10th International Conference on Application of Concurrency to System Design

From POOSL to UPPAAL: Transformation and Quantitative Analysis

Jiansheng Xing*†,B.D. Theelen†,Rom Langerak*,Jaco van de Pol*,Jan Tretmans†,J.P.M. Voeten†‡
*University of Twente, Faculty of EEMCS
7500 AE Enschede, The Netherlands
Email: {xingj,r.langerak,j.c.vandepol}@cs.utwente.nl
†Embedded Systems Institute
‡Eindhoven University of Technology, Department of Electrical Engineering
5600 MB Eindhoven, The Netherlands
Email: {bart.theelen,jan.tretmans}@esi.nl, j.p.m.voeten@tue.nl

***Abstract*—POOSL (Parallel Object-Oriented Specification Language) is a powerful general purpose system-level modeling language. In research on design space exploration of motion control systems, POOSL has been used to construct models for performance analysis. The considered motion control algorithms are characterized by periodic execution. They are executed by multiple processors, which are interconnected by Rapid Input/Output (RapidIO) packet switches. Packet latencies as worst-case latencies and average-case latencies are essential performance criteria for motion control systems. However, POOSL analysis merely allows for estimation results for these latency metrics since it is primarily based on simulation. Because motion control systems are time-critical and safety-critical, worst-case latencies of packets are strict timing constraints. Therefore exact worst-case latencies are to be determined. Motivated by this requirement we propose to use model checking techniques. In this paper we illustrate how a POOSL model of a (simplified) motion control system can be transformed into an UPPAAL model and we verify its functional behavior and worst-case latencies. Moreover, we show that analysis of average-case latencies can also be accomplished with assistance of the model checking tool UPPAAL.** [1]

***Index Terms*—POOSL; UPPAAL; transformation; performance; verification; quantitative analysis;**

I. Introduction

Designing software/hardware systems requires dealing with their increasing complexity within ever-shortening design times. Usually, the design process involves considering alternative ideas and options for realizing the required functionality. In the early phases of the design process, the choice for a specific alternative may have a deep impact on the functionality and performance of the final implementation. System-level design methodologies can be applied to assist the designer in taking well-founded design decisions. POOSL (Parallel Object-Oriented Specification Language) [1] is a system-level modeling language that supports such methodologies by constructing and/or refining models. Based on the developed models, POOSL supports functional and performance analysis by means of simulation[2] and thus significantly reduces the risk of expensive design-implementation iterations. POOSL has been applied in many academic and industrial case studies and proved to be effective, see for example [2]–[4].

Recently, POOSL was used to explore the design space of motion control systems by constructing system models for performance analysis. The considered motion control algorithms are distributed over a multi-processor platform. The various processors in this platform are inter-connected by a network of packet switches that conform to the Rapid Input/Output (RapidIO) standard [5]. Packets are generated by processors to exchange messages with other processors on this network. A packet switch receives packets via its input ports from other packet switches (or processors), routes these packets by its crossbar fabric to appropriate output ports, and then sends those packets to other packet switches (or processors). Each input/output port has its own buffer, which works in a FIFO mode. The crossbar fabric is not buffered. Obviously, the latencies through the RapidIO network are closely related to the performance of the motion control system.

Motion control is characterized by feedback/feedforward control strategies and periodic execution. The main research challenge is how to map the considered motion control algorithms on the multi-processor platform such that the periodic timing constraints are met (i.e., all packets arrive at their destinations before the period ends). Packet latencies as worst-case latencies and average-case latencies are essential performance criteria for finding a feasible mapping. According to the proposed approach, a POOSL model is constructed first for a given mapping and then end-to-end latencies are analyzed. Subsequently, a feasible mapping is found if the end-to-end latencies satisfy the timing requirements; or else, other mappings are explored until a feasible mapping is found.

POOSL analysis gives both average-case and worst-case latencies. However, since it is based on simulation, the results are estimation results. For average-case results, the accuracy is indicated based on confidence intervals. However, such

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

[2]The theory underlying POOSL includes model checking techniques for analyzing functionality and performance, but no tools are available yet for actually doing that [1].

accuracy indication is not possible for worst-case metrics. Because motion control applications are safety-critical and time-critical, worst-case latencies are strict timing constraints. Exact worst-case latencies are therefore demanded. Motivated by this requirement, we propose to use model checking approaches for analyzing the worst-case latencies. To this end, we transform a POOSL model of a (simplified) motion control system into an UPPAAL model to enable formal verification. We then show that worst-case latencies and functional behavior can indeed be verified with the obtained UPPAAL model. We also propose a method for recording time (clock value) such that estimated worst-case (best-case) latencies can be obtained with UPPAAL simulator and then be used as a starting point for determining the exact worst-case (best-case) latencies by UPPAAL verification. Moreover, we show that the average-case latencies for the (simplified) motion control system can be derived from the obtained worst-case and best-case latency results. Our experiments show that the performance results verified with UPPAAL match the estimation results obtained from the POOSL model.

The rest of the paper is organized as follows. Section II describes POOSL modeling. Section III presents a POOSL model for the considered (simplified) motion control system. UPPAAL modeling is discussed in section IV. The transformation from the POOSL model to an UPPAAL model is discussed in Section V. We present the verification and performance analysis with the UPPAAL model in Section VI. Section VII briefly compares the POOSL and UPPAAL models and tools. Finally, conclusions are drawn in Section VIII.

II. Modeling a System with POOSL

POOSL was originally defined in [6] as an object-oriented extension of CCS [7]. Meanwhile, POOSL has been extended with time in [8] and probabilities in [9] to become a very expressive formal modeling language accompanied with simulation, analysis and synthesis techniques that scale to large industrial design problems [1].

POOSL supports three types of objects: data, processes, and clusters. 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.

The definition of data involves a name, a single inheritance relation, instance variables, and instance methods [1]. The instance variables specify the attributes of a data object. The behavior of data objects, which is purely sequential, is defined by (data) methods. Table I summarizes the syntax for expressions used for specifying data methods.

Defining a process class involves specifying a name, instantiation parameters and instance variables, a port interface and message interface, instance methods, an initial method call and an optional single inheritance relation [1]. The instantiation parameters and instance variables are the attributes of a process. The port interface lists the names of the ports via which a process exchanges messages. The message interface lists the signatures of all possible messages and includes for each message the port name, the symbol ! or ? for message send or receive respectively, a message name and a list of the types of the message parameters. The primitives for specifying process methods are listed in Table II.

