CESNET technical report number 8/2005

High-level Modelling, Analysis and Verification on FPGA-based Hardware Design [1]

Petr Matoušek, Aleš Smrčka, Tomáš Vojnar

November 11, 2005

1 Abstract

Implementation of network components in hardware is a trend in advanced high-speed network technologies. Incoming packets can be analysed in fast programmable cards using FPGA. Designing such a system is not easy and requires a detailed analysis. In this paper, we discuss analysis and verification of a non-trivial system-the network monitor and analyser Scampi [Router] that has been developed within the Liberouter project. The Scampi analyser is implemented in FPGA on a special add-on card. It analyses packets incoming with the speed of several Gbps. We created an abstract model of the design and verified several safety properties. Our main task was to check if there is a risk of buffer overflow and how to set the length of buffers to prevent this. First, we made a timed analysis by hand and then we used automated tools - model-checkers Uppaal [Uppaal00] and TReX [TREX01]. In the following text, we show how to model such a complex system and some results of our analysis and verification. We also propose a framework for modelling and analysis of systems where the throughput of requests, their speed, and the length of buffers are important. The proposed models can be reused when verifying and analysing of systems of the given kind.

2 Introduction

One of the current trends in the design of communication systems and embedded applications is to move from a simple one-chip solution with a powerful software in the background to a complex hardware implementation. Programmable hardware is a suitable technology for such a move. FPGA-based hardware can provide a similar functionality of a system as software implemented on

[1]The work presented in this paper has been supported by the FP5 projects No. IST-2001-32603 and IST-2001-32404, and the CESNET project 02/2003

general microprocessors. In comparison to a software solution, programmable hardware is very fast - it is common to communicate in gigabits per second.

The use of programmable hardware is one of the cornerstones of the Scampi project which the work presented here is a part of. Scampi is a European project focused on the development of a scalable monitoring platform for the Internet. Its aim is also to promote the usage of monitoring tools for improving services and technologies. The project develops a network adapter with an extensible monitoring architecture to support a secure and programmable shared monitoring infrastructure.

One of the key requirements put on Scampi and in general on the equipment used in the communication infrastructure - is that it should be reliable and robust to work even under conditions like DoS attacks. In the Scampi project, both the traditional approaches ensuring a correct functionality - namely simulation and testing - are used. Moreover, to achieve an as high as possible confidence in the design, formal verification is extensively used too. The motivation is that simulation and testing can reveal some errors in the design, but they can never prove it to be correct. Moreover, even if formal verification is not always successfully completed, it may find many bugs that tend to be different from the ones found by simulation and testing which is due to a different way of dealing with the state spaces of the considered system.

In Scampi, we apply formal verification on two different levels: (1) on the level of the source code, i.e. VHDL code, and (2) on the level of suitable abstract models. Its advantage is the precise coverage of the real system. The disadvantage is that the system is too complex to be checked in this way completely - we are only able to verify some selected components. To check the key features of the entire system, we work with abstract models.

The source code verification of Scampi is described, e.g., in [TR-4-2004]. In this paper, we discuss our experience from high-level modelling, analysis, and verification of Scampi. Here, a special focus is put on checking the throughput of the system - the optimal length of buffers, timing, and reliability - response time, an appropriate behaviour under extreme conditions such as a DoS attack.

The Scampi design includes many different VHDL components - input buffers, preprocessing units, a lookup processor, output buffers, a statistical unit, etc. We divide the model of the system in three kinds of model components: a model of the environment (generators), a model of buffers (queues, channels), and a model of executive units. We show how the different model components may be constructed and in the case of generators and buffers, we obtain general templates that may be reused in different models of systems of the considered kind.

We further discuss formal verification that we have performed over the obtained model using the Uppaal [Uppaal00] and TReX [TREX01] model checkers. Using TReX, we have performed parametric verification allowing one to automatically synthesise certain constraints on the parameters of the system necessary for its correct behaviour. We also present some manual reasoning that we have initially performed over the system and which we confirm and extend by automatic methods.

Plan of the paper. We start with an overview of the Scampi design. Then, we show how we analysed the system manually. The next section concerns modelling of the system - the creation of an abstract model with respect to the properties we want to verify. We show models of basic parts of the system - the environment, buffers, and executive units. Next, we discuss the results of a fully automated analysis using Uppaal and TReX. Finally, we close by some concluding remarks.

3 Scampi Design

Scampi is a name of the network adapter working at 10Gbps speed. The system design consists of several components. In this paper we are interested in front part of the system - input buffers, preprocessing units (header field extractor), and searching units (lookup processor, processing units).

Scampi adapter reads data from one input port and distributes them into four independent paths working in parallel. Every path consists of several executive components and several buffers. Here, we shortly explain these components and their functionality. A first part of the Scampi adapter is depicted at Figure 1.

