UPPAAL in Practice: Quantitative Verification of a RapidIO Network*

Jiansheng Xing[1,2], Bart D. Theelen[2], Rom Langerak[1], Jaco van de Pol[1], Jan Tretmans[2], and J.P.M. Voeten[2,3]

[1] University of Twente, Faculty of EEMCS
P.O. Box 217, Formal Methods and Tools
7500 AE Enschede, The Netherlands
{xingj,r.langerak,j.c.vandepol}@cs.utwente.nl
[2] Embedded Systems Institute
P.O. Box 513
5600 MB Eindhoven, The Netherlands
{bart.theelen,jan.tretmans}@esi.nl
[3] Eindhoven University of Technology, Faculty of Electrical Engineering
Information and Communication Systems group
5600 MB Eindhoven, The Netherlands
{j.p.m.voeten}@tue.nl

Abstract. Packet switched networks are widely used for interconnecting distributed computing platforms. RapidIO (Rapid Input/Output) is an industry standard for packet switched networks to interconnect multiple processor boards. Key performance metrics for these platforms include average-case and worst-case packet transfer latencies. We focus on verifying such quantitative properties for a RapidIO based multiprocessor platform that executes a motion control application. A performance model is available in the Parallel Object-Oriented Specification Language (POOSL) that allows for simulation based estimation results. It is however required to determine the exact worst-case latency as the application is time-critical. A model checking approach has been proposed in our previous work which transforms the POOSL model into an UPPAAL model. However, such an approach only works for a fairly small system. We extend the transformation approach with various heuristics to reduce the underlying state space, thereby providing an effective approximation approach that scales to industrial problems of a reasonable complexity.

Keywords: UPPAAL; POOSL; transformation; quantitative verification; heuristic

1 Introduction

A packet switched network is a digital communication network that groups all transmitted data into suitably-sized blocks, called packets. The network over

* This work has been supported by the EU FP7 under grant number ICT-214755: Quasimodo.

which packets are transmitted is a shared network which routes each packet independently from others and allocates transmission resources as needed. The principal goals of packet switching are to optimize utilization of available link capacity, minimize response times and increase the robustness of communication. When traversing network adapters, switches and other network nodes, packets are buffered, resulting in variable delay and throughput, depending on the traffic load in the network.

RapidIO is an industry standard [9] for packet-switched networks to connect chips on a circuit board, and also circuit boards to each other using a backplane. It has found widespread adoption in the following applications: wireless base station, video, medical imaging, etc. Key performance metrics for these applications include average-case and worst-case packet transfer latencies.

In this paper, we consider a motion control system that includes a packet switched network conforming to the RapidIO standard. The motion control system is characterized by feedback/feedforward control strategies and periodic execution. It is constrained by strict timing requirements that all packets must arrive at their destinations before the period ends. The considered motion control algorithms are distributed over a multi-processor platform. Various processors in this platform are inter-connected by a RapidIO network. The main challenge for system designers is how to map the motion control algorithms on the multi-processor platform such that the timing constraints are met. Packet transfer latencies as worst-case latencies and average-case latencies are essential criteria for finding a feasible mapping.

A simulation based approach is available that relies on the Parallel Object-Oriented Specification Language (POOSL) for investigating the performance of the motion control system. First, a POOSL model is constructed for a given mapping and then end-to-end packet transfer latencies are analyzed. These steps are repeated for alternative mappings, until a feasible mapping has been found, whose end-to-end latencies satisfy the timing constraints. POOSL analysis gives both average-case and worst-case latencies. The obtained latencies are estimation results since the POOSL analysis is based on simulation. However, as the motion control application is time-critical, worst-case latencies are strict timing constraints. Exact worst-case latencies are therefore demanded.

In this paper, we focus on verifying worst-case packet transfer latencies for a realistic motion control system. In earlier work [11], we have shown that transforming a POOSL model into an UPPAAL model is feasible. Based on the obtained UPPAAL model, we also showed that quantitative verification for worst-case latencies is only feasible for a fairly small system. We extend such an approach to enable analyzing an industrial-sized problem in this paper. First the POOSL model of a realistic motion control system is illustrated and transformed into an UPPAAL model according to the transformation patterns in [11]. Second scalability experiments show that the UPPAAL model is not capable for handling realistic (high volume) traffics. Then we propose some heuristics for the UPPAAL approach. Experiments show that the UPPAAL approach with heuristics can find worse scenarios (worse latencies) than the POOSL approach

and thus is an effective approximation method which complements the POOSL approach for performance analysis.