TABLE I
POOSL data expressions

Expression E	Description
c	Constant
x	Variable
self	Reference Self
new(D)	Data Object Creation
currentTime	Current Model Time
$x := E$	Assignment
$E_1; E_2$	Sequential Composition
$E\ m(E_1, ..., E_i)$	Data Method Call
$E\ \hat{}m(E_1, ..., E_i)$	Superclass Data Method Call
if E_c **then** E_1 **else** E_2 **fi**	Choice
while E_c **do** E **od**	Loop
return E	Return

TABLE II
POOSL process statements

Statement S	Description
$m(E_1, ..., E_i)(v_1, ..., v_j)$	Method Call
E	Data Expression
$S_1; ...; S_n$	Sequential Composition
par S_1 **and** ... S_n **rap**	Parallel Composition
$E_p!m(E_1, ..., E_i)\{E_a\}$	Message Send
$E_p?m(v_1, ..., v_i \vert E_c)\{E_a\}$	(Conditional) Message Reception
$[E]\ S$	Guarded Execution
sel S_1 **or** ... **or** S_n **les**	Non-deterministic Selection
if E_c **then** S_1 **else** S_2 **fi**	Deterministic Choice
while E_c **do** S **od**	Loop
abort S_1 **with** S_2	Abort
interrupt S_1 **with** S_2	Interrupt
skip	Empty Behavior
delay E_t	Time Synchronization

POOSL provides clusters to create a hierarchical structure of processes (clusters). We omit further details since their behaviors are equivalent to the parallel composition of their constituents [9].

POOSL has a formal semantics, which is given in [9]. The semantics of the data part is defined as a denotational semantics. The semantics of the process part is given by a Plotkin-style structural operational semantics and defines for each POOSL model a timed probabilistic labeled transition system of the form $(C, C_s, A, \{\xrightarrow{a} \subseteq C \times D(C) | a \in A\}, T, \{\xrightarrow{t} \subseteq C \times C | t \in T^+\})$. Here, C is the set of configurations (or states). $C_s \in C$ refers to the initial configuration reflecting the start of an execution. A refers to the set of actions and T refers to the time domain (which can be integer or real-valued). There are two sets of labeled transition relations. The set $\{\xrightarrow{a} \subseteq C \times D(C) | a \in A\}$ denotes the action transitions, where $D(C)$ is the set of distribution functions over C; $D(C) = \{\pi : c \to [0,1] | \sum_{c \in C} \pi(c) = 1\}$. The action transitions originate from the use of expressions in table I or statements in table

II, except for the **delay**-statement. The **delay**-statement implies time transitions in the set $\{\xrightarrow{t} \subseteq C \times C | t \in T^{+}\}$.

An important property of the semantics of POOSL is that it relies on the two-phase execution model of [10], which alternates a phase of (asynchronously) performing enabled actions with a phase of advancing time. In other words, the semantics conforms to the property of action urgency by prioritizing performing actions over advancing time.

III. POOSL Model of the Motion Control System

Before presenting the POOSL model for our case study, we first explain some aspects of RapidIO packet switches, which are the most important components for our analysis. RapidIO is a data communication standard provisioned for interconnecting chips on a circuit board and circuit boards using a backplane. The standard now has matured into a cost-effective, switched based replacement for expensive proprietary busses in high-performance embedded systems, such as networking and communications equipment [5].

Figure 1 shows the architecture of a RapidIO packet switch used in our case study. It has 8 input ports and 8 output ports. Input/output ports are buffered using FIFO policy. Each input/output buffer has 8 slots. The input/output buffers work either at store-and-forward mode or cut-through forwarding mode depending on the mode chosen. The crossbar switch fabric is unbuffered. The packet switch is assumed to be lossless as it includes link-level flow and error control. Packets have 4 priority levels (from 0 to 3, where 0 and 3 denote the lowest and highest priority respectively). Each input/output port has an arbiter to decide whether a packet can be accepted or not. Input arbiters accept a packet if and only if the packet satisfies the following condition: *packet priority* $\geq$ *buffer occupancy - 4*. On the other hand, output arbiters work in a hierarchical way. They arbitrate on priority first and then use a Round-Robin policy.

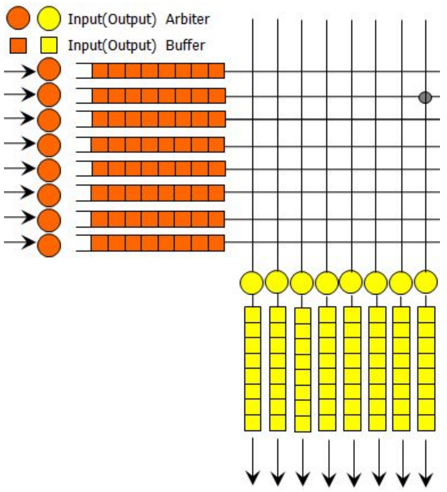


Fig. 1. RapidIO Packet Switch Architecture

Figure 2 depicts a POOSL model of a system consisting of 2 RapidIO switches (Switch1, Switch2) which interconnect 4 processor nodes (N1 to N4). This model is a simplified version of the real motion control system with 10 RapidIO switches for which a bigger POOSL model exists as well. This paper considers this smaller example to focus on the key aspects of transforming POOSL models into UPPAAL models. The nodes in figure 2 represent processors on which certain parts of the motion control algorithms are executed. This means that we have abstracted from the original motion control application model (containing about 1500 concurrent activities in POOSL) by using abstract packet generators in the processor node models. In the POOSL model of figure 2, the nodes can generate packets according to fixed or random patterns (the latter being a probabilistic abstraction of the original motion control application model). In this paper, we only consider fixed patterns because the standard version of UPPAAL does not support probabilistic behavior. Future work includes an investigation of translating probabilistic behavior expressed in POOSL into UPPAAL-PRO models [11].

Next to the processes Switch1, Switch2, N1, N2, N3 and N4, the POOSL model also contains a Monitor process. This Monitor non-intrusively collects individual packet latencies to calculate the minimum, maximum and average packet latencies as well as their variances. The switches and nodes are interconnected by channels c1 to c6, while nodes N1 to N4 communicate with Monitor via channel m. In this paper, we consider a configuration where nodes N1 and N2 send packets to Switch1, which appropriately routes and forwards these packets to Switch2. Switch2 forwards these packets to their destination nodes N3 or N4.

