trace of simulation belongs to the scenario and the system badly-behaves (all **assume** succeed and at least an **assert** failed).

```
natural nb_prod ∈ [0, +∞[;
while (--nb_prod >= 0)
  { reach(...);
    int x = in;
    assume(1 <= x && x <= 255);
    reach(...);
    assert(out == x*x);
  }
assume(in == 0);
```

The scenario besides accepts on the input port `in` all finite streams of characters that ends with '\0' and all infinite streams of non null characters. Basic instructions such as variable declaration, affectation, test and loop are present, together with a specific primitive, *reach*, that allows executing a part of the process of a component. This gives the possibility to play the role of the scheduler and to cope with states explosion.

But traces of simulation may be infinite; scenarios allow expressing them even though they cannot be executed. This is where abstract interpretation and model checking help. Since abstract interpretation has concrete and abstract domains, we give semantics (1) to the infinite execution of a concrete scenario on concrete traces of simulation (this is for the partial scenarios supplied by the user), (2) to the infinite execution of a concrete scenario on abstract and approximated traces of simulation, (3) to the approximate execution of a complete and abstract scenario on abstract traces of simulation. This ends our hierarchy of scenarios.

3 The algorithmic and the user's point of view

The verification methodology consists in several steps.

First, the user comes with a source code (a description of a system). It is parsed with a *SystemC* parser adapted from *elsa* [5] to distinguish the components and their connections. Then he must choose a component to work with, by providing a scenario to describe its behaviour.

The completion algorithm. When the user describes a scenario, he describes the evolution of input ports. Indeed, our algorithm automatically infers within the scenario abstract streams of values on the output ports and shared data. We adapt abstract interpretation techniques to work on symbolic domains. The result is sound w.r.t. the abstract semantics of complete scenarios working on abstracted traces of simulation. We introduce specific captors to detect non-stability. But in some cases the completion may fail, requiring the user to adapt his scenario. The actions of the environment must be lightly specified in the scenario (some refinements are done when assembling components and thus scenarios). This way, we build a complete scenario that embeds every event and that does not rely on the execution of a component anymore.

The scenario assembly algorithm. Once two or more components have been given scenarios, they can be assembled using the connection code present in

SystemC constructors. The algorithm builds within the first scenario a complete view of the system, and then does the same for the second scenario. Since both views see the same system, a second algorithm merges them into a unique view that is a single scenario containing the product of the processes of the components being assembled together.

Abstracting scenarios. From there, to simplify the code, the user may ask that certain variables are forgotten from now on. By introducing such approximations, the user suppresses irrelevant details, shortens the size of the scenario and guides his scenario to express QoS properties.

Proofs of temporal and functional properties. What is interesting is that proving a property on a scenario is equivalent to proving the property in all the simulations that the scenario represents. For this part, we provide a translation of the scenario into a *C* program accepted by the *Caduceus*/*Why* tool [6]. The user only has to annotate its conditions and the properties he wants to prove.

4 Conclusion

To sum up, we described a scenario-based methodology to prove properties of hardware component descriptions. Completeness of scenario is an interesting issue, but we see it as an extension of the PhD. work.

So far, we have implemented a small common language to the various HDL. The execution of precise scenario on this language gave birth to a prototype. A small set of the *SystemC* language is already supported.

First experiments are promising: we used our methodology on an implementation of a simple FIFO given in the *SystemC* distribution. This allowed revealing possible deadlocks depending on the underlying hardware.

The formalization of complete scenarios is coming to an end and should soon be implemented. Assembly will come right after.

References

1. David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., 2003.
2. The SystemC Hardware Description Language. http://www.systemc.org/.
3. Ka L. Man. *SystemC*$^{\mathbb{F}L}$: Formal Specification and Analysis of Hardware/Software Co-designs. In *JTWSEASTCS*, 2006.
4. David Berner, Jean-Pierre Talpin, Paul Le Guernic, and Sandeep Kumar Shukla. Modular design through component abstraction. In *CASES '04*, pages 202–211, New York, NY, USA, 2004. ACM Press.
5. The Elkhound-based C/C++ Parser. http://www.cs.berkeley.edu/s̃mcpeak/elkhound/sources/elsa/.
6. Jean-Christophe Filliâtre and Claude Marché. Multi-Prover Verification of C Programs. In *ICFEM*, 2004.