Exploring Model-Based Development for the Verification of Real-Time Java Code

Niusha Hakimipour[1], Paul Strooper[1], and Roger Duke[1]

University of Queensland, St. Lucia, Queensland, Australia
niusha@itee.uq.edu.au, pstroop@itee.uq.edu.au, rduke@itee.uq.edu.au

Abstract. Many safety- and security-critical systems are real-time systems and, as a result, tools and techniques for verifying real-time systems are extremely important. Simulation and testing such systems can be exceedingly time-consuming and these techniques provide only probabilistic measures of correctness. There are a number of model-checking tools for real-time systems. However, they provide formal verification for models, not programs. To increase the confidence in real-time programs written in real-time Java, this paper takes a modelling approach to the design of such programs. First, models can be mechanically verified, to check whether they satisfy particular properties, by using current real-time model-checking tools. Then, programs are derived from the model by following a systematic approach. To illustrate the approach we use a nontrivial example: a gear controller.

1 Introduction

Real-time [1] is a broad term used to describe applications that have timing requirements. Many safety- and security-critical systems are real-time systems and, as a result, tools and techniques for verifying real-time systems are extremely important. The traditional ways of ensuring that real-time systems operate correctly have been simulation and testing. However, in many cases these techniques are exceedingly time-consuming and provide only probabilistic measures of correctness. Formal methods advocate the use of mathematical reasoning as an alternative; one of the most promising of these methods has been model-checking [2].

There are a number of model-checking tools for real-time systems [3–6]. However, they provide formal verification for models, and no systematic approach for deriving programs from those models. This means it is still necessary to show that the programs that implement those models satisfy the properties as well.

Real-time systems have to generate their output within a finite and predicable time. Therefore, the specification of the language in which real-time systems are implemented is as important as verifying such systems. Real-Time Specification for Java (RTSJ) [1] was proposed in January 2002. Sun has developed a simulator, Java Real-Time System (Java RTS) 2.0 [7], for simulating real-time Java code that is compliant with the RTSJ.

To verify real-time Java code which is compliant with the RTSJ, an approach based on JPF (Java PathFinder) [8] has been proposed by Lindstrom et al. [9].

JPF is a Java model-checker which has a state-exploring JVM (Java Virtual Machine) at its core. However, the approach based on JPF to verify real-time Java code has not been implemented yet, and it only supports properties that are specified as normal Java assertions, without timing constraints.

A real-time model is a simplified representation of a real-time system. Models focus on system behaviour and abstract many details of programs [10]. More importantly, these models can be verified mechanically with real-time model-checkers. This paper investigates a modelling approach to design real-time programs written in RTSJ, by means of an industrial example. In this approach, Timed Automata [11] are used as the modelling language, since Timed Automata have well-defined mathematical properties and a simple graphical representation. Moreover, Timed Automata can capture both qualitative and quantitative features of real-time systems [12]. The next step is to mechanically verify the model using the UPPAAL model-checker [4]. UPPAAL has a graphical user interface; it is well-used and well-supported. After verifying the model, a mapping between the model features and RTSJ are used to derive the RTSJ code from the model. This approach can increase the confidence in the correctness of the program.

In this paper, we present an initial application of the proposed approach to a nontrivial example. The RTSJ code for this example is derived by hand, following the systematic approach. The current mapping we propose does not deal with timing constraints on specific time values (rather than lower- or upper-bounds) which are described in Section 3. We have also left the mapping of a number of challenging Timed Automata features for future work.

In the next section, theories and languages that have been proposed for modelling real-time systems and related work on real-time model-checking are reviewed. In Section 3, we investigate an approach to implement the behaviour exhibited by real-time models in RTSJ. A realistic industrial case study, a gear controller [13], is used as an example in this section. Section 4 provides the verification result of the gear controller and discusses the limitations of our approach, and Section 5 concludes the paper.

2 Background

Traditional formalisms for temporal reasoning deal with the qualitative aspect of time, that is, the order of certain system events (an example of a qualitative time property is: event A occurs before event B). However, real-time systems often require quantitative aspects of time. This means they need to consider the actual difference in time between certain system events.

Timed Automata [11] provide a formalism for the modelling and verification of real-time systems. Examples of other formalisms are Timed Petri Nets [14], Time Petri Nets [15], Timed Process Algebras [16], and Real-time Logics [17].