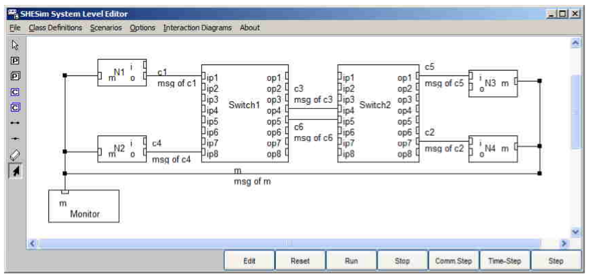
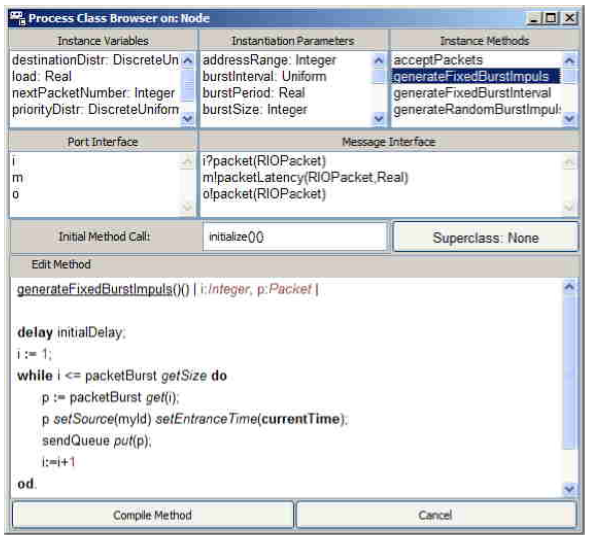


Fig. 2. POOSL Model of the (simplified) Motion Control System

Nodes N1 to N4 are instances of process class *Node* (see figure 3), which is responsible for generating and receiving packets. The generation and reception of packets is modeled in (independent) concurrent activities that are created by the initially called method *initialize* of class *Node* using the **par**-statement of table II. The reception of packets (which are instances of a data class *Packet*) is captured in method *acceptPackets* while the generation and sending of fixed patterns of new packets is specified with methods *generateFixedBurstImpuls*, *generateFixedBurstInterval* and *sendPackets*. Section V discusses this packet generation and sending behavior in more detail.

Switch1 and Switch2 are instances of process class *RIOSwitch*, which is depicted in figure 4. They capture the behavior of routing and forwarding packets. For each of the



```
destinationDistr: DiscreteUn
load: Real
nextPacketNumber: Integer
priorityDistr: DiscreteUniforn
```

```
addressRange: Integer
burstInterval: Uniform
burstPeriod: Real
burstSize: Integer
```

```
acceptPackets
generateFixedBurstImpuls
generateFixedBurstInterval
generateRandomBurstImpul:
```

```
i
m
o
```

```
i?packet(RIOPacket)
m!packetLatency(RIOPacket,Real)
o!packet(RIOPacket)
```

Initial Method Call: initialize()()

```
generateFixedBurstImpuls()() | i:Integer, p:Packet |

delay initialDelay;
i := 1;
while i <= packetBurst getSize do
     p := packetBurst get(i);
     p setSource(myId) setEntranceTime(currentTime);
     sendQueue put(p);
     i:=i+1
od.
```

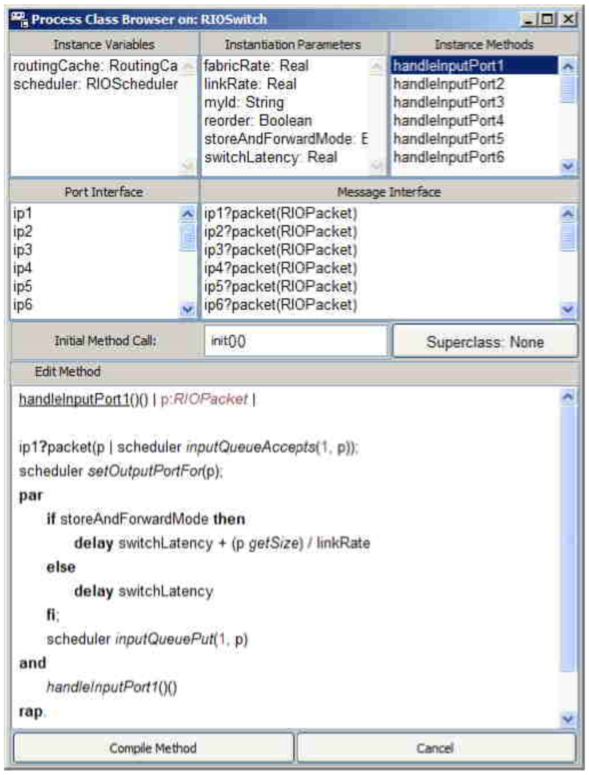
Fig. 3. POOSL Process Class *Node*

8 input ports, a concurrent input handler is created during the initial method call *init*. In addition, 8 output handlers are created. The behavior of each of these handlers is specified in a separate method. For example, *handleInputPort1* handles the packets received at input port ip1 and *handleOutputPort1* captures sending packets to output port op1. Method *schedule-ForOutput* specifies the actual arbitration of packets destined for a specific output port and hence, one such activity is being created for each of the 8 output ports during the initial method call *init*. Therefore, each switch includes a total of 24 concurrent activities. For each switch, its 24 concurrent activities operate on the shared instance variables *routingCache* and *scheduler*. Variable *routingCache* is an instance of data class *RoutingCache* and is used to store routing information. Variable *scheduler* is an instance of data class *RIOScheduler* and it implements the concrete output arbitration. Section V discusses the input and output handlers in more detail.

IV. Modeling a System with UPPAAL

UPPAAL is a tool for modeling, validation and verification of real-time systems. It is based on the theory of timed automata (TA) [12] and its modeling language offers additional features such as bounded integer variables and urgency [13]. The query language of UPPAAL, used to specify properties to be checked, is a subset of CTL (computation tree logic) [14], [15]. UPPAAL is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structures and real-valued clocks, communicating through channels and/or shared data variables. In this section, we briefly discuss the basic concepts and definitions needed in this paper.

