Providing Timing Computations for FUJABA*

Tobias Eckardt, Christian Heinzemann
Software Engineering Group,
Heinz Nixdorf Institute
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
tobie|c.heinzemann@uni-paderborn.de

ABSTRACT

Model-based software engineering aims at specifying the system under construction by abstract models that can be used for formal verification of the system behavior. In the case of real-time systems, such verification requires special algorithms dealing with time computations. These computations can be performed efficiently by using zone graphs [1, 3]. Current implementations, however, cannot be used in FUJABA. Therefore, we introduce a TCP/IP-based client/server architecture wrapping an existing implementation in a server such that it can be used by arbitrary clients. In our evaluation, we show that the TCP/IP overhead is negligible compared to the total run-time.

1. INTRODUCTION

Model-based software engineering aims at specifying the system under construction by abstract models that can be used for formal verification of the system behavior. This approach can also be used in the domain of real-time systems in order to build safe real-time systems by using appropriate models and verification techniques addressing the real-time characteristics [8]. The MECHATRONIC UML approach is one technique for model-based development of real-time systems [9].

A suitable formalism to model the behavior of real-time systems is given by timed automata [1] that have been extended to real-time statecharts [7] in the MECHATRONIC UML. Timed automata have successfully been used in Uppaal as a formal model for the verification of real-time behaviors [3]. Uppaal, however, cannot be used for all analysis techniques being applied in MECHATRONIC UML like refinement checking [10] or a behavioral synthesis [6]. Thus, such algorithms have to be implemented separately which requires the implementation of time computations in the case of real-time systems.

Time computations, as they are needed for our analysis techniques, can be efficiently performed by using so-called *zone graphs* [1] that are also used in Uppaal [3]. For Uppaal, there exists a C++ library, the Uppaal DBM library [4], implementing the necessary functionality for computing zone graphs. Additionally, a Ruby binding of this library exists. Both implementations have in common that they cannot be used in FUJABA directly.

We try to overcome this problem by providing a TCP/IP-based client/server architecture that allows to use the existing Uppaal DBM library by clients being implemented in arbitrary programming languages supporting TCP/IP. On the one hand, our architecture consists of a server, written in Ruby, that directly uses the ruby binding of the Uppaal DBM library. On the other hand, we provide a reference Java interface and a TCP/IP-based implementation of this interface managing the communication with the server. That interface can be used directly in FUJABA to implement the timing computations needed for our analysis techniques.

An alternative to our TCP/IP based client/server architecture would be, obviously, to write specific adapters to the Uppaal DBM library for each programming language and compile it for each operating system. In case of Java, a JNI (Java Native Interface) binding to the C++ library would be possible. Probably, such binding would be more efficient than our approach, but it restricts the usage of the library to one specific language and requires to re-compile the library for all required operating systems. The latter was simply not possible in our case due to missing third-party libraries.

The contribution of this paper is a TCP/IP-based client/server architecture providing efficient clock zone computations to all programming languages supporting TCP/IP.

The paper is structured as follows. First, we introduce the foundations of the paper (Section 2). Afterwards, the general architecture of our approach is described in Section 3. Then, we discuss the client and the server in detail in Sections 4 and 5, respectively. Finally, we present our evaluation results concerning the TCP/IP overhead in Section 6 before we conclude the paper in Section 7.

2. FOUNDATIONS

For the illustration of the possibilities using clock zones and clock zone operations we employ a timed automaton as a behavioral model with timing constraints as it can be specified in Uppaal (Figure 1).

Informally, a timed automaton consists of finite sets of *locations*, *transitions* and real-valued *clocks*. Starting in the *initial location*, it may either rest in a location or switch between locations using transitions and corresponding event occurrences. *Events* are modeled using a synchronous channel concept, where events can either be thrown using the special symbol “!” or received using the special symbol “?”.

The example automaton in Figure 1 describes the behavior of a simple lightswitch. By pressing the switch once the light is switched to dim; by pressing the switch twice within 10 ms the light is switched to bright. If pressing the switch a second time does not happen within 10 ms, the light is switched of again. If the light is currently switched to bright, it can also be switched off

*This work was developed in the course of the Collaborative Research Center 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.
This work was developed in the project "ENTIME: Entwurfstechnik Intelligente Mechatronik" (Design Methods for Intelligent Systems). The project ENTIME is funded by the state of North Rhine-Westphalia (NRW), Germany and the EUROPEAN UNION, European Regional Development Fund, "Investing in your future".

by the press operation. If this is not performed, it switches to off automatically inbetween 59.5 s and 60 s.