Model-checking of Timed Automata representations has become popular for the analysis of real-time systems [18]. In the last decade, there have been a number of tools developed based on Timed Automata to model and verify real-time systems, notably Kronos [3], UPPAAL [4], RT-Spin [5] and MOCHA [6]. Timed Automata can capture both qualitative and quantitative features of real-time systems [12]. For instance, liveness, fairness and nondeterminism are qualitative features and bounded response and timing delays are quantitative features that can be captured with Timed Automata. Timed Automata also have well-defined mathematical properties and a simple graphical representation.

A Timed Automaton is a finite-state automaton extended with a finite set of real-valued variables modelling clocks. Timed words in a Timed Automaton are infinite sequences in which a real-valued time of occurrence is associated with each symbol [11]. Each clock can be reset to zero with the transitions of the automaton, and keeps track of the time elapsed since the last reset. The transitions of the automaton put certain constraints on the clock values. A transition may be taken only if the current values of the clocks satisfy the associated constraints.

Figure 1 shows the Timed Automata used by the UPPAAL model checker for a clutch specified by Lindahl et al. [13]. The UPPAAL Timed Automata [19] extends Timed Automata with a number of additional features such as bounded integers, arrays and urgent locations, as discussed in Section 3.

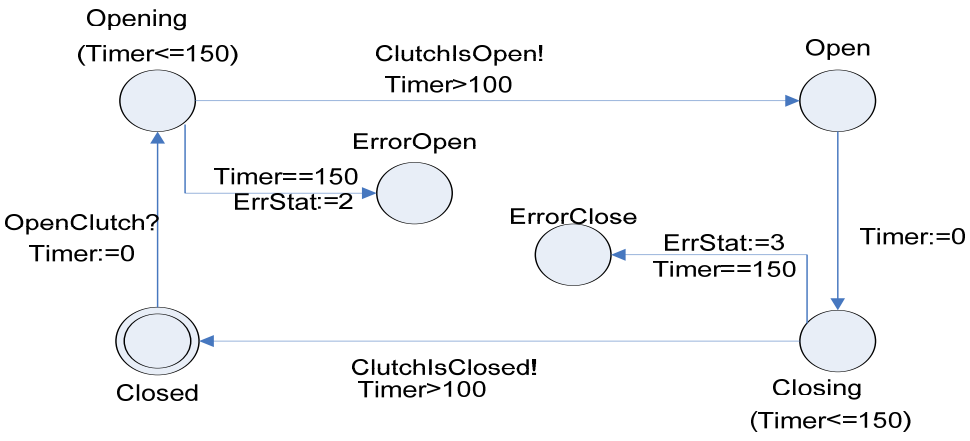

Fig. 1. Timed Automaton representing Clutch

The Clutch provides services to open or close the clutch in 100 to 150μs. In the case that opening or closing the clutch takes more than 150μs, the clutch will

stop in an error state. This example will later be used to illustrate our proposed approach.

Channels in Timed Automata are used to synchronize and communicate. For `Channel CName`, `CName?` represents receiving a message and `CName!` represents sending a message. In Figure 1, a message is sent via `OpenClutch!` and two messages are received via `ClutchIsOpen?` and `ClutchIsClosed?`. `OpenClutch?`, `ClutchIsOpen!` and `ClutchIsClosed!` are in another Timed Automaton not shown in Figure 1. `Timer` is a clock and `Timer==150` is a guard. A guard in UPPAAL is a side-effect-free statement which evaluates to a boolean. The transition from the `Opening` to the `ErrorOpen` can be taken if and only if `Timer==150` is enabled. In the `Opening` state, `Timer<=150` is an invariant. The Automaton needs to leave `Opening` before this invariant is violated.

3 From Models To Implementations

A Model is a simplified representation of the system. We use Timed Automata to describe models. These models represent the behaviour of real-time programs written in RTSJ and they can be verified mechanically with the UPPAAL model-checker. Then, we apply our approach on these models to design real-time programs which still satisfy the properties. Figure 2 shows an overview of this model-based approach, which is similar to the model-based approach proposed by Magee and Kramer to design concurrent Java programs from FSP models [10].