The rest of the paper is organized as follows. Section 2 presents the POOSL model for a realistic motion control system. The transformation from the POOSL model into an UPPAAL model is discussed in Section 3. We present the scalability of the UPPAAL model and then propose some heuristics for worst-case latency analysis in Section 4. Finally, conclusions are drawn in Section 5.

2 POOSL Model of a Realistic Motion Control System

POOSL was originally defined in [7] as an object-oriented extension of CCS [6]. Meanwhile, POOSL has been extended with time in [4] and probabilities in [3] to become a very expressive formal modeling language accompanied with simulation, analysis and synthesis techniques that scale to large industrial design problems [10].

Three types of objects are supported in POOSL: data, process, and cluster. Data models the passive components of a system representing information that is generated, communicated, processed and consumed by active components. The elementary active components are modeled by processes while groups of active components are modeled by clusters in a hierarchical fashion.

Figure 1 depicts the top-level POOSL model for a realistic motion control system. The left part is for channel (refers to a specific type of packet with specified size, source, and destination) generation. The right part represents the underlying RapidIO network of the motion control system. These two parts are connected by message *si*. The left part is implemented as a process class *RIOChannelTrafficGenerator*. More details of this process class are shown in figure 2.

The right part of figure 1 is implemented as a cluster class *RIOResource*. We note that message *si* is multiplexed into 5 messages *slot1,...,slot5* which connect *RIOChannelTrafficGenerator* with different subparts of cluster *RIOResource*. More details of *RIOResource* are shown in figure 3. *RIOResource* is mainly constituted by 5 subparts *CB1,CB2,CB3,CB4*, and *CB5*. Each subpart is an instance of the cluster class *RIOCarrierBlade*.

RIOCarrierBlade represents a pair of packet switches and their connected endpoints. Each switch connects with 4 endpoints directly. The details of *RIOCarrierBlade* are shown in figure 4. *B1_EP1,B2_EP1*,..., etc., each represents an endpoint which is implemented as a process class *RIOEndPoint*. More details of this process class are shown in figure 5.

In figure 4, *SW1* and *SW2* represent two packet switches. A packet switch is implemented as a process class *RIOSwitch*. More details of this process class are shown in figure 6.

The system works as follows: first the channel (each channel refers to a specific kind of packet, which will be divided into several equal-sized packets during generation, see next section for details) generation part (left part of figure 1) generates channels (actually packets); the generated packets are actually put into

