Formal Verification of a FIFO Component in Design of Network Monitoring Hardware [⋆]

Tomáš Kratochvíla, Vojtěch Řehák, and David Šafránek

Faculty of Informatics, Masaryk University Brno
Botanická 68a, 602 00 Brno, Czech Republic
{krata, rehak, dawe}@liberouter.org

Abstract. The paper presents our approach of using a formal verification method, the model checking, to verify whether a particular component of hardware design matches its specification. We have applied this approach in the Liberouter project, which is aimed to develop an FPGA based high-speed network monitoring and routing hardware. In the paper, we focus on a FIFO component – the process of its verification, detected errors, and the way of their correction.

1 Introduction

To maximize efficiency of the hardware development process it is necessary to detect the most of potential errors in the design as soon as possible. In addition to classical simulation and testing methods, we successfully use the state-of-the-art automatic formal verification methods. More particularly, we use the *model checking* technique [1], which takes into account all the possible behavior of the design. The other advantages of this technique are simple reusability in newer versions of the design and counterexample generation showing wrong behavior which helps to fix bugs in the design. With respect to the nature of the *Liberouter* design, we use tools based on the symbolic model checking algorithms [2].

The essential prerequisite of the model checking technique is formal specification of verified properties. For this purpose, we use suitable temporal logics [3]. Strong expressiveness of these logics allows us to express all the typical requirements on the component behavior. We perform a special kind of *assertion-based verification*. Commonly used assertion languages, such as PSL [4], do not have the power of temporal logics. On the other hand, the advantage of assertion languages is availability of assertion libraries like OVL [5] or 0-in CheckerWare [6]. However, properties expressed in assertion languages can be easily translated into formulae of some temporal logic.

The temporal logic formulae we use are typically made from the specification and also from the assertions put in HDL codes. Unfortunately, a nontrivial effort is required

[⋆] This research has been partially supported by the FP5 projects No. IST-2001-32603 and IST-2001-32404, and by the CESNET activity "Programmable hardware". The author Tomáš Kratochvíla has been partially supported by the Academy of Sciences of the Czech Republic grant No. 1ET408050503. Vojtěch Řehák has been partially supported by the research centre "Institute for Theoretical Computer Science (ITI)", project No. 1M0021620808, and David Šafránek has been supported by the Grant Agency of Czech Republic grant No. 201/06/1168.

to use model checking tools to verify such formulae on a very complex design. A lot of expert work has to be done manually to suitably simplify the design – sophisticated abstractions have to be employed. For this reason, model checking is not yet the standard method of verification in projects of this kind. In our project, we have set up a team of experts, who specialize in the formal verification.

This paper describes our approach of utilizing the model checking method to verify a FIFO component implemented using Xilinx Virtex^{TM}-II block RAM [7]. The component is a part of an FPGA-based PCI card, COMBO6, which is the pillar of the *Liberouter* [8] project. The main aim of the *Liberouter* project is to develop a high-speed network monitoring and routing hardware with entirely *open design* [9].

The structure of the paper is the following. The Section 2 give a brief overview of the formal verification process we use. In Section 3, the specification of the verified component **FIFO BRAM** and related modeling and verification details are given. Section 4 shows how the specification is formalized using the temporal logic formulae. Finally, the Section 5 describes detected errors, shows the way of their correction, and highlights the benefits regarding reuse of the verification framework each time the new version of the component is developed.

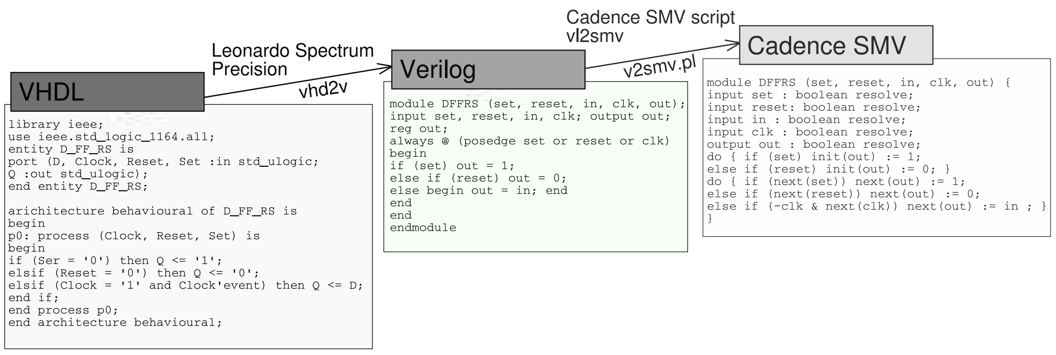
2 Overview of the Verification Process

Before we can apply model checking, we have to formalize the model of the system on the one hand and its specification on the other hand. specification to use model checking.