When an IP packet comes, it is forwarded to one of the paths and marked with a time stamp generated by a Time Stamp Unit (TSU). The packet waits in IO_BUF buffer (which is FIFO) to be processed by a Header Field Extractor unit (HFE) at first. This component divides the packet into two parts - a header and a body of the packet. The IP header is translated into a unified header - an adjusted structure that stores the main information about the packet including a source IP address, a destination IP address, MAC addresses, port numbers, a VLAN id, etc. Unified headers are stored in UHFIFO buffers. The body of the packet is stored in a Packet FIFO (PFIFO) and flows by another path through the system. The searching core of the Scampi is a Lookup processor component.

3.1 Lookup Processor

Lookup processor (LUP) as showed in Figure 2 is the core of the Scampi design which classifies packets according to rules defined over IP headers. LUP has four input buffers with unified headers (UHFIFO) and one output. An output

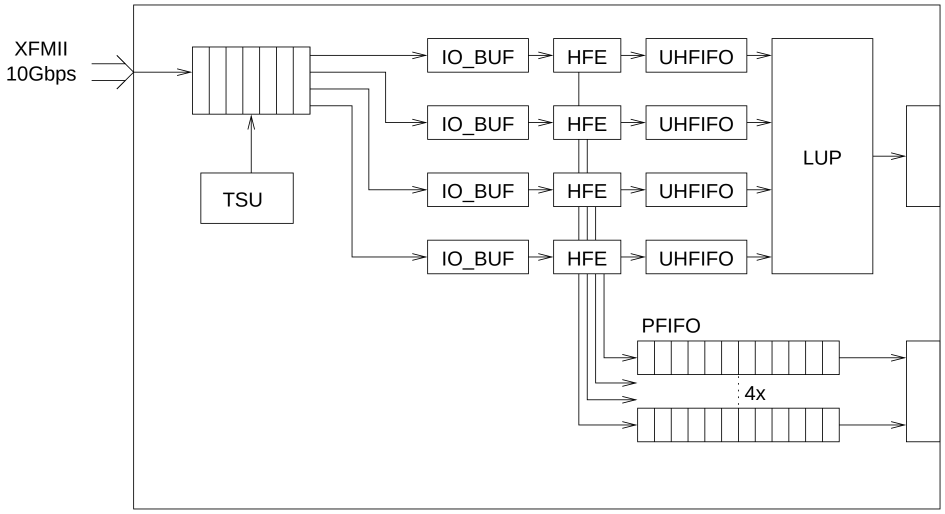

Figure 1: The first part of the Scampi design

data is formed by an identification of the packet and a type of the packet. Every matching rule is encoded into two parts. The first part is a packet matching (parallel searching) encoded into the TCAM memory (Content Address Memory), the second is a sequential searching SSRAM memory performed by Processing Units (PU). In this work we don't examine process of building recognition rules and matching a unified header - if interested in, see [LUP].

Recognition works over the unified header - one part of the header is matched in TCAM (we call it a MP part) and the other part is used for the sequential searching (a SP part). Unified header has the size of 1024 bits, the part for the parallel searching has 256 bits. CAM block sequentially loads MPs from all four inputs (UHFIFOs) and builds a block of data for the matching in the TCAM memory. TCAM compares the data according to matching rules. This is done simultaneously for all four inputs. We can say that there are four parallel paths in the system. Every path has one UHFIFO, one FIFO of results (RFIFO) and one PU.

After TCAM comparison, there is a sequential searching done by PU. PU loads the result from the matching process and uses it as an entry point of the sequential program stored in the SSRAM. When the sequential searching program is done, the PU takes the result from it.

