TEN–155 CZ technical report number 01

Model checking in IPv6 Hardware Router Design

Jiří Barnat, Tomáš Brázdil, Pavel Krčál, David Šafránek and Vojtěch Řehák

19.6.2002

1 Abstract

This report contains information about the model checking method of formal verification and the first steps of using this method in the project of IPv6 hardware router. Moreover, an overview of some model checking tools is given.

2 Introduction

We present first results of work of the formal verification group participating in the CESNET, z. s. p. o. project of IPv6 router. We summarize the usability of the formal verification method model checking for supplying design of the router.

In the first section we give an introduction to formal verification using the model checking method, we describe it's advantages and disadvantages. In the next section we summarize the usability of this method in our project. In the last section, an overview of software tools supporting model checking is given.

3 What is model checking

If we consider the typical design process of a software or hardware system, we will find out that one of the phases of such a process is the testing phase, so called co-verification phase. Especially in the case of design of a hardware system, it is often very difficult to ensure whether the test vectors we choose for testing are the right ones. So that the testing phase relies on experiences and the deep knowledge of possible holes in the designed system. More specifically, to ensure that the system is really designed as intended, i.e. with no bugs, we need to test it in all possible environments it can appear in. Unfortunately, we cannot practically perform such a complete test. The problem is that using test vectors we can only describe a few of all possible situations in which the system can appear.

Computer science brings us methods which offer a solution for the problem we have just sketched out. These methods are called formal methods. They can be viewed as a collection of mathematical methods that allow one to verify correctness of the model of the system in early design phases. The term verification means in this case taking into account all the possible environments the model can appear in. One can consider this as the "complete test". Dealing with verification in the project of IPv6 router, we specialize on such a method – so called model checking [CGP99]. The main advantage of this method is that it's algorithmical. In other words, model checking is a formal method which allows one to automatically prove whether a model of the system at the suitable level of abstraction satisfies given specification.

Before we can apply model checking, we have to formalize the model of the system and it's specification – requirements the system should satisfy. See the schema in the figure 1. On it's own, the process of model checking results in the answer "YES" in the positive case, otherwise, the counterexample is given. In more detail, in the case of negative answer, one can follow the "bad" trace, which does not satisfy any requirement. Therefore, model checking is also useful for correcting the model.

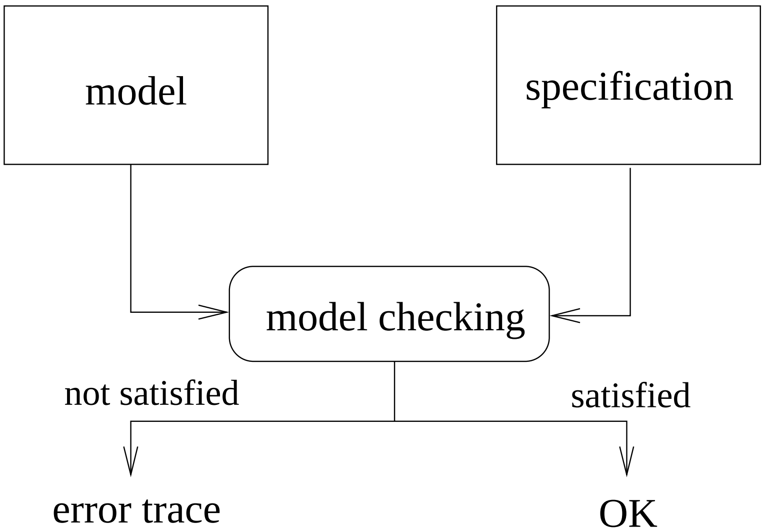

Figure 1: Schema of the model checking method

Technically, algorithms of model checking are based on exploration of the state space of the model. Unfortunately, this is not an easy task. The most painful problem of model checking, and formal verification in general, is so called state explosion problem. In general, there are two reasons for that. The first one comes from the need of dealing with all possible behavior of the environment of the system. Thus, we have to consider all possible inputs of the system. The second reason is dealing with concurrency. Concurrent tasks are viewed as interleaved traces in which actions of each task can appear in an arbitrary order. Hence, we have to consider all those orderings. In our project, we deal mainly with the first problem. An example of this problem is given below.

There exist various academical and commercial tools which support the model checking method. From the external point of view, they differ in languages they use for formalization of the model and it's specification. From the internal point of view, they differ in a way how they deal with the state explosion problem.

3.1 State explosion – an example

Let us consider four queues of the capacity 64 bytes each. We can model it naively as four fields of the type byte and the size 64. How many states can reach this data structure? Each byte can acquire 256 values. In each queue there are 64 different and independent values of the type byte. Thus, one queue can acquire 256^{64} different values. Finally, four queues can result into 256^{256} different states. It is impossible to do the whole state space traversal mechanically.