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 the clocks progress synchronously. A system is modeled as a network of several



```
routingCache: RoutingCa
scheduler: RIOScheduler
```

```
fabricRate: Real
linkRate: Real
myId: String
reorder: Boolean
storeAndForwardMode: E
switchLatency: Real
```

```
handleInputPort1
handleInputPort2
handleInputPort3
handleInputPort4
handleInputPort5
handleInputPort6
```

```
ip1
ip2
ip3
ip4
ip5
ip6
```

```
ip1?packet(RIOPacket)
ip2?packet(RIOPacket)
ip3?packet(RIOPacket)
ip4?packet(RIOPacket)
ip5?packet(RIOPacket)
ip6?packet(RIOPacket)
```

Initial Method Call: init()()

```
handleInputPort1()() | p:RIOPacket |

ip1?packet(p | scheduler inputQueueAccepts(1, p));
scheduler setOutputPortFor(p);
par
     if storeAndForwardMode then
          delay switchLatency + (p getSize) / linkRate
     else
          delay switchLatency
     fi;
     scheduler inputQueuePut(1, p)
and
     handleInputPort1()()
rap.
```

Fig. 4. POOSL Process Class *RIOSwitch*

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 [13] for a more thorough description of the timed automata used in UPPAAL.

We present the basic definitions of the syntax and semantics for timed automata in UPPAAL using the following notations: C is a set of real-valued variables standing for clocks and $B(C)$ is the set of Boolean combinations of clock constraints of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in C$, $n \in N$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. A timed automaton is a finite directed graph annotated with conditions over, and resets of non-negative real valued clocks. Syntactically, a timed automaton is a tuple (L, l_0, C, A, E, I), where L is a set of locations, $l_0 \in L$ is the initial location, C is the set of clocks, A is a set of actions, co-actions (i.e., complementary actions, see below) and the internal action τ. The set $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations annotated with an action, a guard and a set of clocks to be reset. Finally, $I : L \rightarrow B(C)$ assigns invariants to locations.

The semantics of a timed automaton is defined as a labeled transition system $(S, s_0, \rightarrow)$, where $S \subseteq L \times R^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times \{R_{\leq 0} \cup A\} \times S$ is the transition relation such that:

- $(l, u) \rightarrow (l, u + d)$ if $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(l)$

- $(l, u) \rightarrow (l', u')$ if there exists $e = (l, a, g, r, l') \in E$ such that $u \in g, u' = [r \mapsto 0]u$ and $u' \in I(l')$

where for $d \in R_{\geq 0}$, $u + d$ maps each clock x in C to the value $u(x) + d$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over $C \backslash r$.

An UPPAAL model is defined as the parallel composition of a collection of timed automata. Semantically, such a network again describes a timed transition system obtained from those of the components by requiring synchronization on delay transitions and requiring discrete transitions to synchronize on complementary actions. Three types of synchronization are supported: binary synchronization, broadcast synchronization, and urgent synchronization. Suppose c is the channel name. For binary synchronization, an edge labeled $c!$ synchronizes with another labeled $c?$. In broadcast synchronization, a sender $c!$ can synchronize with an arbitrary number of receivers $c?$. For urgent synchronization, delay must not occur if it is enabled.

Finally, we remark that the flavor of timed automata in UPPAAL allows the use of data variables with finite domains, including finite records of Integers and Booleans and finite (multi-dimensional) arrays of data variables as well as urgent channels and locations [13]. It is also possible to declare functions for which the syntax follows the C/C++/Java style, and most control-flow constructs of C are supported. Functions are evaluated atomically and must be deterministic [16]. The next section illustrates how these features have been used to transform the POOSL model in figure 2 into an UPPAAL model.

V. Transformation from POOSL to UPPAAL

From the behavior point of view, POOSL has two parts: the process part covering the behavior of processes and clusters is based on a timed and probabilistic extension of CCS, while the data part (covering data objects) is based on the concepts of traditional object-oriented languages. In this section, we first focus on general transformation patterns matching the expressions and statements in table I for data part and table II for process part. Then these patterns are applied to transform the POOSL model of figure 2 into an UPPAAL model.

A. *Transformation Patterns*

The following transformation patterns illustrate how typical POOSL constructs are transformed into UPPAAL code (or timed automata).

1) ***Data Objects & Expressions****:* Data objects are specified using expressions in table I. It is fairly easy to do data object transformation as most expressions are also supported by UPPAAL. Dynamic object creation, and current model time are not directly supported. However, dynamic object creation can be abstracted away by a declaration in advance and current model time can be modeled by reference to a global clock. Now we give an example, the creation of an array with 8 elements *inputQueues := **new**(Array) size(8)* is transformed into an UPPAAL array declaration *Packet inputQueues[8]* which can then store 8 packets (Packet is a predefined structure for storing packets). Note that UPPAAL requires indication of the type of items stored in the array while for POOSL, the array can contain arbitrary objects. More sophisticated POOSL data objects can be translated into a combination of UPPAAL records and arrays of variables (as long as the involved data structures are finite and fairly static).

2) ***Processes & Statements****:* For process object transformation, we highlight the following patterns.

While processes in a POOSL model are easily identifiable from the component structure diagram like the one in figure 2 (since they are created statically), **par**-statement denotes dynamic creation of additional parallelism by means of concurrent activities within processes. In case the maximum number n of such activities is known in advance, one can translate each of the activities into a separate UPPAAL timed automaton. This should be performed for all processes. However, activities of different processes in POOSL can sometimes be united into a single UPPAAL timed automata when there is clear one-to-one synchronization relation between such activities. We illustrate this in sections V-C2 and V-C3.

The POOSL statement $E_p!m(E_1, ..., E_i)\{E_a\}$ denotes sending a message m through a port E_p with parameters $E_1, ..., E_i$. The complementary statement $E_p?m(v_1, ..., v_i|E_c)\{E_b\}$ specifies receiving a message m from a port E_p, where the parameters $E_1, ..., E_i$ of a matching sending statement are bound to variables $v_1, ..., v_i$. In case the message send and receive statements refer to the same channel, the message names m match and the number of parameters/variables i match, then synchronization will indeed take place (and the parameters are passed) at the moment condition E_c (which may depend on the parameter values and other variables within the sending and receiving processes) evaluates to true. The optional expressions E_a and E_b are evaluated atomically after the message is actually sent/received.

We transform the passing of messages between processes in POOSL into synchronization between timed automata in UPPAAL. The sending and receiving statements are captured using complementary actions ! and ? on a unique channel. The parameters $E_1, ..., E_n$ of the sending statement are transformed into UPPAAL expressions or functions. The variables $v_1, ..., v_i$ of the receiving statement are transformed into corresponding shared variables in the UPPAAL model and are assigned the values of the previously mentioned expressions or functions. The atomic expression E_a is captured as part of the update in the edge of the sending timed automaton. Similarly, expression E_b is captured as update in the edge of the receiving timed automaton. Finally, the condition E_c is transformed into a guard of the receiving timed automaton. Figure 5 illustrates this approach for the example where sending statement $c!m(p, q, r)\{r := 3\}$ matches with receiving statement $c?m(x, y, z|x >= 1)$. This transformation pattern has proven sufficient for translating the message passing actions in the POOSL model of figure 2.