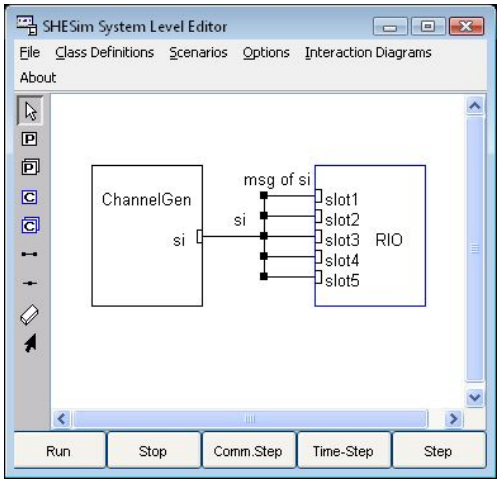

Fig. 1. POOSL model for a realistic motion control system

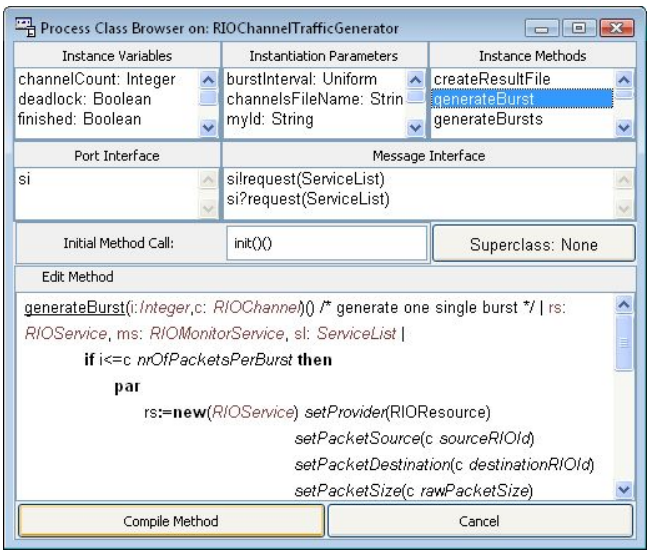

Fig. 2. Process Class RIOChannelTrafficGenerator

corresponding endpoints. Each endpoint represents a processor that is actually sending and/or receiving packets. The generated packets then are sent from the endpoints and routed among the whole RapidIO network until they are received by their corresponding target endpoints. We notice that the whole underlying RapidIO network includes 5 subparts; each subpart contains 2 packet switches and 8 endpoints.

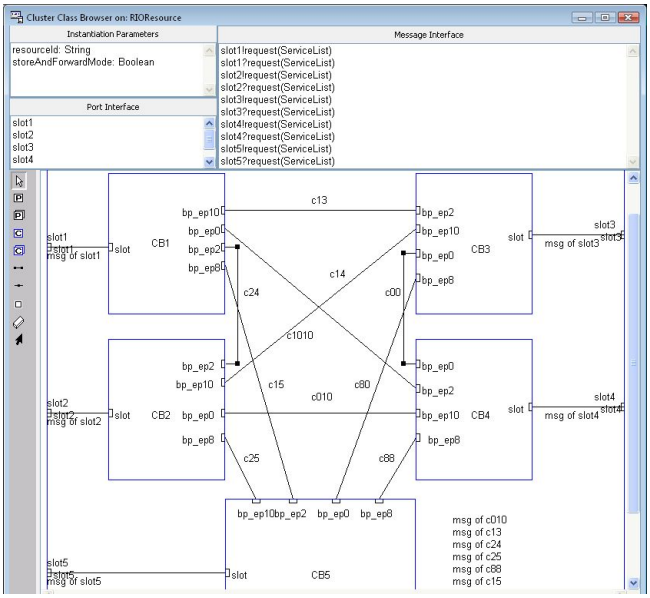

Fig. 3. Cluster Class RIOResource

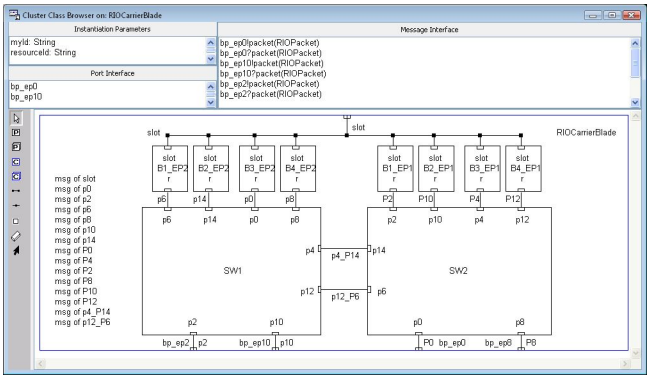

Fig. 4. Cluster Class RIOCarrierBlade

3 Transformation from POOSL to UPPAAL

UPPAAL is a tool for modeling, validation and verification of real-time systems. It is based on the theory of timed automata (TA) [1] and its modeling language offers additional features such as bounded integer variables and urgency [2]. The query language of UPPAAL, used to specify properties to be checked, is a subset of CTL (computation tree logic) [8, 5].

A timed automaton is a finite-state machine extended with clock variables. It uses a dense-time model where a clock variable evaluates to a real number. All