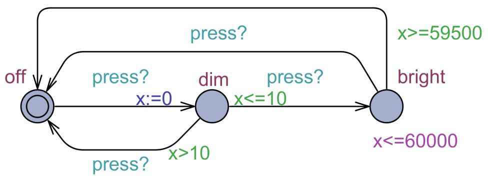

Figure 1: Example of a Timed Automaton describing a lightswitch

The timing of the behavior is specified using *time guards*, *clock resets* and *location invariants*. Initially, all clocks' values are set to zero. From then on, time can only pass, i.e. all clocks' values increase by the same value, while the automaton rests in a location, not while a transition is executed. Clocks can be reset using clock resets and the execution of a transition can be constrained to an integer-bound interval of clock values using clock constraints.

In the example, the clock x is used to measure the time that the system rests in dim. The time guard x<=10 at transition dim to bright further specifies, that this transition can only be executed if the value of x is between 0 and 10. If it is greater and the press-signal occurs, the transition from dim to off will be executed corresponding to its time guard x>10.

Finally, *location invariants* may be used to describe progress conditions. A location invariant describes an upper bound for the clock values in a certain location. In the example, the location invariant of location bright (in combination with the empty transition from bright to off) is used to specify that the light switches to off again automatically after 1 minute (60000 ms) at the latest, if the switch was not pressed before.

2.1 Analysis of Timing Specifications

While the untimed behavior of a state based specification can simply be performed by examining the states and transitions between states, this becomes more complex for timed specifications. Here, clock values, clock resets and clock constraints have to be taken into account. A suitable formalism for analyzing sets of clock values is the clock zone formalism [2, 1, 3] that is briefly described in the following.

A clock zone is syntactically described by a boolean conjunctive formula where the atomic propositions are inequalities with clock references and integer values describing clock value lower and upper bounds. Semantically, it describes an infinite, integer-bound set of clock values that can be visualized as a convex set in a k-dimensional euclidean space for k clocks being contained in the zone [1]. An example of a clock zone describing all values of clock x between 0 and 10 and all values of clock y higher than 20 is $x < 10 \wedge y > 20$.

If a clock zone is combined with a system state, for example a timed automaton location, this clearly defines a distinguishable timed state of the system, where the timing part is respresented as a set of clock values. For the calculation of transitions between states, whose execution is restricted to a distinct time interval only, for the consideration that time may elapse in some states and for the case that clocks may be resetted, operations on clock zones are provided. Four of these operations, which are the most important ones, are explained in the following.

The *time elapse* operation ($^{\uparrow}$), also called up-operation, describes the elapse of an arbitrary amount of time for a clock zone. It is realized by removing all upper bounds of a clock zone. The time elapse operation applied, for example, on the above given zone, denoted $(x < 10 \wedge y > 20)^{\uparrow}$ results in $y > 20$.

The *clock reset* operation describes the appliance of clock resets, that is setting the value of a set of clocks to zero. Applying this operation on the clock y and the example zone, denoted $(x < 10 \wedge y > 20)[\{y\} := 0]$, results in the zone $(x < 10 \wedge y = 0)$.

The *intersection* of clock zones ($\wedge$), also called and-operation, describes the set of clock values that are in both of the intersected zones. Intersecting the example zone with the zone $x > 5$, denoted $(x > 5) \wedge (x < 10 \wedge y > 20)$ results in $(x > 5 \wedge x < 10 \wedge y > 20)$.

The *subtraction* operation on clock zones subtracts one clock zone from another. This means that the subtrahend set of clock values is removed from the minuend set of clock values. An example is $(x < 10 \wedge y > 20) - (x <= 5)$ which results in $(x > 5 \wedge x < 10 \wedge y > 20)$.

In case of substractions on zones, the convexity of the set of clock values is no longer guaranteed. In this case, a time interval can be removed from the zone. The result is a non-convex set of clock values, called a *federation* [3], that can be represented by a finite number of convex sets (zones).