Input for the verified model is taken from the source codes of the design components that is written in *VHDL*. These source codes are then automatically translated into the input language of the **Cadence SMV** model checker [10]. Using the *Mentor Graphics* **LeonardoSpectrum** or **Precision** tools [11], *VHDL* source codes are synthesized into the *Verilog HDL*. The **Cadence SMV** includes translator of *Verilog HDL* source codes into its native *SMV* language. In essence, *Verilog HDL* and *SMV* languages are syntactically very similar. The scheme of the translation process is depicted in Figure 1 and more detailed information can be found in [12].

In general, we specify the properties of verified components using the *SMV* language. More particularly, the properties are encoded as formulae of *LTL* (Linear Temporal Logic) or *CTL* (Computation Tree Logic) [3]. They are taken mostly from the specification of the component behavior. Other properties are given by *VHDL* designers – either verbal descriptions or assertions written as comments in *VHDL* source codes [12].

Once we have obtained an *SMV* code, its verification is performed. Results of the verification are then reported to developers. The verification process is applied incrementally, i.e. the results achieved after changing some steps of the process are compared with the previous results. For this purpose, a lot of information is included in each verification result – *VHDL* code used, parameters of translation to *SMV*, and preconditions of the verified properties. An XML structure has been defined to store all the result information in the form of a verification report [13]. However, it has appeared very inefficient to write the XML documents by hand. Therefore, we have developed an user-friendly verification environment called *Verunka* [14]. The main features of this environment are



```
library ieee;
use ieee.std_logic_1164.all;
entity D_FF_RS is
port (D, Clock, Reset, Set :in std_ulogic;
Q :out std_ulogic);
end entity D_FF_RS;

arichitecture behavioural of D_FF_RS is
begin
p0: process (Clock, Reset, Set) is
begin
if (Ser = '0') then Q <= '1';
elsif (Reset = '0') then Q <= '0';
elsif (Clock = '1' and Clock'event) then Q <= D;
end if;
end process p0;
end architecture behavioural;
```

```
module DFFRS (set, reset, in, clk, out);
input set, reset, in, clk; output out;
reg out;
always @ (posedge set or reset or clk)
begin
if (set) out = 1;
else if (reset) out = 0;
else begin out = in; end
end
end
endmodule
```

```
module DFFRS (set, reset, in, clk, out) {
input set : boolean resolve;
input reset: boolean resolve;
input in : boolean resolve;
input clk : boolean resolve;
output out : boolean resolve;
do { if (set) init(out) := 1;
else if (reset) init(out) := 0; }
do { if (next(set)) next(out) := 1;
else if (next(reset)) next(out) := 0;
else if (~clk & next(clk)) next(out) := in ; }
}
```

Fig. 1. From *VHDL* to *SMV* language

proper calling of the **Cadence SMV** model checker and automatic generation of XML verification reports.

3 FIFO BRAM Component Verification

Data buffering and queuing are common challenges in high-speed network systems. FIFOs are commonly used to connect components to increase the overall performance. This section shows the verification of a particular design component, **FIFO BRAM**, with respect to its specification.

3.1 Specification

The **FIFO BRAM** component implements a FIFO using internal Virtex^{TM}-II block RAM memories [15]. The component is described in the *Liberouter CVS* [8] directory `vhdl_design/units/common/fifo_bram/`.

The size of the FIFO is parametrized by the generic parameter `ITEMS`. Items in the FIFO are divided into blocks and the component allows detection of the situation when only the last free block of the FIFO is available. The number of items in one block is set by the `BLOCK_SIZE` constant. For details, see the documentation of the component [16].