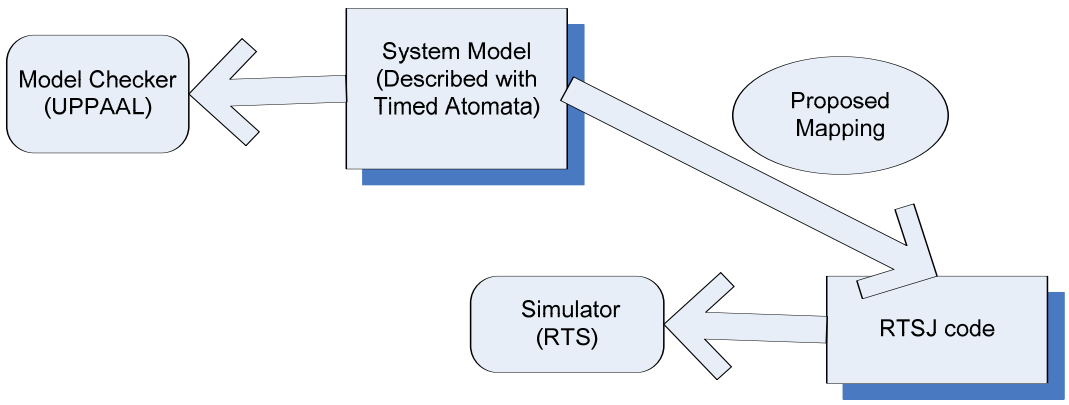

Fig. 2. Architecture

3.1 RTSJ

Real Time Specification of Java (RTSJ) [1] was introduced in January 2002. RTSJ is designed to support both hard and soft real-time applications. RTSJ adds several features to Java, such as Clocks, Time, Scoped Memory Areas which provide guarantees on allocation time, Fixed Priority Scheduling Policy, Asynchronous Events and Real-Time Threads.

Figure 3 shows part of the RTSJ code for the clutch Timed Automaton discussed in Section 2. The details of mapping the Clutch Timed Automaton to the real-time Java code are described in Section 3.3.

`Clutch` is a real-time thread. Two classes, `NonHeapRealtimeThread` and `RealtimeThread`, are defined in RTSJ to support real-time threads. Non-heap real-time threads are not targeted by the garbage collector [1].

`ClutchClock` is declared as a clock. Clocks in RTSJ are derived from an abstract class called Clock. There are three types of clocks in RTSJ:

- A "`monotonic`" clock progresses at a constant rate, suitable for timeouts.
- A "`countdown`" clock can be reset to zero, paused or continued.
- A "`CPU execution time`" clock counts the amount of time that is being consumed by a particular thread.

`MaxTimeClutch` and `CurrentTime` are a relative and absolute time respectively. In RTSJ, time is defined by three classes:

- a duration measured by a particular clock is "`relative`" time;
- "`absolute`" time is a time relative to some epoch, such as system start-up time;
- "`rational`" time is a subclass of relative time to represent the rate of certain event occurrences.

3.2 Model-based approach

An overview of the mapping for different features and expressions in UPPAAL Timed Automata is shown in Tables 1 and 2. The details of the mapping are provided in Section 3.3.

3.3 Mapping Details

Timed Automaton: Every Timed Automaton is mapped to a non-heap real-time Java thread. As non-heap real-time Java threads are not targeted by the garbage-collector, programs using such threads have no non-determinism due to garbage-collection delays or memory allocations. Each thread has a state

```
1.public class Clutch extends NonHeapRealTimeThread{
2.   public void run(){
3.        Environment env = new Environment();
4.        Clock ClutchClock = Clock.getRealtimeClock(); // Timer
5.        RelativeTime MaxTimeClutch = new RelativeTime(0,150);
6.        RelativeTime MinTimeClutch = new RelativeTime(0,100);
7.        AbsoluteTime CurrentTime = ClutchClock.getTime();
8.        String state = "Closed";
9.        while(true){
10.            if(state == "Closed"){
11.                if(env.IsReadyOpenClutch){
12.                   env.IsReadyOpenClutch = false;
13.                   env.ChannelAcknowledgeOpenClutch = true;
14.                   CurrentTime = ClutchClock.getTime();
15.                   state = "Opening";
16.                   continue;
17.                }
18.             }
19.             if(state == "Opening"){
20.                 if(((ClutchClock.getTime().subtract(CurrentTime)).compareTo(MaxTimeClutch) >= 0)){
21.                    env.ErrStat = 2;
22.                    state = "ErrorOpen";
23.                    continue;
24.                 }
25.                 if(((ClutchClock.getTime().subtract(CurrentTime)).compareTo(MinTimeClutch) > 0)){
26.                    env.ChannelAcknowledgeClutchIsOpen = false;
27.                    env.IsReadyClutchIsOpen = true;
28.                    while(!env.ChannelAcknowledgeClutchIsOpen);//busy loop
29.                    state = "Open";
30.                    continue;
31.                 }
32.             }
33.             if(state == "Open"){...}
34.             if(state == "Closing"){...}
35.       }/*while*/
36.  }/*run*/
37.}/*class*/
```