To give an example for the application of clock zones and corresponding operations, we show how a timed analysis model of the example timed automaton (Figure 1) can be created in the following. This timed analysis model (Figure 2), also called the *zone automaton* or *zone graph* [1], can, for example, be used to perform a reachability analysis over the timed system states.

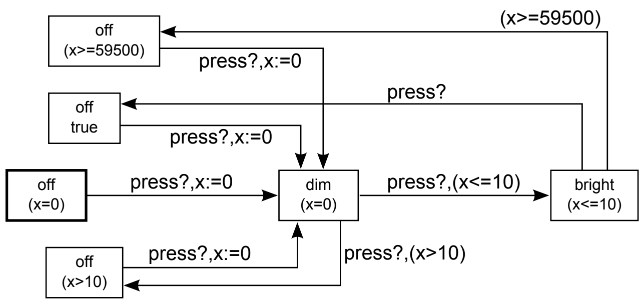

Figure 2: Zone Automaton of the Timed Automaton of the Lightswitch Example According to [1]

The zone automaton is created by starting in the first location, and the zone where all clocks are set to zero, in this case $(x = 0)$. For each outgoing transition, a successor zone location is now created by (1) applying the time elapse operation on the original zone, (2) applying an intersection with the location invariant of the source location, (3) applying an intersection with the time guard of the transition, (4) applying the clock resets of the transition and, finally, (5) applying an intersection with the location invariant of the target location. The resulting clock zone describes those clock values that are possible at the moment where the next target location is entered. In the example, the transition from (off,x=0) with the clock reset x:=0 leads to (dim,x=0) as the clock x must be zero when entering dim. On the other hand the transition from (dim,x=0) with the time guard x<=10 leads to bright with the zone x<=10 as the exact value of clock x is not known when entering bright, only that it is somewhere between 0 an 10.

After computing a successor zone in a zone automaton, a so-called normalization can be applied [3]. The normalization computes a canonical form of the zone and guarantees that the corresponding zone automaton of a timed automaton is always finite.

Other application examples, appart from model checking timed automata, are checking a refinement of timed automata as described in [6] or applying a reachability analysis on timed graph transformation systems as described in [11].

3. GENERAL ARCHITECTURE

The Uppaal DBM[1] library (UDBM, [4]) is a C++ library that was originally designed for the Uppaal model checker [3]. It implements operations on clock zones and federations (cf. Section 2.1) using DBMs [5] as an internal data structure for efficient memory management. We integrated this library into FUJABA using a client/server approach. The general architecture is shown in Figure 3.

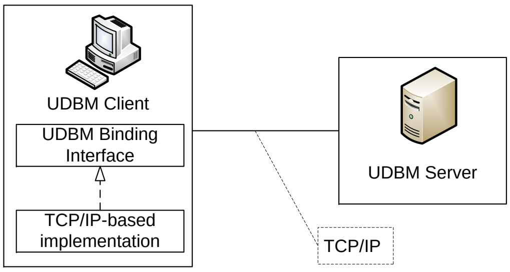

Figure 3: Architecture of the UDBM Integration

The UDBM Server executes the DBM operations using the Uppaal DBM library implementation. The UDBM client consist of an abstract UDBM Binding interface which can be used by application programs and a TCP/IP-based implementation of the interface managing the communication with the server in order to execute the operations requested by the application programs. Detailed information on our server and client implementation can be found in the subsequent Sections 4 and 5.

The client/server architecture allows to implement more than one client (even in different languages) for the same server as well as implementing more than one realization of the DBM computations without changing the client interface. Additionally, our architecture allows to execute client and server on different machines using different operating systems.

4. UDBM SERVER

The UDBM server is implemented in Ruby[2] and uses the precompiled Ruby binding of the Uppaal DBM library. The server manages the communication with the client and delegates DBM operation requests to the UDBM library. By default, the server opens a socket on port 8326 on localhost for client communication, but it is possible to pass a different port and hostname to the server on start up as a parameter.

The server implements the statemachine shown in Figure 4 that specifies the protocol to interact with it. The events before the "/" have to be passed as strings to the server, the events after the "/" are sent as strings back to the client. Strings in italics denote ruby code that is passed and directly executed by the server as described below.