The component is shown as a black box with input and output ports in Figure 2. There are the following four regular input ports in the component interface:

- `CLK` is a port of the Clock signal
- `RESET` is a port of the Reset signal
- `WR` is the Write Request port
- `RD` is the Read Request port

The data ports of the component are the following:

- `DO` is 16bit Data Output port
- `DI` is 16bit Data Input port

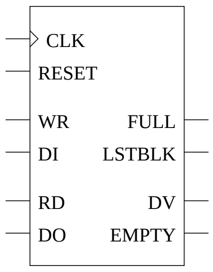


Fig. 2. BRAM FIFO component

And the output ports are defined as follows.

- EMPTY transmits the signal which is active if and only if the FIFO is empty.
- FULL transmits the signal which is active if and only if the FIFO is full.
- LSTBLK transmits the last block detection signal which is active if and only if the FIFO contains just BLOCK_SIZE or less than BLOCK_SIZE number of free items.
- DV transmits the data valid signal which is active as soon as the valid output data are available on the DO port.

Since the **FIFO BRAM** component is parametrized in actual versions the output of the verification is some subset of possible parameter values for which the property is satisfied. It has four generic parameters (ITEMS, BLOCK_SIZE, BRAM_TYPE, and DATA_WIDTH). We have verified *VHDL* codes for each purposeful combination of these parameters. To save our computational resources while covering almost all the hardware designers needs, we have considered the ITEMS parameter restricted. The restricted ITEMS parameter puts a limit to other parameters. This limitation is necessary because the verification process cannot be executed for infinitely many values of the parameters. We do not use parametrized verification techniques [17, 18], because they consume too many time and space resources for verification of such a complex code. However, it suffices to verify the design only for some choices of parameter values. A lot of design errors can be still investigated following this way.

3.2 Modeled Environment

To verify the component, a process which represents behavior of the component environment has to be modeled. We have modeled such an environment process using the **Cadence SMV** language. In our verification we focus mainly on the values of the control signals EMPTY, FULL, and LSTBLK. To enable checking of these signals behavior, we have to observe the FIFO behavior from the outer view and maintain in the environment model the necessary information about the FIFO processing. Therefore, the environment model contains the variable counter which saves the actual number of items in the FIFO. Moreover, the environment process is able to detect when the FIFO items are being read or written.

Two registers, MINUS and PLUS, have been defined to manage the value of counter. The MINUS register is defined to become active just after the moment in which an item has been successfully read from the FIFO. This situation is emerged when the signal RD is active and the EMPTY signal is not active. Similarly, the PLUS register is defined to become active when an item has been successfully written to the FIFO. This situation corresponds to the behavior of the WD and FULL signals.

4 Verified Properties

This section describes the verified properties of the FIFO component. Each property is presented here as an natural language sentence followed with its formalization in the form of a *CTL* formula. Explanation of the CTL logic is not the topic of this paper. Readers, who are not familiar with CTL, can skip the formulae without any lost of understanding.

We categorize the verified properties into two groups – generic properties and specific properties. The generic properties are general requirements on the FIFO behavior. The specific properties are derived from the component specification.

4.1 Generic Properties

Some common properties are applicable to any FIFO equipped with FULL and EMPTY signals and with the counter of items.

The generic properties we have considered are the following:

(1) The FIFO is always eventually full:
```
AG EF ( counter = ITEMS )
```
(2) The FIFO is always eventually empty:
```
AG EF ( counter = 0 )
```
(3) The FIFO is full iff the FULL signal is active:
```
AG( ( full <-> counter = ITEMS ))
```
(4) The FIFO is empty if and only if the EMPTY signal is active:
```
AG( ( empty <-> counter = 0 ))
```
(5) The counter does not overflow:
```
AG( ! ( counter = ITEMS + 1 ))
```
(6) The counter does not underflow:
```
AG( ! ( counter = -1 ))
```
(7) It is not possible to read from and write to the same address at the same time:
```
AG ( ! ( MINUS & PLUS
 & reg_write_addr0 = reg_read_addr0
 & reg_write_addr1 = reg_read_addr1
 & reg_write_addr2 = reg_read_addr2
 & reg_write_addr3 = reg_read_addr3
 & reg_write_addr4 = reg_read_addr4
 & reg_write_addr5 = reg_read_addr5
   )   )
```

The last property is not satisfied only when the read and write addresses are the same (the read registers of the form `reg_write_addr_*_` equals to the write registers of the form `reg_read_addr_*_`) and an item is read from and written to the FIFO at the same time (which corresponds to the formula `MINUS & PLUS`). These generic properties are used also for verification of other FIFO components. We have successfully verified that all these generic properties hold in every version of the **FIFO BRAM** component.