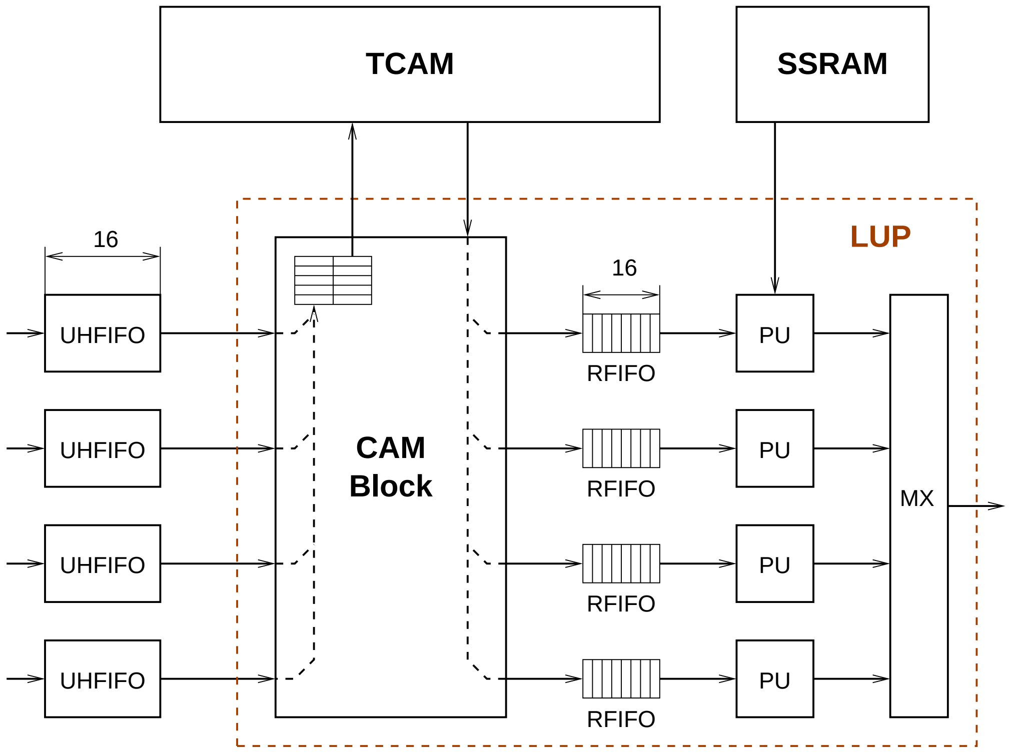

Figure 2: Structure of Lookup Processor

Every PU has to wait for a multiplexer MX to receive data from the Lookup processor. The result of PU processing represents a numeric information what to do with a packet - e.g. increment the counter of the dangerous packet occurrences, forward the body of the packet to the software, copy a input packet to every output (broadcast), or simply release the packet from the Scampi system.

4 First Analysis of the System

Here we will present our result of Scampi analysis made by hand - a calculation of minimum packet throughput through the system. We are only interested in the throughput caused by the Lookup processor. Minimum throughput can be calculated from the number of results per second and the minimum size of the packet. This computation can be done because the system is synchronised by one global clock. We must calculate the delay for the matching and the sequential searching to obtain the result frequency. There are two parts working in parallel in Lookup processor (Figure 3). The first is a matching part (CAM Block and TCAM) and, the second a sequential searching (four Processing Units).

4.1 Rule Matching Analysis

Unified header stored in UHFIFO has the size of 1024 bits. The matching part of the size 256 bits is loaded to the TCAM memory. TCAM works in parallel for four input buffers. CAM Block loads four 256 bit parts from four UHFIFOs and prepares them for the TCAM memory. We will concentrate in our computation only on one UHFIFO now. Data bus width between UHFIFO and CAM Block is 32 bits, and we have to load 256 bits. So, it takes eight ticks to load the matching part of the unified header from one input (UHFIFO).

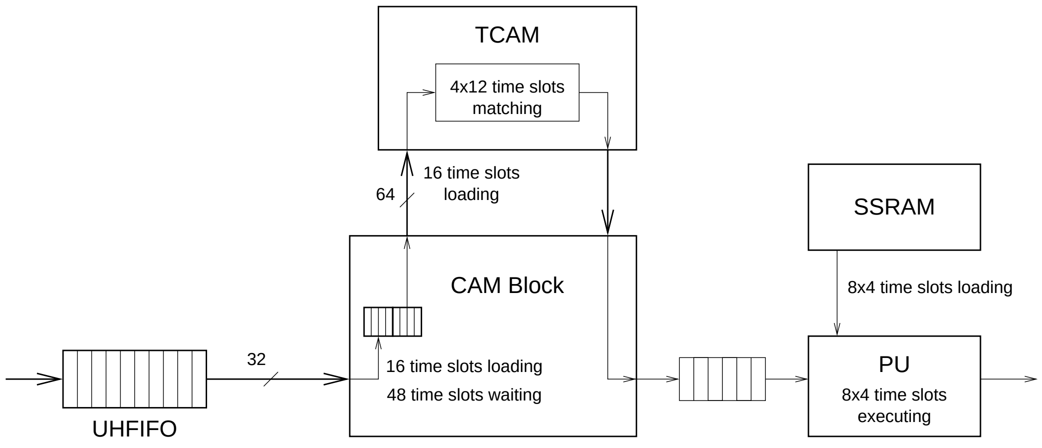

Figure 3: Lookup Processor with timing

In Scampi design two 32-bits blocks are loaded at the same time from two UHFIFOs while other two UHFIFOs are waiting. Loading process from all four ways takes exactly 16 time slots together. A data from the CAM Block is loaded to the TCAM within the same time. It must be loaded 1024 bits via 64-bits data bus. This 1024-bits block is formed by four 256-bits parts of four input buffers that will be processed in TCAM in parallel.