The server starts in state Idle. First, a so-called *context* has to be created by passing the command createContextReq to the server. A context is required for the execution of the DBM operation as it defines the names of the clocks to be used. The server answers

[1]http://www.cs.aau.dk/~adavid/UDBM/
[2]http://www.ruby-lang.org

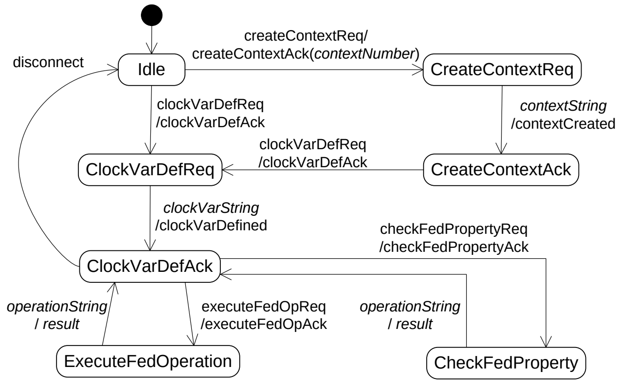

Figure 4: Statemachine of the UDBM Server

with an acknowledgement and a unique number for the next context to be created. Then, the client can submit an operation creating a context. This operation is submitted as ruby code which is then directly interpreted by the Ruby interpreter. In our example, a context for one clock x has to be created using the ruby code `c = Context.create('c0',:x)` for the context number 0.

The server answers with contextCreated to acknowledge the creation of the context before switching to CreateContextAck. The creation of a new context can be omitted if the context did not change compared to the last operation being executed. That allows to reuse contexts from prior operations thereby reducing the memory consumption of the server.

The second step is defining the clock variables. In order to use clock variables for the specification of clock constraints, the clocks being defined in the context have to be bound to variables. The client submits a clockVarDefReq to the server which switches into ClockVarDefReq. Again, the definition of the variables is encoded into ruby code submitted as a string. For our clock x, the submitted ruby code would be `x=c0.x;` As before, the server acknowledges the operation and switches to ClockVarDefAck.

The state ClockVarDefAck has two outgoing transitions representing the two possible classes of DBM operations. First, a property of a DBM can be checked. Such an operation always evaluates to either *true* or *false*. Second, an operation can be executed on a DBM such as intersection with another DBM. Such an operation always evaluates to a DBM. The client can select the desired operation by submitted either checkFedPropertyReq or executeFedOpReq to the server. Then, the actual operation to be executed has to be passed as ruby code as before and is directly interpreted. The result, either a Boolean or a DBM, is returned to the client. For instance, the operation `((x>=0)).and!(x<=10)` will evaluate to the DBM `((x>=0)&(x<=10))`.

After all operations have been executed, the client can send disconnect to the server causing it to switch to idle.

Then, a new context having a different number of clocks compared to the prior context can be created. That allows to support changing DBM sizes during the run of an algorithm on the client side.

5. JAVA UDBM CLIENT

In addition to the server, we have implemented a Java side client for the UDBM server. As introduced in Section 3, the client consists of an abstract interface for modeling DBMs as shown in Figure 5 as well as a TCP/IP-based implementation managing the communication with the server. Both are implemented as Eclipse plugins