4.2 Specific Properties

An important liveness property of the FIFO is that it should eventually reach the state, in which the `LSTBLK` signal is active. To check whether the signal `LSTBLK` satisfies at least this functionality the following property has to be employed:

(8) The `LSTBLK` signal is always eventually active.
```
AG EF ( LSTBLK )
```

This property was satisfied in every version of the **FIFO BRAM** component. The following "last free block" property is taken from the specification (as it has been already stated in Section 3.1):

> *The* `LSTBLK` *signal is asserted if and only if the FIFO contains* `BLOCK_SIZE` *or less than* `BLOCK_SIZE` *free items.*

This property should be more formally formulated as:

> *In every possible execution of the system, the number of free items in the FIFO is less than or equal to the size of a block if and only if the* `LSTBLK` *signal is active.*

The number of free items in the FIFO is the value of the expression (`ITEMS - counter`). Then the corresponding formula in the *CTL* form is the following:

(9)
```
AG((( ITEMS-counter ) <= BLOCK_SIZE)
     <->
     LSTBLK )
```

Using **Cadence SMV** it has been shown that this formula holds in the actual version of the component. In the previous versions, its validity was violated. In the next section, the explanation of what kind of errors were found and how they were corrected will be delivered.

5 Evolution of the Component

In this section, the evolution of the **FIFO BRAM** component is presented. Moreover, we show the consequences of the fact that the specification of this hardware component was not originally implemented correctly.

In general, it is a harder task to achieve the required functionality in the hardware than in the software. The main reason is that concurrency, speed, and sophisticated resource consumption are very critical properties of such a hardware system. More particularly, it is usually very difficult to instantly change the value of a simple signal such as `LSTBLK` before the next clock tick comes.

5.1 Earliest Versions

The LSTBLK signal in the **FIFO BRAM** component could not be directly computed in the hardware without a significant increase in resource consumption. In the earliest versions of the component, the block referred by the read or write address register was being determined using the highest bits of the address. The LSTBLK signal was raised whenever the read address was referring to the block placed just next to the block referred by the write address (modulo the number of blocks). This FIFO implementation is illustrated on a simple example which follows.

Assume, e.g., that BLOCK_SIZE = 4. In other words, we suppose the number of items in one block to be set to 4. In this situation, when the number of free items in the FIFO is lower than or equal to 4, the LSTBLK signal should be active according to the specification. This example is illustrated in Figure 3. There is shown the part of the FIFO with the engaged items marked with x and the free items represented as empty boxes. Read and write address registers are depicted as arrows referring to the relevant items.

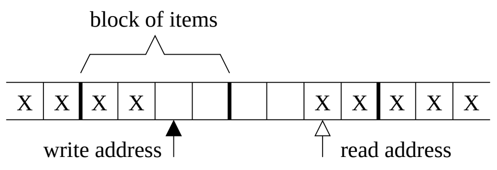


Fig. 3. Read and write address registers refer to items in FIFO with blocks of size 4

The implementation described above is simple and saves a lot of hardware resources. But we have encountered an inaccuracy which has later turned to be a significant problem. The LSTBLK signal was not being generated just when the last BLOCK_SIZE number of free items was remaining in the FIFO. We have verified the intended behavior using the following properties which restrict the interval when the LSTBLK signal should be raised.

The *VHDL* designers focused mainly to save the hardware resources at the expense of accuracy. They believed that the LSTBLK signal was raised when at most 2∗BLOCK_SIZE and at least BLOCK_SIZE number of free items was remaining. We have set up the following two formulae to verify the behavior of the LSTBLK signal:

(i) The LSTBLK signal is not active if the number of empty items is greater than the size of two blocks.

```
AG( (ITEMS-counter) > (2*BLOCK_SIZE)
     -> ~LSTBLK
  )
```

(ii) The LSTBLK signal is active if the number of empty items is smaller than the size of one block.

```
AG( (ITEMS-counter) < BLOCK_SIZE
    -> LSTBLK
  )
```

The formula (i) has been satisfied in every version of the **FIFO BRAM** component which means that the LSTBLK is never active when there are more than two free blocks.

On the contrary, a critical error has been found using the formula (ii). This formula did not hold in the earliest versions of the FIFO. More specifically, the LSTBLK signal turned to zero when the number of empty items became smaller than the size of one block.