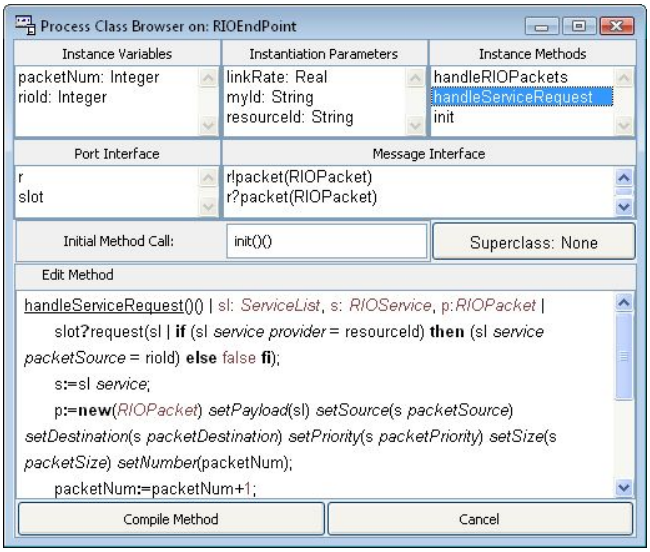

Fig. 5. Process Class RIOEndPoint

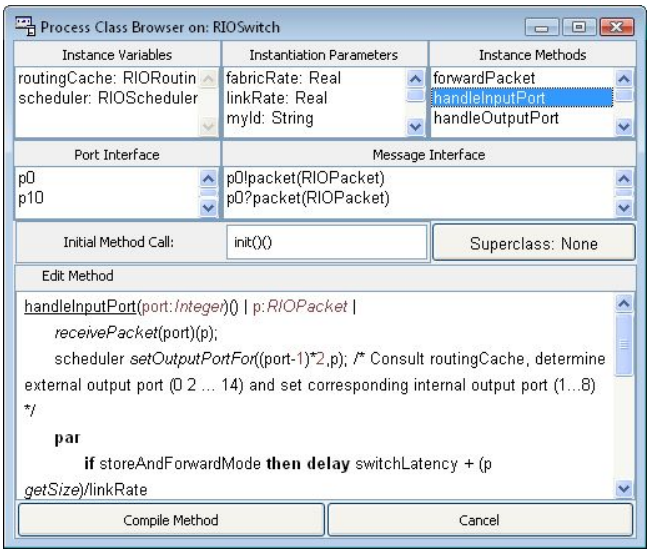

Fig. 6. Process Class RIOSwitch

the clocks progress synchronously. A system is modeled as a network of several such timed automata in parallel. The state of a system is defined by the locations of all automata, the clock constraints, and the values of the discrete variables. Every automaton may fire an edge (sometimes called a transition) separately or

synchronize with another automaton, leading to a new state. We refer the reader to [2] for a more thorough description of the timed automata used in UPPAAL.

Several general transformation patterns from POOSL to UPPAAL have been characterized in [11]. In this section, we will mainly follow such patterns for the transformation. However, we note that more abstractions and techniques (such as timed automation template) have been used to simplify the transformation as well as the model.

3.1 Data part transformation

The POOSL model in figure 1 includes various data classes. *RIOPacket*, *RIOChannel*, *RIOQueue*, *RoutingCache* and *RIOScheduler* are the most important ones. We only explain the transformation for *RIOChannel* as it is a newly introduced data class. The transformation for other data classes has been introduced in [11] and is omitted here.

RIOChannel *RIOChannel* is a data structure that refers to a specific kind of packet which has a specified size, source, and destination. Realistic traffic scenarios can be easily expressed by grouping different kinds of channels. This data class is transformed into an UPPAAL struct *channel*. Data methods for accessing its member variables come for free in the UPPAAL model. Other data methods such as *nrOfPacketsPerBurst* (a channel is divided into several predefined equal-sized packets, *nrOfPacketsPerBurst* refers to the number of such packets) are also easily transformed into the UPPAAL model.

3.2 Process part transformation

Three main improvements have been made for the process part transformation. First, timed automaton templates are used to abstract typical activities such that similar activities can be modeled by instantiating a timed automaton template with different parameters. Second, transfer activity is abstracted into a non-time-consuming activity, which greatly simplifies the model. Third, the endings of concurrent timed automata are synchronized as they consume the same time (all packets have the same size), which also simplifies the model.

From the POOSL model point of view, each switch contains at most 24 concurrent activities: 8 input activities (one for each input port that is actually used), 8 transfer and 8 output activities (one for each output port that is actually used). Each endpoint includes a sending and a receiving activity, depending on whether the endpoint is actually used to send/receive packets into/from the network. Besides, a traffic generation activity, a traffic sending activity and a traffic monitor activity exist in the process class *RIOChannelTrafficGenerator*. As there are 10 switches and 40 endpoints in the POOSL model, the maximum number of concurrent activities is 323 (24*10+40*2+3) for the POOSL model.