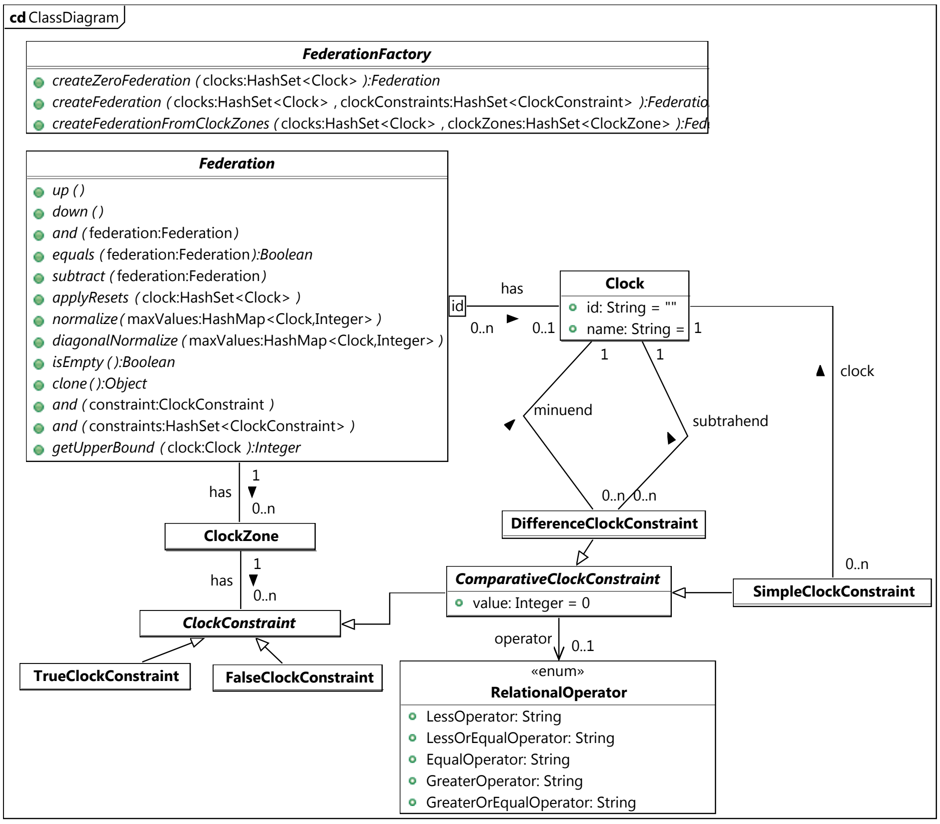

Figure 5: Class Diagram of the UDBM Java Client

and can be used in any Eclipse based tool such as FUJABA.

The bottom part of the client model allows the definition of clock constraints as defined in Section 2.1. In the two simplest cases, a clock constraint is either true or false represented by the classes TrueClockConstraint or FalseClockConstraint. In a ComparativeClockConstraint, a comparison with an integer is supported. Therefore, these clock constraints have a value and an operator. In a SimpleClockConstraint, the value of one clock is compared to the integer while in a difference clock constraint the difference of two clocks is compared to this value. In our example, $x <= 10$ of $x == 0$ are instances of a SimpleClockConstraint using the clock x. The classes can be used to model all valid clock constraints.

The left hand side of the model (ClockZone and Federation) is used to model clock zones and federations. For the sake of consistency, each zone must be contained in a federation even if there is only one zone in the federation. In our example in Figure 2, each represented zone can be represented in one zone and thus, each federation consists of one zone, only.

The class Federation also defines the interface to the operations which can be performed on a federation. The operation and, e.g., allows to intersect a federation with additional clock constraints or another federation. The executed operation is then transformed into a query to the server and the provided result and parsed back into a federation.

Clocks are assigned to federations because all zones in one federation must be specified over the same set of clocks. In order to improve memory efficiency, clocks can be used for different federations. In our example, all federations share the same clock object x.

The Java interface allows to add and remove clock instances from federations. The addition and removal of clocks can be easily done on the object level. Clocks being added to a federation are initialized with the value 0.

In some application scenarios, e.g. the reachability analysis introduced in [11], fast equivalence checks on DBMs are required. Therefore, the client interface implements a hash algorithm on federations fulfilling the general hash function contract.

$$f_1 \equiv f_2 \Rightarrow hash(f_1) = hash(f_2)$$

That means whenever the federations are equal, their hash values are equal as well. Thus, the equality check invoking the server only has to be executed in case of equal hash values.

6. EVALUATION

We evaluated the performance of our server and the TCP/IP connection using a socket via localhost utilizing the rechability analysis and the example presented in [11]. There, nine samples for run-times of a reachability analysis were presented. We choose to use this example, although it produces some odd numbers of DBM operations, because we wanted to have a realistic sample of DBM operations. During the reachability analysis, the size of the DBM varies such that multiple contexts have to be created (cf. Section 4). The results are summarized in Table 1. In the table, one DBM operation refers to the execution of one operationString in the protocol of Figure 4.