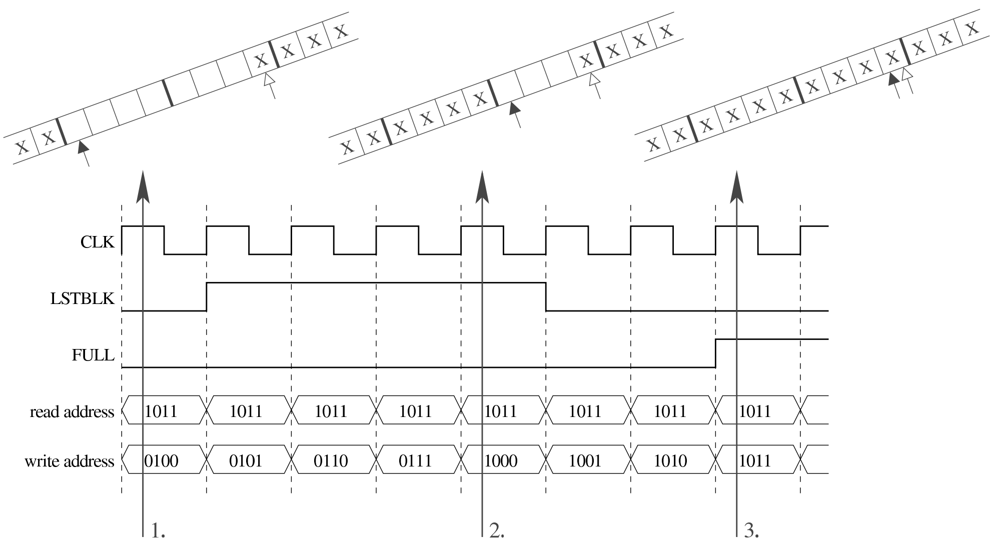


Fig. 4. Timing diagram showing the wrong behavior of the FIFO component

The corresponding counterexample generated by the model checker is shown in Figure 4. Only the signals LSTBLK, CLK, and FULL and the read and write address registers are significant for the observed anomaly. The read address register refers to the same item all the clock ticks while the write address register is increased with each rising CLK edge. The content of the FIFO is observed in three different instants indicated in the figure.

In the first instant, the FIFO contains seven free items. At the same time, the highest bits of the write address, incremented by one, equal to the highest bits of the read address, and hence the LSTBLK signal is changed by the next rising CLK edge. In the next four clock ticks, four items are written to the FIFO.

Subsequently, the second indicated instant comes. At this instant, the highest bits of the write address equal to the highest bits of the read address. This makes the LSTBLK signal to be changed to zero with the next rising CLK edge. This is the incorrect behavior which contradicts the formula (ii).

The last indicated instant is reached just after the next three items are written to the FIFO. In that moment, the FULL signal is raised while the LSTBLK signal is still not active. Now the FIFO is in the state in which, with respect to the LSTBLK signal, it is able to accept some block of items, but in fact it cannot accept any.

A necessary precondition for correctness of this implementation is expressed by the following property:

(iii) The LSTBLK signal is active before the number of empty items is smaller than the size of one block. The formula checks whether the FIFO can reach the state in which less than one free block is available without the rising of LSTBLK signal. The operators G, U, and | stand for *globally*, *until*, and *or* respectively.

```
( (ITEMS-counter) >=
   BLOCK_SIZE U LSTBLK )
|
G (ITEMS-counter) >= BLOCK_SIZE
```

The formula (iii) has been satisfied in earliest versions. In further versions, the validity of this formula has been violated as it is shown in the next subsection.

5.2 Wrong Block Size

The hardware designers have reimplemented the component. Unfortunately, the improvement has been achieved at the expense of consumption of more hardware resources. In more detail, the register cnt_diff, showing the difference between the read and write address registers, has been added. Hence, in this register the number of free items is being stored. The LSTBLK signal is controlled in the way very similar to the previous case – it is active when the highest bits of the cnt_diff register are equal to zero. When the implementation was changed we verified the new *VHDL* sources using the same properties.

The last block formula (ii) was not satisfied again and, moreover, the last block formula (iii) was not satisfied in spite of that it had been satisfied before. The **Cadence SMV** tool generated the counterexample which showed that the behavior of the new FIFO implementation corresponded to the case of the FIFO with the block size one item smaller. More particularly, the highest bits were equal to zero when the cnt_diff register value was smaller than or equal to $2^k - 1$, where $k = \log_2$ BLOCK_SIZE.