However, we will follow a new point of view to characterize such activities in the UPPAAL model. Three types of concurrent activities have been characterized:

1. Packet transfer from endpoint to switch which combines the sending activity of an endpoint and the input activity of a switch; 2. Packet transfer from switch to switch which combines the output activity of a switch and the input activity of a switch; 3. Packet transfer from switch to endpoint which combines the output activity of a switch and the receiving activity of an endpoint. Each endpoint has its own traffic sending activity. Traffic generation activity and transfer activity within a switch are abstracted into non-time-consuming activities. Traffic monitor activity is abstracted into the worst-case latency verification problem of the obtained UPPAAL model.

Channel traffic generation activity The POOSL model splits generation of channel traffic from actually sending them. Method *generateBurst* (POOSL code is shown at the bottom part of figure 2) includes most of the details for such activity. First, the POOSL model reads channel traffic information from a file *channels.txt* and creates sending queues for each involved endpoint. Then channel traffic will be generated in a uniformly random order according to the channel traffic information and stored in corresponding sending queues. Later, the sending queues will be used by corresponding endpoints for actually sending packets.

Such activity is transformed into an UPPAAL function *handleRIOChannel* where only a specific ordering (refers to the sequence of channels in sending queues) is considered (the ordering is embedded in the UPPAAL code). Sending queues are implemented as arrays, additional functions are also provided such that a FIFO accessing policy is enforced. The timed automaton which includes this activity is shown in figure 7.

Fig. 7. Channel Traffic Generation

If more orderings for channel traffic generation are considered, traffic generation for each channel is implemented as a timed automaton template shown in figure 8. When the UPPAAL model starts, all such timed automata will fire the edge from the initial location and execute the update *channelGen(i)* to generate the traffic for the channel indexed by *i*. The nondeterminism among these timed automata models the uniformly random generation of channel traffic in POOSL model.

Packet Transfer from Endpoint to Switch We illustrate transforming the sending of packets by an endpoint and the input handling of packets received by a switch, which connects the output of an endpoint to an input port of a switch with a message. For example, message *P2* connects the output of endpoint

Fig. 8. Channel Generation

B1_EP1 to the input port *p2* of switch *SW2* as in figure 4. The sending of packets for a *RIOEndPoint* is specified by method *handleServiceRequest* shown in figure 5. The input handler for a *RIOSwitch* is specified by method *HandleInputPort()* shown in figure 6.

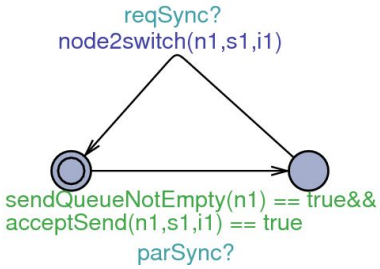

Fig. 9. Packet Transfer from Endpoint to Switch

Such activity is transformed into a timed automaton template shown in figure 9. Three parameters are provided to specify the endpoint, the switch and its input port. When broadcast synchronization signal *parSync* arrives, and the condition *sendQueueNotEmpty(n1) == true && acceptSend(n1,s1,i1) == true* is satisfied, the timed automaton will move to the second location. The condition checks if endpoint *n1* is not empty and the input queue *i1* of switch *s1* can accept the current packet of endpoint *n1*. The timed automaton will stay in the second location for the time it takes to transmit the packet over the link. In our approach, the time consumed here is abstracted into the synchronization timed automaton (see figure 10) as all concurrent activities consume the same time (all packets actually transferred are equal-sized as mentioned earlier). The timed automaton will move the current packet of endpoint *n1* to the input queue *i1* of switch *s1* by function *node2switch(n1,s1,i1)* when it receives the signal *reqSync* and then return to its initial location.

The other two kinds of concurrent activities can be transformed in a similar way. We omit the details due to space limitation.

Transfer activity The transfer activity refers to the actual scheduling of packets from input queues to a specific output port (namely output arbitration). This behavior is executed for each output port of a *RIOSwitch* and is specified by method *scheduleForOutput* in the POOSL model.

According to the configuration of the POOSL model, the transfer activity starts when the head of a packet has arrived in the input queue. Then the head is immediately forwarded to the output queue before the packet has completely arrived in the output queue. The reason is: the switch fabric rate is always larger than the link rate and therefore the packet is forwarded in "cut-through "mode. As transfer activity seamlessly connects an input queue to an output queue as if time is not consumed for this activity, we abstract it into a non-time-consuming activity which is implemented as an update as shown in figure 10.