The passage of time in a POOSL model originates from the use of statement **delay** E_t, where E_t evaluates to a positive

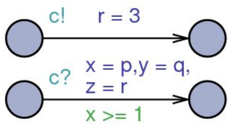


Fig. 5. Sending and Receiving Message Timed Automata

Real or *Integer* value. In case E_t evaluates to an Integer, this statement can be transformed into a time transition in UPPAAL. For instance, **delay** a (where a is an Integer) is translated to the timed automaton in figure 6. Here, c denotes a local clock. The synchronization *parSync* is added to enforce the start of this time transition. This pattern is used in several timed automata (as shown in figure 9,10,11).

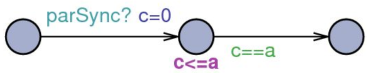


Fig. 6. Delay Timed Automation

Since time transitions in UPPAAL are selected non-deterministically, we need to enforce that all such transitions run simultaneously whenever they are enabled. Therefore, we propose to use the timed automaton in figure 7. When a request for broadcast synchronization through signal *reqSync* arrives, this timed automaton will initiate a broadcast synchronization signal *parSync* to start all enabled time transitions. *reqSync* signals are generated when system starts (as shown in figure 8) or whenever a concurrent activity finishes (as shown in figure 9,10,11).

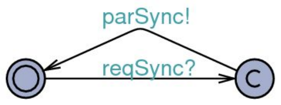


Fig. 7. Synchronization Timed Automaton

POOSL follows an action urgency policy, which means that time can only advance if there are no actions ready to be performed. In UPPAAL such actions are modeled as updates on edges. We enforce action urgency by putting them on the edges immediately before/after time transitions. Moreover, when necessary, a location is labeled as committed or urgent to enforce that time does not advance there.

B. Data part transformation

The POOSL model in figure 2 uses various data classes. *RIOPacket*, *RIOQueue*, *RoutingCache* and *RIOScheduler* are the most important ones. We briefly sketch the transformation for these data classes.

*1) **RIOPacket**:* *RIOPacket* is a simple data structure that contains information about an individual packet. The methods for *RIOPacket* allow accessing the instance variables without further complex operations on them. Such data classes can be easily transformed into an UPPAAL struct, for which access of its member variables comes for free. The UPPAAL struct modeling *RIOPacket* is named *Packet*.

*2) **RIOQueue**:* *RIOQueue* is a specialization of the more general data class *Queue*, which can be used to model bounded and unbounded queues with FIFO or LIFO queuing policies. The POOSL model in figure 2 only includes FIFO queues of bounded capacity (8 *RIOPacket*s). So, the transformation to UPPAAL of *RIOQueue* can be based on an array of size 8 that stores *Packet*s. *RIOQueue* has methods to handle the access of *RIOPacket*s. For example, the method *accepts* checks the condition *packet priority* $\geq$ *buffer occupancy - 4* of whether a new packet to arrive can be added to a *RIOQueue* (see also section III). Another method reorders the packets in a *RIOQueue* in correspondence to their priority levels. To capture these more complex behaviors, functions have been defined in the UPPAAL model that captures the same behavior.

*3) **RoutingCache**:* *RoutingCache* is transformed into a two-dimensional array of routing information in UPPAAL. The routing information consists of a struct with 4 member variables: *switch*, *source*, *destination*, and *localPort*. In addition, UPPAAL functions are provided to access the routing information. For example, function *getLocalPort(int switch, int source, int destination)* finds the local output port for a packet in a specific switch with given source and destination.

*4) **RIOScheduler**:* *RIOScheduler* is a complex data structure storing all relevant information (including all input and output queues) referring to the output arbitration for a switch. This data structure and its accompanying behavior can be transformed straightforwardly into UPPAAL by defining almost the same UPPAAL functions for each method of *RIOScheduler*. The main difference of these functions with the corresponding methods in POOSL model is that some extra information must be provided to specify the involved switch, routing table and matrix for Round-Robin arbitration.

C. Process part transformation

We first identify all active concurrent activities in the POOSL model of figure 2 to enable specifying timed automata in UPPAAL for each of them. Although each switch in the POOSL model contains 24 activities, only a few of them are actually active as a consequence of how the 2 switches interact with each other and the nodes. In fact, there are 2 input activities, 2 scheduling activities and 2 output activities within each *RIOSwitch* active. Moreover, there is one packet generation activity and one sending activity for Node1 and Node2, and one receiving activity for Node3 and Node4. For simplicity, we only show the transformation for some typical activities.

*1) **Packet generation activity**:* The POOSL model splits generation of packets from actually sending them. Methods *generateFixedBurstImpuls* and *generateFixedBurstInterval* implement the generation of a fixed sequence of packets that are stored into a FIFO queue *sendQueue* for sending. The packets are sent from this queue by method *sendPackets*. The timed automaton in figure 8 models the behavior of *generateFixedBurstImpuls* (POOSL code is shown at the bottom part of figure 3). The timed automaton starts when receiving the binary synchronization *reqStart* signal. It then calls function *generatePacket()* to generate a burst of predefined packets. Subsequently a *reqSync* signal is raised to ask for broadcast synchronization of all timed automata, after which it returns to the initial location waiting for next *reqStart* signal. The transformation of method *generateFixedBurstInterval* is omitted due to space limitation.

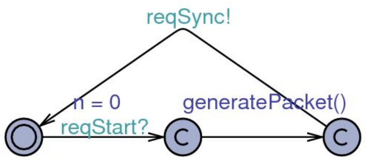


Fig. 8. Impuls Generation Activity Timed Automaton

*2) **Sending activity & input handler:*** We illustrate transforming the sending of packets by a node and the input handling of packets received by a switch with the communication over channel c1 in figure 2, which connects the output of node N1 to the input port ip1 of Switch1. The input handler for port ip1 of a *RIOSwitch* is specified by method *HandleInputPort1* shown in the bottom part of figure 4. Since channel c1 is only used for sending packets (specified by method *sendPackets*) by Node1 to the concurrent input handler specified by *HandleInputPort1*, we can unite the combined behavior into one timed automaton. Method *HandleInputPort1()* first specifies receiving a packet *p* in case the condition that *p* still fits into the input queue for port ip1 is satisfied. After setting the output port to which the packet should be forwarded, two concurrent activities are created with the **par**-statement. The first activity further deals with accepting and storing packet *p*, whereas the second activity enables receiving the next packet. For the parameter settings of the POOSL model that we consider in this paper, packets are never actually received concurrently. The completion of receiving *p* involves delaying for the time it takes to transmit the packet over the link connected to port ip1 (which depends on the packet size and link rate) and actually storing the packet into the input queue (which, for the parameter settings that we consider, is done when the packet has arrived completely to ensure store-and-forward operation).