We reuse the established verification framework each time a change in source codes or in the specification appears. This process is in most cases very easy and quick (it typically takes minutes of manual work plus time spent on automatic computation). When the specification is changed the corresponding properties have to be changed too.

6 Conclusion

We have demonstrated the way how a very significant bug can be found in the FPGA design using the model checking method. It is worth noting that it could be very unlikely to discover such a bug using standard simulation or testing methods.

Moreover, the verification framework established during this verification process (i.e. the formal CTL specification of FIFO properties) can be suitably reused in future verification of new versions of this component and also in verification of other similar FIFO components. FIFO components are used very frequently in FPGA designs and we have established a base for a library of formal properties which should be checked to ensure that the FIFO design behaves correctly.

In comparison to simulation, which suffers the drawback of being incomplete, formal verification of FIFO implementations with respect to the rigorous specification stated in this paper brings designers another tool which can help them to purify the code and to ensure its correctness. Furthermore, such a process of verification can also lead to backward refinement of component specification, which is contributive to future reuse of the design.

There is another way of using formal verification in projects of this kind. An extensive method is creation of an abstract model of the design. We also apply this approach in our project [19, 14]. The problem of this approach is to ensure the abstract model to be sound. In comparison, the code-level way of verification presented in this paper has the advantage of being very accurate. Our current experiences show that it is useful to combine both approaches.

References

1. E. M. Clarke, O.G., Peled, D.A.: Model Checking. Cambridge, MIT Press (1999)
2. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
3. Emerson, E.A.: Temporal and Modal Logic. In: Handbook of Theoretical Computer Science: Formal Models and Semantics. Volume B. MIT Press (1990) 995–1072
4. Foster, H., Marschner, E., Wolfsthal, Y.: IEEE 1850 PSL: The Next Generation (2005) `http://www.pslsugar.org`.
5. Accellera: Accelera Open Verification Library (2005) `http://www.eda.org/ovl/`.
6. Mentor Graphics: CheckerWare Datasheet (2005) `http://www.0-in.com`.
7. Xilinx: Xilinx, Inc. (2005) `http://www.xilinx.com`.
8. Liberouter: Liberouter Project (2005) `http://www.liberouter.org`.
9. Novotný, J., Fučík, O., Antoš, D.: Project of IPv6 Router with FPGA Hardware Accelerator. In: Field-Programmable Logic and Applications. Volume 2778 of LNCS., Springer (2003) 964–967
10. Cadence: Cadence SMV tool home page (2005) `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`.
11. Mentor: Mentor Graphics (2005) `http://www.mentor.com`.
12. Holeček, J., Kratochvíla, T., Řehák, V., Šafránek, D., Šimeček, P.: Verification Results in Liberouter Project. Technical Report 03/2004, CESNET z.s.p.o. (2004)
13. Kratochvíla, T., Řehák, V., Šimeček, P.: Verification of COMBO6 VHDL Design. Technical Report 17/2003, CESNET z.s.p.o. (2003)
14. Holeček, J., Kratochvíla, T., Řehák, V., Šafránek, D., Šimeček, P.: Verification Process of Hardware Design in Liberouter Project. Technical Report 05/2004, CESNET z.s.p.o. (2004)
15. Xilinx, Inc.: XAPP258 - FIFOs Using Virtex-II Block RAM Application Note. (2005) `http://direct.xilinx.com/bvdocs/appnotes/xapp258.pdf`.
16. Martínek, T.: Basic Documentation of FIFO BRAM Component (2005) `http://www.liberouter.org/cgi-bin2/cvsweb.cgi/liberouter/vhdl_design/units/common/fifo_bram/doc/`.

17. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich ssertional languages. In: CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification, London, Springer (1997) 424–435
18. Abdulla, P.A., Bouajjani, A., Jonsson, B., Nilsson, M.: Handling global conditions in parameterized system verification. In: CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification, London, Springer (1999) 134–145
19. Antoš, D., Řehák, V., Kořenek, J.: Hardware Router's Lookup Machine and its Formal Verification. In: ICN'2004 Conference Proceedings. Gosier, Guadeloupe, French Caribbean. (2004) 1002–1007