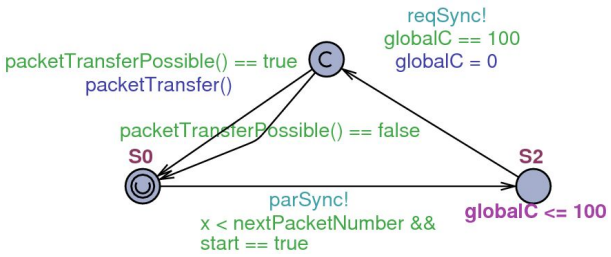

Fig. 10. Synchronization

Synchronization To enforce that only selected timed automata are fired concurrently and are synchronized with other timed automata, we use the following timed automaton shown in figure 10. When the condition $x <$ *nextPacketNumber* && *start* == *true* is satisfied, a broadcast synchronization signal *parSync* will be raised. Then the timed automaton will move to the second location. x denotes the number of received packets and *nextPacketNumber* denotes the number of generated packets. The condition checks if there still exist packets that have not been received and all channel traffic generations have finished. The timed automaton stays at the second location for a constant length of time (as discussed above). Then it raises a *reqSync* signal to end all concurrent activities and then return to its initial location. It will handle the transfer activity *packetTransfer* if the condition *packetTransferPossible() == true* is satisfied during the return. Or else, it will return to the initial location directly.

Worst-case Latency Verification The traffic monitor activity is transformed into the worst-case latency verification problem in the UPPAAL model. As we only consider the scenarios that all channels are generated in a burst, the worst-case latency is defined as the time when the whole burst has been handled (all packets have been received). We constructed the timed automaton in figure 11 to verify the worst-case latency. When condition $x ==$ *nextPacketNumber* && $x > 0$ is satisfied, the timed automaton will move to location *S1*. A self-loop is added here to distinguish the deadlock from end of operation. The condition

checks if all packets have been received. The *endSync* signal is defined as an urgent channel to enforce that the transition will be taken immediately when the condition is satisfied. The global clock *globalT* perfectly describes the worst-case latency for the burst. Assume W is the worst-case latency, we just need to check if the property $A[]\ globalT \leq W$ is satisfied. If this property is not satisfied, we can increase W step by step until this property is satisfied. In other words, the smallest upper bound for the worst-case latency can be found iteratively.

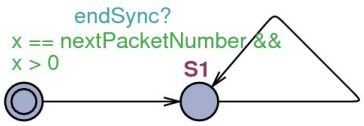

Fig. 11. Worst-case Latency Monitor

System Declaration Based on the building blocks obtained above, we can instantiate timed automata templates into concrete timed automata according to the system configuration. In the UPPAAL model, a concrete timed automaton will be instantiated for each concurrent activity. Each endpoint is assigned an id according to the file *RIOIds.txt*, whereas the id for a switch is actually implemented as an index of an array (the total 10 switches are grouped into an array). For example, *n14s1i5 = Node2Switch(14,1,5)* denotes the instantiation of a concurrent activity referring to the packet transfer from endpoint 14 to switch 1 at input port 5. In total about 80 concurrent activities are instantiated in the UPPAAL model. Compared with more than 300 concurrent activities in the POOSL model, the UPPAAL approach is considerably simplified.

4 Heuristics

In this section, we first present the scalability of the UPPAAL model. Then we propose some heuristics to reduce the underlying state space such that realistic application scenarios can be analyzed.

4.1 Scalability of the UPPAAL model

Different orderings for channel traffic generation result in different sending orders for the packets in endpoints which are also referred to as traffic scenarios. The difficulty of the state space exploration for the RapidIO network worst-case latency analysis comes from this combinatorial problem. Experiments have been run to show the scalability of the UPPAAL model and the results are listed in table 1. The second row refers to number of packets included in the corresponding channels. The channel information is selected from the file *channels.txt*. If n

channel generation timed automata are declared in the model and the worst-case latency can be verified, we say that the UPPAAL model support $n!$ orderings (verification time is not constrained here, JVM memory is set to the maximum value (1500M) supported in our windows vista system).