The loading into TCAM takes also 16 time slots. When the data is loaded into the TCAM, the matching process starts. TCAM matches four results, each of them takes 12 time slots. All four results are matched within 48 time slots. While doing this, the CAM Block is stopped and waits for the matching process. Altogether the matching process with data loading takes 64 time slots. That means every 64 time slots CAM block provides four results for four input queues.

After this, four Processing Units process the second part of the matching - a sequential searching. The sequential searching is done by execution of a program stored in the SSRAM. The program consists of several instructions. It takes four time slots to load an instruction from the SSRAM memory. The execution is done during next four time slots, so the minimum delay of execution of one instruction takes eight time slots. With eight instructions (at most) PU provides the same throughput like CAM Block and TCAM. That is why there are supported programs with one instruction at least and eight instruction at most. The four-paths output is serialised by the Multiplexer, so that every 64 time slots it forwards four results - one result takes an average 16 time slots.

4.2 Throughput Analysis and Consequences

The minimum throughput of the Scampi system caused by the Lookup Processor can be now calculated for the smallest supported packets which have size of 64 bytes. The system is set to frequency 50 MHz, time slot takes 20 ns. Rule matching is done within 16 time slots which is 320 ns. The minimum throughput of the system is $(64*8)/(320*10^{-9}) = 1.6$ *Gbps*.

Since the Processing Unit provides the same speed as the TCAM memory, the size of a result buffer between CAM block and PU is at most one. We can not eliminate the whole buffer because of the difference in TCAM and PU timing. What cannot be easily done by hand is finding the optimal length of UHFIFO buffers or to detect deadlock system because of buffer overflow.

5 Modelling the Scampi Design

Scampi design is a complex system that cannot be fully modelled and analysed. Some parts are not interested from point of view of analysis of the throughput of the system. In this section we present several models of components that are important for further analysis of the system. We don't present here a full design. Here, we concentrate on basic parts of the design that are typical for many hardware designs - queues, packet generators, processing units etc.

In our approach we recognise three basic types of components that occur in some form in many complex systems:

- FIFO queues (or buffers, channels) - they can be deterministic or non-deterministic, stochastic. We can model lossy queues, delayed queues etc.
- executive components - e.g., processing units (lookup processors, preprocessing units), multiplexors etc.
- environment - e.g., generators of incoming requests (packets), output units (waiting for a result)

5.1 Modelling FIFO Queues

A FIFO queue (buffer, channel) is a typical abstract data structure that contains a sequence of stored data. FIFO queue is used to represent transmitting channels, intermediate buffers between a processing unit and a memory etc. We can have lossy queues where some data may be lost. These are important to model communication channels. There are delayed queues where data are delayed. Then we can model parametric queues where a symbolic constant value - a parameter - defines the maximum length of the queue. In parametric verification we are interested in values for which the parameter satisfies expected properties. This approach is called parameter synthesis [AHV93]. In our abstract model we do not observe a type of data, so we need only one value - the number of items in the queue.

Simple FIFO queue. Simple FIFO queue can be easily modelled in abstract way consistent with the goal mentioned above as a finite automaton with three states - empty, nonempty, and full. There is a special variable *count* that represents a number of items currently present in the queue. If we add a new item in the queue, counter *count* is incremented. If it is removed from the queue by reading, counter is decremented. The length of the queue is a constant value *FIFOsize*. A model of simple FIFO queue is depicted in Figure 4. Notation *x?* means reading the queue into a variable x, *x!* means writing *x* in the queue (handshake communication between two processes).

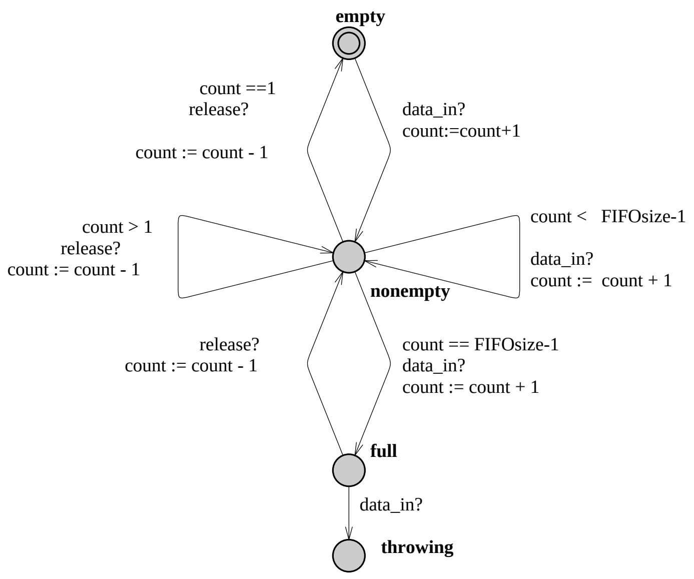

Figure 4: Model of a simple FIFO queue

Notice a special state named *throwing*. This state is a so called observer - a state that observes examined behaviour of the system. Here we are mostly interested in queue overflow. If queue overflows a system moves to the state *throwing*. In verification we define property "buffer will never overflow" as a logical statement "the system never enters the state *throwing*", *AG ~ throwing* in CTL logic.