Fig. 3. Potential RTSJ code corresponding to Clutch Timed Automata

Table 1. Features Mapping Table

Feature	**Description**	**Currently supported**	**Mapped to**
Timed Automaton	a finite-state machine extended with clock variables	Yes	Real-time Thread
Broadcast channels	channels that are not blocking	Yes	A variable in the `Environment` class
Binary synchronisation	channels are declared as `chan c`	Yes	Two variables in the `Environment` class
Urgent location	time is not allowed to pass in an urgent location	Yes	Resetting the value of the Clock
Urgent synchronisation	delays must not occur if its channel is enabled	No	
Committed location	a state that cannot delay	Partially	Using RTSJ Priorities
Initialisers	used to initialise integers and arrays	Yes	Assignments in the thread constructor

Table 2. Expressions Mapping Table

Expression	Description	Currently supported	Mapped to
Assignment	an expression with a side-effect	Yes	An assignment in RTSJ
Guard	a side-effect free expression associated with a transition	Partially	An `if` condition
Invariant	a side-effect free expression associated with a state	Partially	An `if` condition except for time invariants

variable that is initialised to the initial state of the Automaton. The behaviour for the Automaton is encoded in an infinite loop in the thread `run` method. This loop contains several `if` statements on the `state` variable and each `if` statement contains the behaviour of the Timed Automaton in a state with at least one outgoing edge. As an example, Figure 3 shows the real-time thread corresponding to the Clutch Timed Automaton. Inside the `run` method of the Clutch thread in Figure 3, the string variable `state` represents the state, which is initialised to `Closed`. The infinite while loop contains four `if` statements corresponding to the four states with at least one outgoing edge: `Closed`, `Open`, `Closing` and `Opening`.

Global Variables and Broadcast Channels: To model global variables, one additional class, `Environment`, is introduced to implement the environment. The `Environment` contains global variables as static variables and all threads that need to access global variables create an instance of the `Environment` object. Broadcast Channels are considered as global variables, since they are non-blocking. For example, inside the `run` method of the Clutch thread in Figure 3, an instance of `Environment` is created to access the shared variable, `ErrStat`.

Binary synchronisation: In order to model a synchronous channel `C`, two boolean variables are introduced, `IsReadyC` and `ChannelAcknowledgeC`. The variable `IsReadyC` is set to *true* by the sender to inform the receiver that a new message is put in the channel `C` and receiver sets this boolean to *false* whenever it reads a new value from the channel variable. The `ChannelAcknowledgeC` ensures that the sender will not progress until the receiver receives the message. Whenever the sender sets its channel variable, it also sets the `ChannelAcknowledgeC` to *false* and will not continue until this variable is *true* again. Receiver sets this `ChannelAcknowledgeC` to *true* when it has read the message. The initial value of `ChannelAcknowledgeC` and `IsReadyC` are *true* and *false* respectively. In Figure 3, the clutch is the receiver for the `OpenClutch` channel. A transition from `Closed` to `Opening` is taken when a new message is put in the `OpenClutch` channel (line 11). When the clutch receives the `OpenClutch` message, it sets `IsReadyOpenClutch` to *false* to be ready for the next message and also sets `ChannelAcknowledgeOpenClutch` to *true* to inform the sender that it received the message (lines 12 and 13). On the other hand, the clutch is the sender for the `ClutchIsOpen` channel. It sets the `IsReadyClutchIsOpen` and

`ChannelAcknowledgeClutchIsOpen` variables (lines 26 and 27) and it waits until the receiver receives this message (line 28).

Urgent Locations: Time is not allowed to pass when the system is in an urgent or committed location. For a Timed Automaton, this is semantically equivalent to a location with incoming edges resetting the Timed Automaton clock and labelled with the invariant `Clock<=0`. However, interleavings with normal states are allowed. To model urgent locations we will add an assignment that saves the value of the clock after all lines of code that lead to an urgent location and then set back the clock to this value when the program leaves the code corresponding to such a location. However, the discrepancy between model and code must be noted and analysed. This feature does not occur in the gear-controller example.