Table 1. Scalability of the UPPAAL model

Number of Channels	10	20	30	40
Number of Packets	(129)	(300)	(372)	(459)
Supported orderings	6!	4!	4!	3!
Whole orderings	10!	20!	30!	40!

From table 1, the following conclusions can be drawn: 1. With the increase of the number of channels (as well as packets), state space explosion occurs. 2. The UPPAAL model can only support investigating a small part of the state space for high volume traffics. Further, the heavier the traffic, the smaller part of the state space it investigates. Table 1 actually shows why the verification for worst-case latency is so difficult.

4.2 Heuristics

From the above analysis for the scalability of the UPPAAL model, we see that exact verification is not possible for high volume traffics. We then turn to heuristic methods and anticipate that worse packet transfer latencies can be found (than the POOSL approach). In this section, we will introduce some heuristics for state space reduction of the UPPAAL model such that high volume traffic scenarios can be analyzed.

Before we go into the details of the heuristics, we assume that: 1. The routing information and the output arbitration are fair. 2. Channel traffic is uniformly distributed in the endpoints and switches involved. 3. Traffic is never overloaded such that only a few collisions occur.

Heuristics H1 Each endpoint has its own sending queue. The sending of an endpoint is independent from others unless collisions occur. As collisions are few compared with the whole traffic load according to the assumption, the sending of each endpoint can be seen as independent most of the time. The orderings among different endpoints are not relevant for the worst-case latency and will not be considered. We can thus focus on the orderings within each endpoint. Besides, when two or more channels within an endpoint have the same type, the orderings among these channels are also irrelevant and will not be considered (as they are symmetric). We combine these two heuristics and refer it as H1. Table 2 shows the result obtained by applying heuristic H1. Compared with table 1, we can see that: the state space obtained by using the heuristic H1 is still too large to be analyzed.

Table 2. Heuristic H1

Number of Channels	10	20	30	40
Number of Packets	(129)	(300)	(372)	(459)
Reduced orderings with H1	5!	5!5!4!2!	8!6!5!5!2!2!	9!6!6!5!5!3!2!

Heuristic H2 Among all the activities operated in the RapidIO network, collisions only occur when two or more input queues compete for one output port (queue) within a switch. Based on the above assumptions, the following observations hold. 1. The longest sending queue (endpoint) is most probably to be the last one finished. 2. The more collisions within a sending queue, the worse scenario it is. A heuristic easily follows: generating more collisions for the longest sending queue. However, it would be too complicated to implement as the detailed routing information must be investigated.

We then come up with another idea: generating more collisions for all endpoints (the whole system). It is obvious that: the more switches a packet goes through, the more likely collisions occur for this packet. We can thus put such packets in the front end of all endpoints to generate more collisions. The number of switches a channel goes through is reflected in the latency listed in table 3 (packet size is assumed to be 100 bytes). These latencies are obtained by simulation (both POOSL and UPPAAL simulator can do) for each channel (packet) type. It is obvious that: the larger the latency, the more switches it goes through. In table 3, the first row and first column both denote the id of endpoints. For example, the element 5 in the second row and fifth column means the latency from source endpoint 2 to destination endpoint is 5. Due to the space limitation, only part of the possible packet types is listed (there are 40 endpoints in the system).

Table 3. Latency for Channel (Packet) Types)

	2	3	4	5	6	7	8	9
2	0	2	2	5	5	5	4	4
3	2	0	2	5	5	5	4	4
4	2	3	0	5	5	5	4	4
5	5	5	5	0	2	2	3	3
6	5	5	5	2	0	2	3	3
7	5	5	5	2	2	0	3	3
8	4	4	4	3	3	3	0	2
9	4	4	4	3	3	3	3	0

We have developed a java application program to sort all channels in each endpoint such that channels are put in sending queues in descending order (the larger latency of the channel type is, the more forward this channel is in the

sending queue of the endpoint) according to the table. As we have mentioned earlier, only a few orderings can be considered. We thus only investigate several different orderings for front channels in the largest sending queue.

Experiment Results Experiments have been run with the use of the above heuristics H1 and H2 in the UPPAAL model. POOSL results are also provided for comparison. The first row in table 4 refers to the number of channels. The results are the worst-case packet transfer latency obtained by two approaches.