FIFO queue with delays. Delayed FIFO is a model of FIFO with time where it is guaranteed that every request is delayed at least *DELAY* time units before released. Delayed FIFOs are modelled using Timed Automata [Alur99]. In Figure 5 there is a delayed FIFO queue with clock *y*. Transitions that release an element of the queue are augmented with time constraints allowing to release an item only if the constraint is satisfied. The time of release is non-deterministic and must be greater then *DELAY*, i.e., *y >= DELAY*. If we want to define precise maximal delay of the queue we have to add an invariant *y <= MAX_DELAY* to each state where data is released.

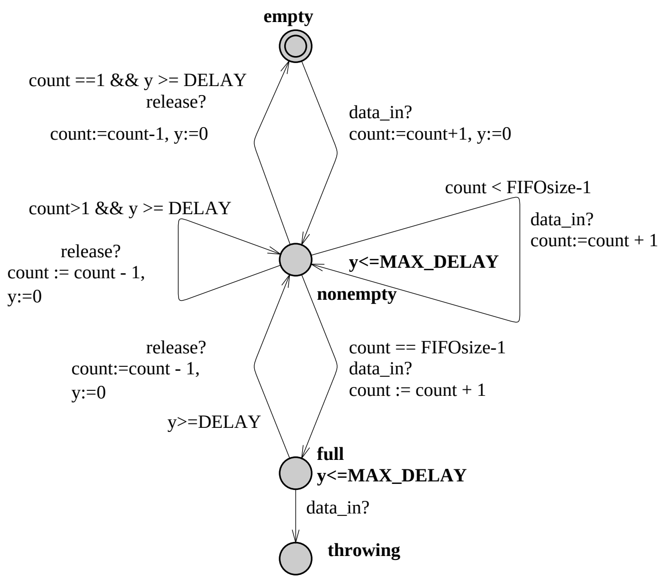

Figure 5: Model of a delayed FIFO queue

Lossy FIFO queue. Lossy FIFO queues can be timed or untimed. In Figure 6 we consider untimed lossy FIFO queue.

The model includes a special counter lossy period *lp* that counts a number of successfully sent elements. If the count reaches constant *NLOSS* it generates a loss that means it decrement the number of items of the queue without sending it to the next process. In our model we have NLOSS equal to 5, i.e., every fifth element is lost. Using this abstraction we can model queues where we know the rate of losses. Distribution of losses is uniform, that means, we don't model chunks of losses.

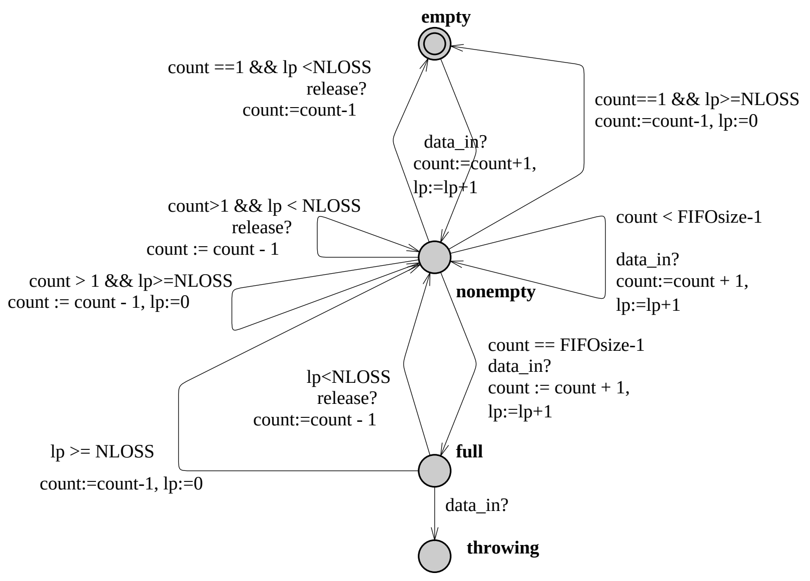

Figure 6: Model of a lossy FIFO queue

5.2 Modelling Executive Components

Executive components by contrast to the buffers can not be modelled using some *pattern* of the model. While creating the models we have to follow the purpose of the verification process. In the Scampi system design we are interested in timing of the components because the number of items in the buffers depends only on the delay of the executive components.

If executive components have an accurate timing plan, we distinguish two kinds of stated in the model. The first is an urgent state that we use for observers. The second type is a state which models delay of the system. This state (e.g. *sending_result* on Figure 7) has an incoming transition which resets a clock (*t := 0*), an invariant that defines a time constraint over clocks for the state, and an outgoing transition constrained by condition on a clock (*t == delay*).