Urgent Synchronisation: In an Urgent Synchronisation, if a synchronisation transition on an urgent channel is enabled, delays must not occur. In RTSJ, a priority scheduler is defined and the priority of an object that extends the Schedulable class can be set. However, even running the object with the highest priority will take some amount of time after it is enabled. The problem is even more challenging when the model contains more than one Urgent Synchronisation. We have not dealt with this feature as it did not occur in the gear-controller example.

Committed Locations: Committed Locations are urgent Locations that can not be delayed when they are enabled. Therefore, the discrepancy between model and code must be noted and analysed. This feature occurred in one Automaton, `Controller`, of our example [13]. The RTSJ code for this Automaton is available online [20].

Clocks: Each instance of a clock in a Timed Automaton is mapped to a clock in RTSJ. To check an upper- or lower-bound on a clock, a relative time is declared in Java for each bound. In addition, every thread contains an absolute time and a clock. To check the time elapsed from a particular moment, the absolute time is set to the current value of the clock. Then, the difference between the current value of the clock and the absolute time will be checked with the corresponding relative time. For example, a clock, two relative times and an absolute time are introduced for timing issues in Figure 3 (lines 4-7). In the Clutch Timed Automaton, when the transition from `Closing` to `Opening` is taken, the clock will be reset. The corresponding code for this action is shown in line 14, in which the absolute time `CurrentTime` is set to the current value of the clock `ClutchClock`. Therefore, the time elapsed from this moment will be measured. Inside the `else` statement, the program checks if the time is more than $100\mu s$ (line 25).

Guards: A transition from one state to another state can be taken if and only if the guard on the transition is enabled. A Guard is translated to an `if` statement. The code inside the `if` block corresponds to the transition updates (assignments). In Timed Automata constraints on the value of the clocks or

clock differences are only compared to integer expressions; guards over clocks are essentially conjunctions [19]. Line 20 in Figure 3 indicates that if the time elapsed since the last time at which the variable `CurrentTime` is set is equal to or more than the relative time `MaxTimeClutch`, 150, the `ErrStat` will be set to 2 and the `state` will be set to `ErrorOpen`. However, in the Timed Automaton, the guard on the transition from the `Opening` to the `ErrorOpen` is `Timer==150` and not `Timer >= 150`. Since there is no guarantee that the thread will execute this code at exactly one time, we need to be more flexible in the code than in the model. However, in this case, rather than noting and analysing the difference between model and code, we can actually modify (re-engineer) the model to match the code and then repeat the analysis of the properties we want to check on the modified model.

Dealing with non-determinism: A Timed Automaton can contain more than one transition with an enabled guard. If a state contains more than one outgoing edge with an enabled guard, one of them will be taken non-deterministically. Following the standard notion of refinement, an implementation can be more deterministic than the model. However, if we want to implement the non-determinism we can use a random variable in RTSJ. This feature occurred in one Automaton, `Interface`, of our example [13]. The RTSJ code for this Automaton is available online [20].

Invariants: The Automaton needs to leave a state before its state invariant is violated. In other words, Timed Automaton must take one of the enabled transitions if the current state invariant is violated. In RTSJ we cannot guarantee that the thread has a CPU before a certain time limit. However, we can accumulate the upper-bound of the run time of RTSJ code. To accumulate this run time upper-bound we can assign a fixed RTSJ run-time (based on the version of RTSJ and the hardware we use) to each line of code. We can then add these times to accumulate the run time of code corresponding to a state with an invariant. In Figure 1, the `Opening` has a state invariant, `Timer<=150`. Therefore, the clutch cannot stay in the `Opening` more than 150 ms and it needs to go to either `ErrOpen` or `Open` before this invariant is violated. This invariant is an assumption that should independently verified for a particular hardware and version of the RTSJ to check opening the clutch should take no more than 150 ms. In other words, executing the code in lines 14-16, 19-20 and 25 or lines 14-16 and 19-21 in Figure 3 should take less than $150\mu s$.

4 Verification result

To illustrate the applicability of the proposed method, we applied this approach to the gear-controller [13]. The model presented by Lindahl et al. contains 5 Timed Automata with a total of 63 states and 83 transitions. We recreated this

model and verified it with UPPAAL. However, the verification results were not entirely consistent with the result provided by Lindahl et al. [13]. We had to add the timing invariants on all states and increase the time bounds in the timing properties to satisfy them. The RTSJ derived from the Timed Automaton had 5 Java threads, 1320 lines of code and 16 assumptions for 16 time invariants in the model. The Timed Automata for the gear controller and the RTSJ code are both available online [20].