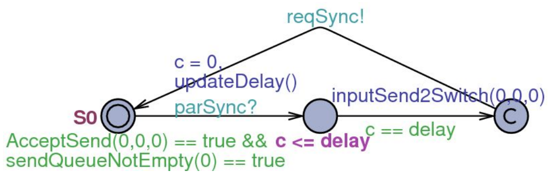


Fig. 9. Sending Activity and Input Handler Timed Automaton

The combined sending of a packet by node N1 and the reception of that packet as just described is captured by the timed automaton in figure 9. When broadcast synchronization signal *parSync* arrives, if the sending queue of node N1 is not empty (*sendQueueNotEmpty(0)==true*) and the first packet to send can be accepted by the input queue for port ip1 (*AcceptSend(0,0,0)==true*), the timed automaton delays for the appropriate amount of time determined by *updateDelay()*. This delay equals the packet transmission time over the link to port ip1. Function *inputSend2Switch(0,0,0)* is subsequently executed to move the packet from the sending queue of N1 to the input queue of port ip1. Finally, broadcast synchronization signal *reqSync* is initiated to request for synchronization of all timed automata.

*3) **Output handler & input handler:*** As the output port op1 of Switch2 is connected to the input port of node N3, the output activity of Switch2 specified by method *HandleOutputPort1* and the input activity *acceptPackets* of N3 can be united into one activity. This activity is modeled with the timed automaton of figure 10. When a broadcast synchronization signal *parSync* arrives, if output queue for port op1 of Switch2 is not empty (*outputQueuenotEmpty(1,0)==true*), the timed automata delays for an amount of time corresponding to the transmission time over the link between output port op1 and the input port of N3 (determined by *updateDelay()*). Subsequently, function *output(1,0)* is executed to delete the transmitted packet from the output queue for port op1 of Switch2. When the clock value corresponding to this packet is larger than the current candidate worst-case latency (*burstC[currentPacket.number] > worstLatency[currentPacket.number]*), a binary synchronization signal *calcLatencyW* is raised to update the candidate worst-case latency. Notice that this part of the timed automaton captures part of the quantitative analysis capabilities, which are discussed in more detail in section VI. Finally, the broadcast synchronization signal *reqSync* is raised to request for synchronization as above and the number *x* of received packets is incremented.

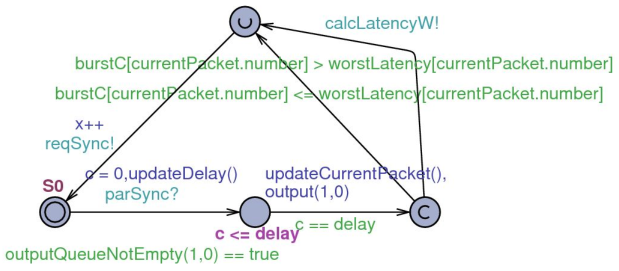

calcLatencyW!
burstC[currentPacket.number] > worstLatency[currentPacket.number]
burstC[currentPacket.number] <= worstLatency[currentPacket.number]
x++
reqSync!
S0
c = 0,updateDelay()
parSync?
updateCurrentPacket(),
output(1,0)
c == delay
c <= delay
outputQueueNotEmpty(1,0) == true
C

Fig. 10. Output handler and input handler Timed Automaton

*4) **Transfer activity:*** The transfer activity refers to the actual scheduling of packets from input buffers to a specific output buffer (namely output arbitration). In the POOSL model, this behavior is executed for each output port of a *RIOSwitch* and it is specified by method *scheduleForOutput* (see section III). As before, we use separate timed automata in UPPAAL for each of these concurrent activities that are actually active. Figure 11 depicts the timed automaton for the transfer activity in Switch1 corresponding to output port op4. When a broadcast synchronization signal *parSync* arrives, if a packet can be transferred to the output queue for port op4 of Switch1 (*packetTransferPossibleForOutput(0,3)==true*), this timed automaton executes three functions (*initiatePacketTrans-*

ferForOutput(0,3), *transferPacketForOutput(0,3)*, and *finishPacketTransferforOutput(0,3)*) to handle the actual transfer. Finally, it initiates a broadcast synchronization *reqSync* signal as above.

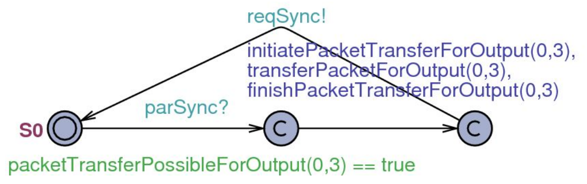

reqSync!
initiatePacketTransferForOutput(0,3),
transferPacketForOutput(0,3),
finishPacketTransferForOutput(0,3)
S0
parSync?
C
C
packetTransferPossibleForOutput(0,3) == true

Fig. 11. Transfer Activity Timed Automaton

VI. Verification and Analysis

The POOSL model in figure 2 contains a *Monitor* to analyze various properties of the model. Such a reflexive approach to functional and performance analysis is also used in the UPPAAL model. Functional analysis can be done by model checking, whereas, performance analysis will be done by UPPAAL simulator. Common use of the UPPAAL simulator is for the user to go through a trace (saved or imported from the verifier) to see if certain states are reachable. In this paper, we investigate the potential of using the simulator quantitative analysis. This section first presents how the functionality and worst-case latency can be verified. Then we show that further quantitative analysis is possible with UPPAAL simulator by introducing an approach to record time (clock value).

A. *Verification*

1) ***Functional Behavior Verification****:* Model checking techniques have mainly been applied to verify the functional behavior of systems. Such analysis is not always conclusive when simulation-based techniques are used, as in the case of POOSL, for properties like absence of deadlock. UPPAAL is however a very suitable tool for such verification. Considering absence of deadlock as a key functional property for the RapidIO network modeled in sections III and V, we constructed the timed automaton in figure 12 to verify this property. Here, x denotes the number of received packets and *nextPacketNumber* denotes the number of generated packets. A self-loop is used to prevent the deadlock from end of operation. The urgent channel *endSync* guarantees that this transition is taken immediately when the guard (x == *nextPacketNumber*) is satisfied. Absence of deadlock is expressed as *A[] not deadlock*.