Modelling CAM block and TCAM model. The simple analysis of comparison of delays of CAM block and TCAM shows that matching process in TCAM takes more time then preprocessing in CAM block. Our model, on Figure 7, has two waiting states: *filling* when data from a UHFIFO is loaded and the TCAM memory is filled with data (constant *FILL* is set to 16 time slots), and *matching_result* state which waits for one result from TCAM (constant *RESULT* is set to 12 time slots). The result is subsequently sent via channel *rfifo_in* to the next component. When the fourth result is forwarded, CAM block and TCAM start loading new four parts of the unified header.

Modelling Processing Unit. Processing units (Figure 8) are the executive

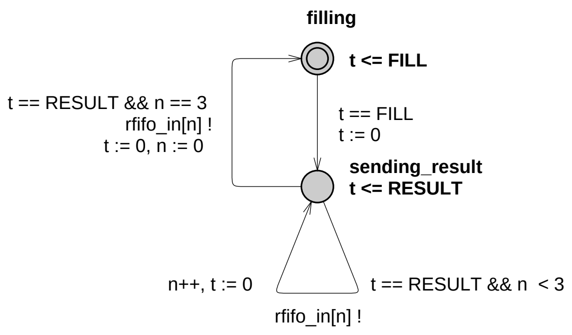

Figure 7: CAM block and TCAM model

components that perform sequential searching for an appropriate matching rule. The process is done by executing instructions stored in the SSRAM memory. Each PU waits for its time slot to access SSRAM and loads instructions to be executed. Loading and execution take together eight time slots (state *executing*). We expect to have at most eight instructions to be executed, so the transition is allowed when *t==seqsearch*. Then the result is sent to multiplexer.

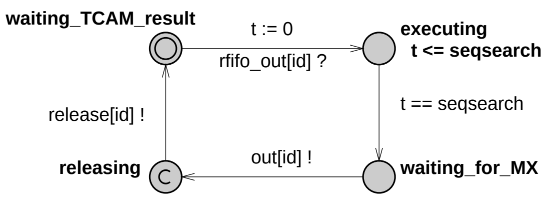

Figure 8: Processing unit

Multiplexer. Multiplexer subsequently checks every PU in every time slot whether it already has a result to be forwarded. In our model we abstract out of subsequent checking every PU because of complexity of the model. However, we are able to compute the maximum delay caused by subsequent polling - it is 4 time slots delay for four PUs.

The model of Multiplexer (Figure 9) has two interesting parts. The first part reads the results of PUs (states *start*, *first_out*, *waiting*, *has_result*), the second part is a observer (formed by other states) which is used to check the throughput and the delay of the system.

The forwarding part has two states in which the system is waiting for the result in a PU (*start* and *waiting* states). If the PU has the result, it is read via channel *out*.

For the performance checking, we are interested in the throughput of the system which can be calculated from the size of an incoming packet in bits (*size*), the

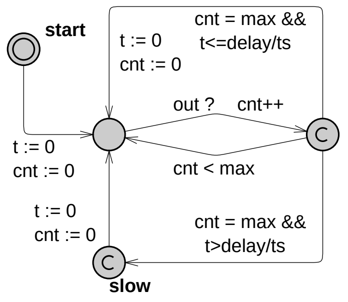

Figure 9: Model of the multiplexer

average delay of rule matching process in time slots (*delay*) and time of a time slot in seconds (*ts*): *throughput = size/(delay*ts)*. We obtain the average delay from the ratio *(delay for x results)/x*. To compute delay for *x* results we can use the counter of outgoing results and a clock. When counter reaches the maximum (user chosen value e.g. 1000), system fires the transition to the *slow* state (the observer) only if the clock value is greater than allowed delay. The performance checking of the throughput is based on manual finding of the minimum value of delay i.e. model checking running several times to find appropriate delay value where system does not reach the *slow* state.

The Figure 10 shows the dependence between the throughput, calculated as average number of requests, and real incomming requests. An angle of a dashed line expresses the throughput calculated from the delay of wave of the results (*wave* is user chosen value of number of results).