We unintentionally made an error in the UPPAAL model (when the clutch Automaton transitions from `Opening` to `ErrOpen`, we did not set `ErrStat` to 2). As a result, one of the system properties was violated. This property required the gear controller to notice that the clutch reached `ErrOpen`, before $300\mu s$. UPPAAL detected this error and we fixed the model. We wanted to see whether the same error would be detected in RTSJ. Therefore, we removed line 22 from Figure 3. However, the error was not detected since the offending code was not executed in the simulator as the timer never exceeds $150\mu s$. Then, we changed the invariant on `Opening` from $150\mu s$ to $50\mu s$ (line 20) and the error was detected. This shows why the model-checking approach is useful, as it detected an error in the model that is more difficult to detect in the code.

To demonstrate the discrepancy between the models, in which we make assumptions about invariants, and the code, where lines of code take a certain amount of time to execute, we changed the timing in both the model and implementation. In the original model, the invariant on the `Opening` state is $150\mu s$ and the transaction to `Open` can only be made after $100\mu s$. These properties are used to prove that if the clutch transits to `ErrOpen`, then the gear controller notices this before $300\mu s$. We changed the invariant $150\mu s$ to $2\mu s$ and the guard on transition to `Open` to $1\mu s$. In the model this is still sufficient to prove the gear controller notices the error before $4\mu s$. However, if we make these changes in the code, then the error is only detected after $50\mu s$.

5 Conclusion

In this paper, a nontrivial real-time example, a gear controller, was used to investigate a model-based approach to derive an RTSJ program from an UPPAAL model. We started from an existing UPPAAL Timed Automata for the gear controller, model checked it and followed our systematic approach to derive an RTSJ program from it. However, this approach has some limitations. As an example, when the model contains specific time values, rather than upper- or lower-bounds, it cannot be straightforwardly mapped to RTSJ code. Some other features, such as urgent synchronisation, were also left for future work.

References

1. Andy Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley and Sons Ltd, 2004.
2. Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
3. Sergio Yovine. Kronos: a verification tool for real-time systems. *Journal on Software Tools for Technology Transfer*, 1(1–2):123–133, 1997.
4. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.
5. Stavros Tripakis and Costas Courcoubetis. Extending Promela and Spin for real time. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 329–348. Kluwer Academic Publishers, 1996.
6. Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in model checking. In *Computer Aided Verification*, pages 521–525. Kluwer Academic Publishers, 1998.
7. Java SE real-time system - evaluation downloads. `http://java.sun.com/javase/technologies/realtime/rts/`. Date accessed: 20 August 2007.
8. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - a second generation of a Java model checker. In *Proceedings of the Workshop on Advances in Verification*, pages 164–169. World Scientific Publishing Company, 2000.
9. Gary Lindstrom, Peter Mehlitz, and Willem Visser. Model checking real time Java using Java PathFinder. In *3rd Int'l Symposium on Automated Technology for Verification and Analysis*, pages 444–456. Springer-Verlag, 2005.
10. Jeff Magee and Jeff Kramer. *Concurrency State Models and Java Programs*. John Wiley and Sons Ltd, 2005.
11. Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets. W. Reisig and G. Rozenberg (eds.), LNCS 3098*, pages 89–90. Springer-Verlag, 2004.
12. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
13. Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller: an industrial case study using UPPAAL. Technical Report ASTEC 97/09, Advanced Software Technology, Uppsala University, 1997.
14. C. Ramchandani. Analysis of asynchronous concurrent systems by Timed Petri Nets. Technical Report TR120, MIT (Massachusetts Institute of Technology), 1974.
15. Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using Time Petri Nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, 1991.
16. Anton Wijs. Achieving discrete relative timing with untimed process algebra. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 35–46. IEEE Computer Society, 2007.
17. Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *IEEE Symposium on Foundations of Computer Science*, pages 164–169. IEEE Computer Society, 1989.
18. Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, 2005.
19. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf, 2004. Date accessed: 15 May 2007.
20. Niusha Hakimipour's home page. http://itee.uq.edu.au/∼niusha/GearControler.rar. Date accessed: 1 May 2008.