The runtime results have been obtained by first measuring the runtime on the Java side in order to obtain a runtime result including the TCP overhead. Second, we measured the runtime inside the ruby server to obtain a runtime result without the TCP overhead. Finally, the TCP overhead has been obtained by arithmetics. The

Table 1: Evaluation results

# of DBM Operations	Run-time of DBM Operations in s			Server memory in MB
	Server incl. TCP	Server excl. TCP	TCP	
108	0,3	0,3	0,0	15
342	1,0	1,0	0,0	15
737	3,4	3,1	0,3	16
1329	8,5	8,0	0,6	25
2154	17,1	16,4	0,7	25
3248	33,4	32,4	1,0	31
4647	61,1	59,1	2,0	35
6387	109,9	106,4	3,4	45
8504	190,1	185,3	4,8	63

run-time results in Table 1 are the sum of all executed DBM operations. The results show that the runtime increases slightly faster than the number of executed DBM operations. This is due to the fact that the maximum size of the DBMs increases from row to row. Thus, the additional runtime results from the fact that operations on larger DBMs consume more computation time. The overhead introduced by the TCP/IP connection to the server is approximately 3% of the overall runtime which we consider as quite low.

The memory consumption of the server includes the memory consumption of the ruby interpreter running the server script and the ruby binding of the Uppaal DBM library. Due to the reuse of contexts, the memory consumption increases only slowly for a large number of DBM operations. In cases where the DBM dimension does not change during runtime, the increase in memory consumption will be 0.

7. CONCLUSION AND FUTURE WORK

In this paper, we introduced a client/server architecture for integrating time computations into FUJABA. The Java client allows to model clock as well as constraints on these clocks that can be represented by clock zones. The server uses the Uppaal DBM library to perform the actual time computations. Our presented architecture is flexible as the server can be used by application programs written in any programming language supporting TCP/IP communication. Additionally, our client interface is independent of the actual server implementation. The overhead introduced by the TCP/IP communication is negligible according to our evaluation results.

Our implementation allows for an easy integration of time computations in any real-time analysis algorithm.

In our future work, we will try to apply further optimizations to our implementation. One of these optimizations is the support of concurrency in the server by allowing and processing multiple connections in parallel. Additionally, different server implementations could be evaluated for obtaining the most efficient realization of time computations.

8. REFERENCES

[1] R. Alur. Timed Automata. In N. Halbwachs and D. A. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99), July 6-10, 1999, Trento, Italy*, volume 1633 of *Lecture Notes in Computer Science (LNCS)*, pages 8–22. Springer Verlag, 1999.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, D. L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *Proceedings of the Third International Conference on Concurrency Theory (CONCUR '92)*, Lecture Notes in Computer Science (LNCS), pages 340–354, London, UK, 1992. Springer-Verlag.

[3] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[4] A. David. *UPPAAL DBM Library Programmer's Reference*, Oct. 2006. http://www.cs.aau.dk/~adavid/UDBM/manual-061023.pdf.

[5] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, volume 407 of *Lecture Notes in Computer Science (LNCS)*, pages 197–212. Springer Berlin / Heidelberg, Feb. 1990.

[6] T. Eckardt and S. Henkler. Component behavior synthesis for critical systems,. In *Architecting Critical Systems*, volume 6150 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin / Heidelberg, 2010.

[7] H. Giese and S. Burmester. Real-time statechart semantics. Technical Report tr-ri-03-239, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Germany, June 2003.

[8] H. Giese, S. Henkler, M. Hirsch, V. Roubin, and M. Tichy. Modeling techniques for software-intensive systems. In D. P. F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*, pages 21–58. Langston University, OK, 2008.

[9] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time uml designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[10] C. Heinzemann, S. Henkler, and A. Zündorf. Specification and refinement checking of dynamic systems. In P. V. Gorp, editor, *Proceedings of the 7th International Fujaba Days*, pages 6–10, Eindhoven University of Technology, The Netherlands, November 2009.

[11] C. Heinzemann, J. Suck, and T. Eckardt. Reachability analysis on timed graph transformation systems. In *Proceedings of the Eighth International Workshop on Graph Based Tools (GraBaTs 2010)*, volume 31 of *Electronic Communications of the EASST*, 2010.