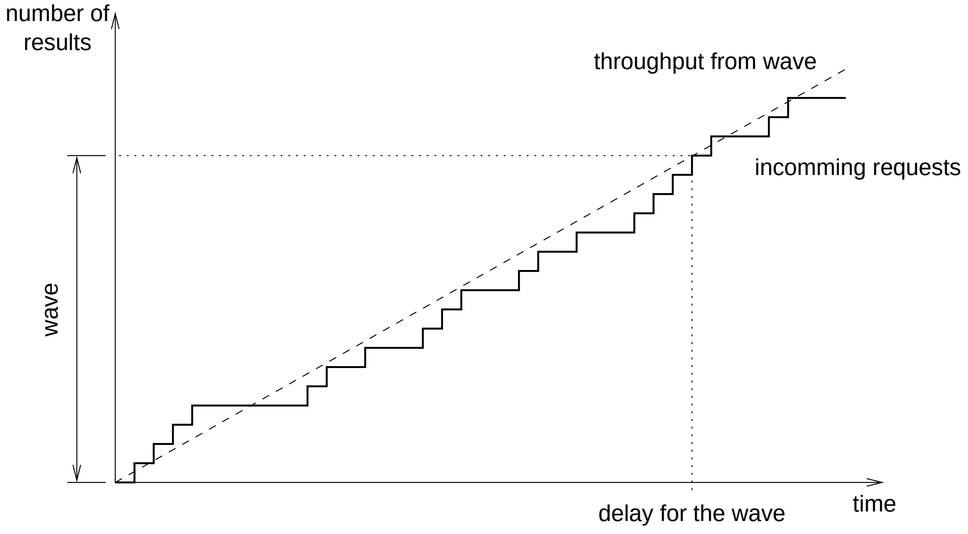

Figure 10: Wave of outgoing results

5.3 Modelling Environment

Environment of the system has to be added to the model in order to check how the system responds to the environment. In systems with queues like our model of packet analyser is, we have to model an input - a generator of incoming requests (packets) and an output - a receiver of the result.

We are not able to successfully model and verify the model with real generator because of expressivity of modelling language and state explosion. We define following properties of the generator:

- distribution of incoming requests - random, uniform, given by a specific function
- time constraints on new requests - time span between two consequent requests, minimum and maximum delay between requests
- request properties - a size (minimal, maximal, average), a kind
- extreme conditions - flooding of requests, losses on the input

We are not aware of a verification tool that is able to express and successfully verify above written condition completely. However, we can create a set of generators such that every generator models only one of these properties and then we can verify model with every item of this set. This kind of abstraction provides one view on the system for a specific condition. Then, if the property is satisfied for the given environment (a generator) we can change/enriched environment (change kind of generator) and verify the property under different input/output conditions.

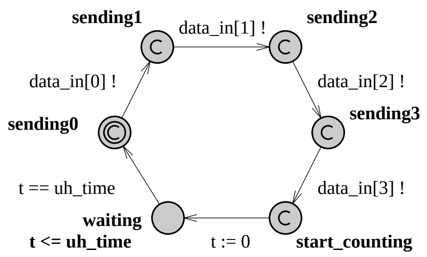

Figure 11: A model of generator

In Scampi project we created a simple model of input packets as Figure 11 shows. This model generates incoming packets for every input buffer under extreme conditions, that is, a new packet is generated immediately when the previous packet was accepted. This is described by sequence of states *sending0* to *sending3* when a new packet is generated to every queue. We use so called committed states (with 'c' mark) where no time progress is possible. That means in one moment (no time progress between) four transitions are passed and four new packets are generated in every input buffer. This behaviour performs an attack from the network (flooding), and we probe how the system responds to a such condition.

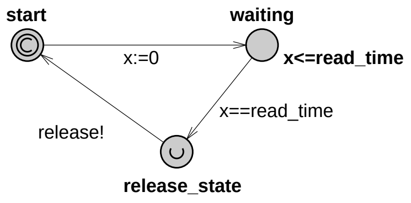

Figure 12: A model of a receiver

Similar analysis can be applied on the model of a receiver. In our case of studying behaviour of simple FIFO queue we created a model of the receiver that reads data whenever they are available, see Figure 12.

The receiver has three states - *start*, *waiting* and *release_state*. It models a process that reads data every *read_time*. After reaching *read_time* the control moves to *release_state* where the process stays until a transition is allowed. This state is urgent - defined by letter 'u' - that guarantees the outgoing transition is taken immediately after a corresponding guard is satisfied. In comparison to committed state 'c' time progress is allowed there.

In case of delayed FIFO queues it is important to mention that the receiver should not put time constraints on transitions before data are read. The receiver should read data from delayed FIFO queue whenever they are available. Otherwise, deadlock will appear.

6 Verified Properties

6.1 Throughput Verification

This section discusses verified properties of the model obtained by model checker Uppaal [Uppaal00] on machine P4-2.8 GHz with 512 MB. Properties are written in form of CTL invariance formulas - remember *A* means over all paths, *G* means globally (always).

- *A G ˜ deadlock* - model checking time was 10.8 s. There is never deadlock in the system, which means that the system will always work. Model satisfies this property.

- *A G ˜ (UHFIFO0.throwing or UHFIFO1.throwing or UHFIFO2.throwing or UHFIFO3.throwing)* - model checking time was < 0.1 s. This property shows us whether the unified header is ever thrown away from the UHFIFO. Model not satisfies it which means that the input stream is faster than the output stream (this is the first result of the performance checking).