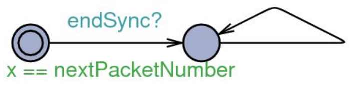


Fig. 12. Deadlock Free Timed Automaton

2) ***Worst-case Latency Verification****:* The worst-case latency of interest in this paper is defined as the largest latency for transferring individual packets through the RapidIO network. The timed automaton in figure 13 cooperates with the timed automaton in figure 10 to verify this worst-case latency metric. To this end, *worstLatency[index]* refers to a candidate worst-case latency for the packet with a specified index and *burstC[index]* refers to the clock for this packet. The binary synchronization signal *calcLatencyW* is raised when a packet has arrived at an output port of Switch2, see figure 10. Using this approach, the worst-case latency can be specified as a reachability property $E <>$ *WorstVerification.Violation*, where the latter part refers to the right-hand location in figure 13. If this property is not satisfied, then *worstLatency[index]* is a valid upper bound for the latency. To verify this, *worstLatency[index]* must be set to a specific value. It is obvious that the smallest upper bound for the latency (i.e., the worst-case latency) can be found iteratively by increasing the value of *worstLatency[index]* until this property is not satisfied. Section VI-B2 presents a novel approach to determine an initial value for *worstLatency[index]*. The proposed iterative approach follows the traditional way for modeling checking tools. A model checking approach that gives exact quantitative results without the need to iterate over a number of verification runs is an interesting research problem and will be considered as future work.

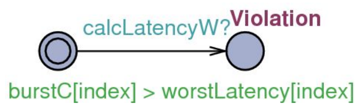


Fig. 13. Worst-case Latency Verification Timed Automaton

B. *Quantitative Analysis*

In the POOSL model of figure 2, the *Monitor* collects packet latencies to analyze performance metrics. We propose to use a similar approach for UPPAAL. However, UPPAAL does not support recording time directly. We therefore introduce the following approach to record time such that quantitative timing analysis is possible with UPPAAL simulator.

1) ***Recording Time****:* The technique for recording time is illustrated by the timed automaton in figure 14. It uses the binary search algorithm of [17] to find the value of a clock c and record it into a variable t. The initial values for l and h should be set based on the expectation that c is within [*low*, *high*].

2) ***Advanced Worst-case Latency Analysis****:* Using the timed automaton for recording time, we first show how worst-case latencies can be analyzed. The timed automaton in figure 15 presents the main idea. From the timed automaton in figure 10, a packet is sent to output op1 of Switch2. In case the corresponding clock value for this packet is larger than the current candidate worst-case latency, the binary synchronization signal *calcLatencyW* is raised to start the timed automaton in figure 15 which records the new latency as the new

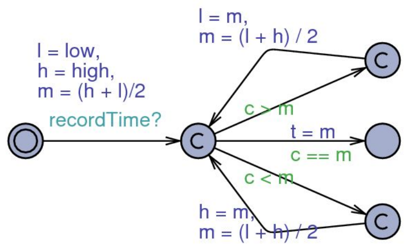


Fig. 14. Recording of Time

candidate worst-case latency. This approach allows evaluating the worst-case latency by means of simulating the UPPAAL model. This gives an approximation for the worst-case latency, which can subsequently be used as initial candidate for the iterative verification approach discussed in section VI-A2. This combined approach advances the overall verification process considerably.

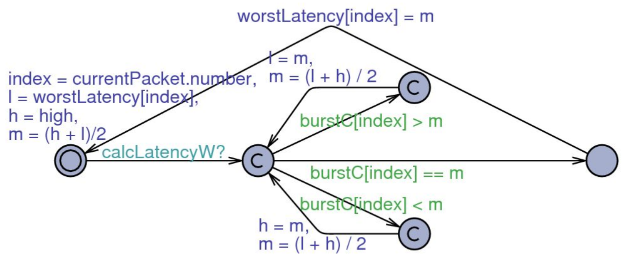


Fig. 15. Worst-case Latency Analysis Timed Automaton

Given different bursts of packets, we run a number of experiments to analyze the effectiveness of our approach. Table III presents the results and compares them with the approximate worst-case latencies obtained from simulating the POOSL model. We remark that the timing values of the POOSL model are scaled up by a factor 10 in the UPPAAL model to accommodate for using *Real*-valued time in the POOSL model. In the table, "Num" refers to the number of packets that is being handled by the RapidIO network. The more packets in the system, the more they have to wait for passing through the various queues and hence, the larger the latencies are. We can see that the UPPAAL results match with the POOSL results.

TABLE III
WORST-CASE LATENCY COMPARISON: UPPAAL VS. POOSL

Num	4	8	12	16	20	24	28	32
UPPAAL	40	60	80	100	120	140	160	180
POOSL	4.0	6.0	8.0	10.0	12.0	14.0	16.0	18.0

*3) **Average-case Latency Analysis:*** The approach presented in the previous section can also be used for analyzing the best-case latency. Given approximate results for the worst-case and best-case latency obtained by UPPAAL simulation, the question rises whether the average-case latency can be analyzed similarly as well. Given the fairly clear way of how the packets are generated, a natural conjecture is that the average-case latency is the average of the worst-case and best-case latencies: average-case latency = (total worst-case latencies + total best-case latencies)/(2*(number of packets)). We illustrate this approach of computing the average-case latency by considering the situation where nodes N1 and N2 both generate a burst of 5 packets. Table IV gives the resulting average-case latency where the packet with index "Index" has the listed worst-case and best-case latencies.

TABLE IV
AVERAGE-CASE LATENCY ANALYSIS

Index	0	1	2	3	4	5	6	7	8	9
Worst	30	40	50	60	70	30	40	50	60	70
Best	15	20	25	30	35	15	20	25	30	35
Average					37.5					

For several different settings, experiments have been run to find the average-case latency. The results are shown in table V. "Num" refers again to the number of packets that is being handled by the system. We can see that the average-case latencies obtained from the UPPAAL model match well with the results from the POOSL model. These experiments show that our conjecture seems valid for the given way of how the packets are generated. The experiments also confirmed that the results obtained by POOSL are indeed quite accurate estimations, both for the worst/best-case latency as well as for the average-case latency.

TABLE V
AVERAGE-CASE LATENCY COMPARISON: UPPAAL VS. POOSL

Num	4	8	12	16	20	24	28
UPPAAL	26.2	33.7	41.2	48.7	56.2	63.7	71.3
POOSL	2.62	3.37	4.12	4.87	5.62	6.38	7.12

VII. COMPARISON OF MODELS AND TOOLS

Compared with the UPPAAL model, the POOSL model is more understandable. Moreover, POOSL is object-oriented which eases expressing the system architecture as shown in figure 2. UPPAAL does not provide any means to get such a hierarchical view. On the other hand, UPPAAL gives a more intuitive picture of all concurrent processes in the system.