We have to use suitable data abstraction preserving all important information. For example, if we want to verify only the size sufficiency of each queue, we can abstract away from particular values in each queue and consider only the length of each queue. Thus we obtain the state space of the size approximately 64^4, which is considerably smaller than in the case of the naive model of queues. This abstraction, however, cannot be done automatically, but it must be carefully done by verifier (i.e. people doing formal verification).

There are many other approaches how to solve the state explosion problem, namely automata based, games, symbolic, structural, etc. Nevertheless, all of them are based on heuristics and so none of them solve the state explosion problem at the general level. The groundwork of a verifier is to find the best approach to solve a particular model checking problem.

4 Application of model checking in the project of IPv6 router

In the project of the IPv6 router, we have chosen model checking as the main formal verification method. Our decision is based on the fact, that all members of the verification group are familiar with this method as researchers who work on improving it. Dealing with other methods of formal verification such as equivalence checking or theorem proving would be actual after we find out that we are not able to apply model checking as the only verification method. Moreover, model checking is the most state-of-the-art formal method, and we believe it is the most efficient one for our needs.

The main aim of the work of our group is to ensure that the design made by the designers group is correct and satisfies all the needs. Unfortunately, it is impossible to support the process of design on it's own, how is naturally expected by the designers group. We cannot apply any formal verification method unless we have a formal model of the router. In other words, to verify ideas made by designers at first steps of the design phase, we need a precise behavioral model of the system at the suitable level of abstraction.

We found out that deep understanding between the designers group and the verification group is the key property of such a large hardware design project. Unfortunately, it is not easy to satisfy this property. The main reason for that is the fact, that model checking and formal verification in general are still pioneering methods not spread widely in the community of hardware designers. Moreover, there is often wrong picture of what is formal verification (i.e. model checking) really useful for. In our project we have organized a seminar to present the main ideas model checking is based on and by this way we have tried to inform our colleagues about what they could expect from us and what we can offer to them. Without this exchange of knowledge and opinions it is impossible to do the formal verification in a project of this size.

The main result of communication between our group and the designers group is the need for precise behavioral model of the router. Having this model we will be able to try to prove if it does what is expected. Finding of the suitable level of abstraction will be necessary. This will be our job. Designers are currently working on writing algorithms which describe behavioral aspects of the router.

We discussed with the designers group the choice of possible languages for that purpose. We have found out that it is comfortable for designers to use the C language for description of algorithms. Our work is to transform these algorithms to the input language of the tool we use for verification. We have found out that Promela language used in SPIN model checker is quite similar to C. Unfortunately, there are no automated tools for transformation of C code to Promela, so we have to develop them or to do this transformation by hand. The second idea has the advantage that during this hand made transformation we can get closer to the right meaning of these algorithms. During this process we are forced to consult any misunderstanding with designers. Therefore, we can deeply understand the algorithms we have to verify, and this is one of the most important assumptions for doing the formal verification.

For the final model we have agreed on the use of the VHDL language. The VHDL language was chosen because of it's support for universality and hierarchy, which makes it, in our opinion, the best language suitable for the process of hardware design as a whole. VHDL also supports description of algorhitmical aspects at it's so called behavioral level. Unfortunately, although there are some tools supporting formal verification at the RTL level of VHDL, we have not found any verification tool, which would support the behavioral level. Overview of verification tools we have explored is presented in the next section.

5 Choice of tools

We have been just looking for the most suitable tool for our project. In this section we give an overview of verification tools we found interesting and relevant for our needs.

5.1 Commercial tools

RuleBase[RB] (IBM, Haifa) employs (symbolic) CTL model checking of the Verilog models, VHDL is supported only via Synopsis Design Compiler (DC licence is required). Specification language is a high-level superstructure over CTL, called Sugar. Tool has a trial version (for one month), but we do not consider it worthy without VHDL support.

FoCs[FoCs] (IBM, Haifa) creates so called "simulation checkers". It translates a property expressed in Sugar (RuleBase specification language) into Verilog code, which automatically announces the violation of the checked property during testing when added to the checked Verilog program. Tester need not to evaluate each test run manually. Therefore, massive testing is supported and worthy tester's time is saved.

FormalPro[FP] (Mentor Graphics) is an equivalence checker, which verifies equivalence between RTL descriptions, RTL design and gate-level design, and between gate-level designs. It has a trial version (available for one month). We would rather do model checking, but this tool still can be interesting. However, to evaluate it might be meaningful at the moment, when we have got some RTL and gate-level designs to verify. FormalPro supports VHDL.