- *A G ˜ (RFIFO0.throwing or RFIFO1.throwing or RFIFO2.throwing or RFIFO3.throwing)* - model checking time 2.3 s. This property is similar to the one above. We use this to get needed minimum size of RFIFO. This property still holds even if the size is set to only one item. It is obvious since the Processing Unit has the same speed as TCAM.

- *A G ˜ MX.slow* - model checking time 2.2 s. This property expresses the throughput checking that the *wave* of the results from the multiplexer is sent at time corresponding to the average delay for one result. This property was satisfied for the constant *wave* set to 1024 and the constant *results* set at least to 16 time slots (which is 320ns at 50MHz). Today, system design supports IP packets with size between 64B and 2kB. If the input for the whole system is the stream of the smallest packets (64B), the throughput caused by Lookup processor can be theoretically 1.6Gbps at most *(64*8/320 ns)*.

6.2 Parametric Verification of a FIFO Queue

Parametric model of the FIFO queue has three processes - a generator, a receiver and an simple FIFO queue. We considered three parameters: the maximal length of the queue *FIFOsize*, the rate of incoming request *uh_time* and the rate of reading data from the queue *read_time*. We used the model of simple FIFO queue (see Figure 4) with an initial constraint *FIFOsize > 0*. Then, we obtained by TReX following results:

- If *uh_time >= read_time* there is not *throwing* state in the configuration space. The queue never overflows and the length of the buffer is not important. Verification takes only few seconds because there are only five symbolic configuration states.

- For *uh_item < read_time* analysis did not finished. So we decided to set a fixed size of the queue *FIFOsize = 3* and then we tested whether the buffer overflows. It happened for *4 * read_time = uh_time*. In this case we know the length of the queue and are interested in time constraints at *throwing*. In other case we set a specific input condition (the rate of incoming requests) and we verify for what kind of time constraints the queue of given length will be overfilled.

Parametric verification of the FIFO queue is undecidable problem [AHV93], however for specific initial constraints we can obtain some results. These results help us to find a configuration of the system where the verified property is not satisfied.

7 Conclusion

In this paper we introduced our experience with verification of the behaviour of the Lookup processor and its close neighbourhood of the Scampi project. We created models of several components of the system and analysed its behaviour. We were especially focused on the throughput of the system and a risk of buffers overflow. At first we did a manual analysis of the system, then we used model checkers Uppaal and TReX. Using Uppaal we were able to prove correctness of the systems - deadlock detection, buffer overflow for specific time conditions, maximal latence of the system etc. Using parametric verification by TReX we observed constraints on the buffer length, the rate of incoming packets and the rate of reading a buffer. We detected several states where throwing appeared and found time relations for these states.

We noticed a typical structure of the hardware design and defined three kinds of basic components that are present nearly at every hardware design - FIFO queues, executive components and environment of the system. We proposed patters for modelling and analysing these components and discussed their behaviour.

However, there are still many challenges for the future research. We noticed that system with global clocks and synchronized events can be easily modelled and analysed even in untimed model checkers. This makes possible to joint untimed verification of some parts of the system and timed verification of important events. Another challenge in the Scampi project is to set a common framework for analysis on the level of the source code and high-level abstract analysis. At the moment we did this as independent tasks.

References

[AHV93] R. Alur, T.A. Henzinger, and M.Y. Vardi. Parametric Real-time Reasoning. In *ACM Symposium on Theory of Computing*, pages 592-601, 1993.

[Alur99] R. Alur. Timed Automata. In *Proceedings of 11th CAV*, volume 1633 of *LNCS*, pages 8-22, 1999.

[LUP] D. Antoš, P. Zemčík, and J. Kořenek. Lookups in FPGA hardware accelerator for IPv6 routers. *CESNET Technical Report 2002.*

[TREX01] A. Bouajjani, A. Collomb-Annichini, and M. Sighireanu. TREX: A tool for reachability analysis of complex systems. In *Proceedings of CAV*, volume 2102 of *LNCS*, pages 368-372, Springer Verlag, 2001.

[TR-4-2004] J. Holeček, T. Kratochvíla, V. Řehák, D. Šafránek, and P. Šimeček. How to formalize FPGA hardware design. *CESNET Technical Report 4/2004.*

[Router] J. Novotný, O. Fučík, and D. Antoš. Project of IPv6 router with FPGA hardware accelerator. In Cheung P.Y.K., Constantinides G.A., and de Souza J.T., editors *Field-Programmable Logic and Applications, 13th International Conference FPL 2003*, pages 964-967, Berlin-Heidelberg, Springer, 2003.

[Uppaal00] P. Pettersson and K.G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40-44, February 2000.