Table 4. Experiment Results: UPPAAL Heuristic vs. POOSL

Number of Channels	20	40	60	80	100	120	140	160
POOSL Result	128	227	298	350	504	501	504	515
UPPAAL Heuristic Result	129	220	322	371	538	539	539	522

Both heuristics H1 and H2 are applied in the experiment. From table 4, we can see that: for low volume traffic scenarios, the UPPAAL approach using heuristics cannot guarantee to find worse latencies. This phenomenon comes from the fact that: POOSL's simulation engine is very effective and can explore a large part of the state space for low volume traffic scenarios. Whereas, even for low volume traffic scenarios UPPAAL can only explore a small part of the state space (see table 1). However, for high volume traffic scenarios, with the progress of state space explosion, the superiority of the high-speed engine for POOSL disappears. The UPPAAL approach using heuristics can always find worse scenarios than with the POOSL approach. The UPPAAL approach using heuristics is an efficient approach to complement the POOSL approach for finding approximate worst-case latencies.

5 Conclusions and Future Work

The exact worst-case packet transfer latency is an important metric for motion control applications that run on multiple processors interconnected by a RapidIO network. We have proposed a model checking approach using UPPAAL for this problem [11]. However, such an approach only applies to small scale models. In this paper, we extend such an approach and apply it to a realistic application scenario. First, we transform the POOSL model of a realistic motion control system into an UPPAAL model. Then we show that the application of the UPPAAL approach for exact worst-case packet transfer latency verification is limited to low volume traffics. We propose to use heuristics for high volume traffics. Although only approximate results can be obtained, the heuristics is still valuable as worse scenarios can be found than POOSL approach. Experiments show that the UPPAAL approach with heuristics is effective to complement the POOSL approach for finding approximate worst-case latencies.

In this paper, we only apply the heuristics to the UPPAAL approach. It is somewhat unfair for the comparison of the two approaches. We will apply the heuristics to POOSL approach and compare their performance in a fairly fashion. Besides, the RapidIO network (includes 323 concurrent activities) discussed in our paper is only a small part of the POOSL model of the total system, which includes an estimated 2500 processes that all include between 1 and about 10 concurrent activities. Further abstraction techniques are still needed to scale up to such real industrial sized problems. Future work also includes transforming POOSL into UPPAAL at a more semantical level by means of additional transformation patterns[11].

References

1. Alur, R., Dill, D.: Automata for modeling real-time systems. In: Proceedings of the seventeenth international colloquium on Automata, languages and programming. pp. 322–335. Springer-Verlag New York, Inc., New York, NY, USA (1990)
2. Behrmann, G., David, R., Larsen, K.G.: A Tutorial on UPPAAL. In: LNCS, Formal Methods for the Design of Real-Time Systems (revised lectures). pp. 200–236. Springer (2004)
3. van Bokhoven, L.: Constructive Tool Design for Formal Languages: From Semantics to Executing Models. Ph.D. thesis, Eindhoven University of Technology (2002)
4. Geilen, M.: Formal Techniques for Verification of Complex Real-Time Systems. Ph.D. thesis, Eindhoven University of Technology (2002)
5. Logothetis, G., Schneider, K.: Symbolic model checking of real-time systems. International Syposium on Temporal Representation and Reasoning 0, 0214 (2001)
6. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
7. van der Putten, P., Voeten, J.: Specification of Reactive Hardware/Software Systems. Ph.D. thesis, Eindhoven University of Technology (1997)
8. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in Cesar. In: Proceedings of the 5th Colloquium on International Symposium on Programming. pp. 337–351. Springer-Verlag, London, UK (1982)
9. Shippen, G.: A technical overview of RapidIO. `http://www.eetasia.com/ART_8800487921_499491_NP_7644b706.HTM` (Nov 2007)
10. Theelen, B.D., Florescu, O., Geilen, M., Huang, J., van der Putten, P., Voeten, J.: Software/hardware engineering with the Parallel Object-Oriented Specification Language. In: MEMOCODE '07: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign. pp. 139–148. IEEE Computer Society, Washington, DC, USA (2007)
11. Xing, J., Theelen, B.D., Langerak, R., van de Pol, J., Tretmans, J., Voeten, J.: From POOSL to UPPAAL: Transformation and quantitative analysis. In: Proceedings of the International Conference on Application of Concurrency to System Design. pp. 47–56. IEEE Computer Society, Los Alamitos, CA, USA (2010)