FormalCheck[FC] (Cadence) employs linear time model checking [Va01], but verified properties are expressed using the predefined templates only, so the expressivness is restricted. Other key property of this tool is that it supports VHDL. Unfortunately, it does not verify higher level designs than RTL ones.

We conclude that the most interesting tool is FormalCheck, because it has VHDL support and it enables linear model checking. We do not have enough information about these tools because, on one hand, their pages do not present any details, and on the other hand, these tools are not widely distributed among users, so it is difficult to get any references. Now we try to contact the tool authors to get more information.

5.2 Academical tools

VIS[VIS] (University of California at Berkeley) is a CTL (branching time logic) model checking tool with Verilog as a modeling language (it has front end, which translates Verilog into the VIS modeling language – blif-mv). VIS can also verify model in EDIF format via another front-end. The tool is free for academical use, but it has worse support from the authors than the commercial tools (authors do not solve the problems with the Verilog to blif-mv translator, etc.). We are going to evaluate VIS when we have a program in EDIF format.

Prism[PRISM] (University of Birmingham) employs probabilistic model checking,its specification language is PCTL, which is an extension of CTL. Supported modeling languages are Markov chains and Markov decision processes. Tool enables to verify probabilistic properties such as "in the long-run, the queue is full with probability at most 0.05". This can be an interesting property when verifying the time critical parts of the design. Unfortunately, we do not have any experiences with this kind of model checking.

Spin[SPIN] (Bell Labs) supports LTL (linear temporal logic) model checking of its (high level) modeling language – Promela. We have many experiences with this tool. Unfortunately, SPIN is not suitable for the verification of the VHDL design. That means, that we would have to rewrite verified algorithm into Promela and translate the result back into VHDL. We think about using SPIN for the verification of algorithms at the first phases of the design, before they are written in VHDL (but after they are formalized, for instance written in C). However, significant effort of the verifier would be necessary, mainly during abstraction of the verified system.

SMV[SMV] (Cadence Berkeley Laboratories) employs symbolic CTL model checking of the SMV language. This modeling language is unintelligible for the users and CTL is a logic not expressive enough. Otherwise we have some experiences with this tool and we are ready to use it when it is suitable.

Among the academical tools, VIS and SPIN are the most interesting. The first one is interesting because it supports model checking of the programs in the EDIF format, the second tool provides good possibilities to verify algorithms in a higher level language and we have many experiences with that. These tools are all free for academical use.

6 Conclusion

We have presented here the basic ideas of finding the most efficient way how to do formal verification in the project of IPv6 router. To inform other groups of the project about what our group aims at we gave an introduction to the formal method we tend to apply – model checking.

In connection with applying model checking in our project, we have described the problems we will probably deal with and sketch the possible solutions. In the last section we have given an overview of verification tools we found relevant for our project. We have discussed their advantages and disadvantages.

To sum up, it is worth noting that there is no clear procedure for our work in general. Formal verification seems to be a very useful method, but it has not been widely used yet and there are still some problems we will face. Solving these problems is the main goal of our work.

References

[CGP99] Edmund M. Clarke, Orna Grumberg and Doron A. Peled: Model checking
MIT Press, 1999

[Va01] Moshe Y.Vardi: Branching vs. Linear Time: Final Showdown
http://www.cs.rice.edu/ vardi/papers/[1]
invited ETAPS'01 paper, 2001

[RB] RuleBase pages
http://www.haifa.il.ibm.com/projects/verification/RB“_Homepage/[2]

[FoCs] FoCs pages
http://www.haifa.il.ibm.com/projects/verification/focs/[3]

[FP] FormalPro pages
http://www.mentor.com/formalpro/[4]

[FC] FormalCheck pages
http://www.cadence.com/datasheets/formalcheck.html[5]

[1] http://www.cs.rice.edu/ vardi/papers/
[2] http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/
[3] http://www.haifa.il.ibm.com/projects/verification/focs/
[4] http://www.mentor.com/formalpro/
[5] http://www.cadence.com/datasheets/formalcheck.html

[VIS] VIS pages
http://www-cad.eecs.berkeley.edu/Respep/Research/vis/[6]

[PRISM] PRISM pages
http://www.cs.bham.ac.uk/ dxp/prism/[7]

[SPIN] SPIN pages
http://netlib.bell-labs.com/netlib/spin/[8]

[SMV] SMV pages
http://www-cad.eecs.berkeley.edu/ kenmcmil/smv/[9]

[6]http://www-cad.eecs.berkeley.edu/Respep/Research/vis/
[7]http://www.cs.bham.ac.uk/ dxp/prism/
[8]http://netlib.bell-labs.com/netlib/spin/
[9]http://www-cad.eecs.berkeley.edu/ kenmcmil/smv/