Compared with POOSL, the simulation power of UPPAAL is weak. It takes about 2 to 3 seconds for a single run of the POOSL model (for a burst of 14 packets), whereas it takes about 30 seconds using UPPAAL simulator for the same setting. For large models, the POOSL tool called rotalumis [1] is especially developed for high-speed execution. Simulating 100 bursts of hundreds of packets in the system only takes about 2 seconds. UPPAAL simulator cannot handle such large models. For this case study, it can handle at most 6 bursts of 32 packets in the system (the main problem is the amount of memory needed). The reason is that the UPPAAL simulator is primarily designed for going through trace (especially for counter example generation) and has not yet been optimized for simulation. This paper provides a novel method to do quantitative analysis with UPPAAL simulator. It shows that UPPAAL simulator certainly has potential for doing quantitative analysis. However, the capabilities are at an elementary level. More research is needed to explore its true potential and to extend its capabilities. First a more efficient technique for recording time is needed. Second, a more powerful simulation engine which conforms to the verification engine will be a key research problem. Other efforts for performance analysis with UPPAAL can also be found in the literature [18], [19]. However, the orientation is to use extended priced/weighted timed automata for optimal scheduling problems. Instead of using UPPAAL simulator to do quantitative analysis, they interpret such problems as cost-optimal reachability problems and use another branch UPPAAL-CORA [20] to solve them.

VIII. Conclusions and Future Work

The exact worst-case packet latency is an important metric for motion control applications that run on multiple processors interconnected by a RapidIO network. Although a POOSL model has been developed of this system, it cannot give exact performance results since POOSL analysis is based on simulation. We therefore propose a model checking approach using UPPAAL for this problem. To this end, we show that transforming a POOSL model into an UPPAAL model is feasible and how this can be done. We illustrate the advantage of using UPPAAL by verifying that the system is deadlock-free (which is an essential functional property of the system). Moreover, we explain how the exact worst-case latency can be determined by means of model checking. Unfortunately, this approach is rather time consuming due to its iterative manner. Hence, we propose an alternative approach to approximate worst-case latency. To this end, we introduce a method to record time such that quantitative analysis for worst-case latencies (as well as best-case latencies) is possible by simulation with UPPAAL. In addition, we show that average-case latency for the particular system configuration under study can be obtained as the average of the best-case and worst-case latencies. Several experiments have been conducted to confirm that the simulation results obtained with the POOSL model are indeed quite accurate both for the worst/best-case latency and average-case latency. The deficiency of the proposed method for quantitative analysis is that the UPPAAL simulator cannot handle large models.

Future work includes focus on a larger, more realistic case study to see if more abstractions can be made to obtain a UPPAAL model that can still be verified efficiently. We anticipate that by focusing on a specific performance metric such as the worst-case latency by omitting non-related details, larger models for more realistic systems can be handled. Next to this direction, we intend to elaborate our research on transforming POOSL into UPPAAL at a more semantical level by means of additional transformation patterns.

References

[1] B. Theelen, O. Florescu, M. Geilen, J. Huang, P. van der Putten, and J. Voeten, "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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 139–148.

[2] B. Theelen, J. Voeten, and R. Kramer, "Performance modelling of a network processor using poosl," *Comput. Netw.*, vol. 41, no. 5, pp. 667–684, 2003.

[3] B. Theelen, "Performance modelling for system-level design," Ph.D. dissertation, Eindhoven University of Technology, 2004.

[4] F. van Wijk, J. Voeten, and A. ten Berg, "An abstract modeling approach towards system-level design-space exploration," *System Specification and Design Languages*, vol. 22, pp. 267–282, 2003.

[5] G. Shippen, "A technical overview of rapidio," http://www.eetasia.com/ART_8800487921_499491_NP_7644b706.HTM, Nov. 2007.

[6] P. van der Putten and J. Voeten, "Specification of reactive hardware/software systems," Ph.D. dissertation, Eindhoven University of Technology, 1997.

[7] R. Milner, *Communication and Concurrency*. Prentice Hall, 1989.

[8] M. Geilen, "Formal techniques for verification of complex real-time systems," Ph.D. dissertation, Eindhoven University of Technology, 2002.

[9] L. van Bokhoven, "Constructive tool design for formal languages: From semantics to executing models," Ph.D. dissertation, Eindhoven University of Technology, 2002.

[10] X. Nicollin and J. Sifakis, "An overview and synthesis on timed process algebras," in *In A. K. G. Larsen, editor, Proceedings of the 3rd Workshop on Computer-Aided Verification*. Aalborg, Denmark: Springer-Verlag, 1991, pp. 376–398.

[11] A. David, A. Haugstad, and K. Larsen, "Uppaal-pro," http://www.cs.aau.dk/~arild/uppaal-probabilistic.

[12] R. Alur and D. Dill, "Automata for modeling real-time systems," in *Proceedings of the seventeenth international colloquium on Automata, languages and programming*. New York, NY, USA: Springer-Verlag New York, Inc., 1990, pp. 322–335.

[13] G. Behrmann, A. David, and K. Larsen, "A tutorial on uppaal," pp. 200–237, 2004. [Online]. Available: www.cs.aau.dk/~adavid/publications/21-tutorial.pdf

[14] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in cesar," in *Proceedings of the 5th Colloquium on International Symposium on Programming*. London, UK: Springer-Verlag, 1982, pp. 337–351.

[15] G. Logothetis and K. Schneider, "Symbolic model checking of real-time systems," *International Syposium on Temporal Representation and Reasoning*, vol. 0, p. 0214, 2001.

[16] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *Third International Conference on the Quantitative Evaluation of SysTems (QEST 2006),* 11-14 September 2006, Riverside, CA, USA. IEEE Computer Society, 2006, pp. 125–126.

[17] D. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 3rd ed. Addison-Wesley, 1997.

[18] G. Behrmann, K. G. Larsen, and J. I. Rasmussen, "Optimal scheduling using priced timed automata," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 4, pp. 34–40, 2005.

[19] K. G. Larsen, "Quantitative verification and validation of embedded systems," in *Proceedings of of 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE09, Tianjin, China*, 2009, to appear.

[20] G. Behrmann and K. G. Larsen, "Uppaal-cora," http://www.cs.aau.dk/~behrmann/cora/.