Testing of Timing Properties in Real-Time Systems: Verifying Clock Constraints

Mehrdad Saadatmand
XDIN AB
Stockholm, Sweden
mehrdad.saadatmand@xdin.com

Mikael Sjödin
Mälardalen Real-Time Research Centre (MRTC)
Mälardalen University, Västerås, Sweden
mikael.sjodin@mdh.se

***Abstract*—Ensuring that timing constraints in a real-time system are satisfied and met is of utmost importance. There are different static analysis methods that are introduced to statically evaluate the correctness of such systems in terms of timing properties, such as schedulability analysis techniques. Regardless of the fact that some of these techniques might be too pessimistic or hard to apply in practice, there are also situations that can still occur at runtime resulting in the violation of timing properties and thus invalidation of the static analyses' results. Therefore, it is important to be able to test the runtime behavior of a real-time system with respect to its timing properties. In this paper, we introduce an approach for testing the timing properties of real-time systems focusing on their internal clock constraints. For this purpose, test cases are generated from timed automata models that describe the timing behavior of real-time tasks. The ultimate goal is to verify that the actual timing behavior of the system at runtime matches the timed automata models. This is achieved by tracking and time-measuring of state transitions at runtime.**

***Keywords*—Real-Time, Timing Properties, Testing, Runtime Verification, Temporal Correctness, Model-Based Testing.**

I. Introduction

In building real-time systems, it is very important to ensure that timing properties are in accordance with the specified timing requirements and constraints. The correctness of these systems is not only dependent on the correctness of the logical results of computations, but also on the time at which the results are produced [1]. The criticality of this issue can vary from one real-time system to another, such as in a real-time media player vs. the airbag system in an automobile. Different methods have been suggested in order to verify the correctness of timing properties in real-time systems. For example, there are diffeent schedulability analysis methods [2] that help to determine whether a set of real-time tasks are schedulable or not. There are several assumptions that are taken into account in performing such analyses, for instance, Worst-Case Execution Times (WCETs) of tasks. Some of the approaches for determining execution times can result in assigning very pessimistic values [3]. Moreover, at runtime, situations may occur that lead to the violation of the assumptions that were taken into account for performing the analyses, and thus invalidation of the analysis results [4], [5]. It should also be noted that for complex systems, such as those in telecommunication domain with huge number of concurrent tasks handling big loads of calls, data connections, billing, routing, and so on, it can be very hard in practice to get such detailed information about each task in the whole system in order to perform schedulability analysis [4], [6].

In this paper, we introduce an approach to test the timing behavior of real-time systems at runtime. The main intention is to basically verify that the timing properties of the tasks constituting the system match the specifications by testing the running system. This is achieved in our approach by consulting the timed automata models [7]–[9] of the system and automatically generating test cases from them[1]. Timed automata models representing the timing constraints are used as the source for the generation of test cases and determining pass/fail criteria for them. The test cases when executed against the running system determine whether the observed timing behaviors match what is specified for the tasks in the timed automata models or not. The focus in this work is mainly on the verification of the clock constraints that are specified in the models.

While most of the methods for verification of timing properties mainly target development phases before the actual execution of a real-time system, such as static analysis and model checking methods [1], our approach provides a way to dynamically test and verify the actual running system. This is especially important to alleviate the issues mentioned above, particularly scenarios which may occur at runtime that can cause invalidation of the results of static analysis methods, and to identify such misbehaviors. On the other hand, the approach can also be very well used to complement static analysis methods to gain more confidence in the correctness of designed systems in terms of their timing properties. We demonstrate the applicability of our approach by using it for testing of timing properties in a Brake-By-Wire (BBW) system from automotive industry. As the platform for implementation of the system, we have used OSE Real-Time Operating System

[1]In this paper, the term *verify* is used as its ordinary meaning in English and not to refer to 'formal verification' in software engineering, unless explicitly stated. Moreover, the term *state machine* is used as a synonym to refer to a timed automaton whenever the main concern is only the states and transitions in the model regardless of the timing specifications.

(RTOS) [10], on top of which, BBW application is developed in C/C++.

The remainder of the paper is structured as follows. In Section II, background information about the used technologies and the BBW system is provided. Section III describes the details of the proposed approach and in Section IV, we discuss how we have implemented the approach and applied it on the BBW system. In Section V, related works are discussed and possible scenarios for combination of our approach with those works are also identified. Finally in Section VI, a summary of the paper along with highlights and conclusions are provided.

II. Background Context

A. Timed Automata

Timed Automata (TA) are essentially finite state machines which are annotated and extended with real-valued clocks [7]–[9]. The clocks, initially set to zero at system start-up, progress and increase synchronously and at the same rate. The value of clocks can also be reset if needed. Timed automata are used to provide an abstract model of real-time systems. Timing constraints are specified using clocks whose values can be checked in the form of guards and invariants. Invariants can be regarded as progress conditions and are used to restrict the way that time may elapse at a state (location). For example, using invariants, it can be specified that the system is not allowed to stay in a state more than some time units and the transition has to be taken by the specified amount of time. Guards are specified on a transition (edge) as conditions to restrict its temporal occurrence.

Figure 1 shows an example timed automata for modeling a real-time lamp introduced in [8]. In this system, there is a timing requirement on how the user presses a button. The action of the user in pressing the button is modeled with the right-hand automaton. The timing requirement in this example states that if the user presses the button, the lamp is switched off or on (on in the low mode). However, if he is fast enough in pressing the button again, the lamp is switched on and ends up in the bright mode. The decision whether the user has been fast in pressing the button or not is determined by using the clock *y*.

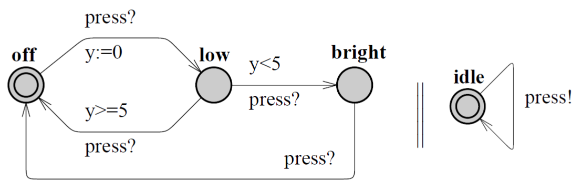


Fig. 1. TA model of a real-time lamp [8]

B. OSE RTOS & Farkle

OSE is a commercial and industrial real-time operating system developed by Enea [10] which has been designed from the ground specifically for fault-tolerant and distributed systems. It provides preemptive priority-based scheduling of tasks. OSE offers the concept of direct and asynchronous message passing for communication and synchronization between tasks, and OSE's natural programming model is based on this concept. Linx, which is the Inter-Process Communication protocol (IPC) in OSE, allows tasks to run on different processors or cores, utilizing the same message-based communication model as on a single processor. This programing model provides the advantage of not needing to use shared memory among tasks. The runnable real-time entity equivalent to a task is called *process* in OSE, and the messages that are passed between processes are referred to as *signals* (thus, the terms process and task in this paper can be considered interchangeably). Processes can be created statically at system start-up, or dynamically at runtime by other processes. Static processes last for the whole life time of the system and cannot be terminated. Types of processes that can be created in OSE are: interrupt process, timer interrupt process, prioritized process, background process, and phantom process. A process can be in one of the following states: ready, running or waiting. One interesting feature of OSE is that the same programming model based on signal passing can be used regardless of the type of process.

Farkle is a test execution framework that is originally developed for testing systems built using OSE. It enables testing embedded systems in their target environments. This capability has become possible by using the signal passing mechanism of OSE which allows Farkle to run on a host machine and communicate with the target by passing signals. In other words, Farkle basically enables testing an embedded system by providing certain inputs to the target in the form of signals and receiving the result as signals containing output values. The test scripts that are used to send and receive signals, and also decide the verdict of test cases are implemented in Python. Figure 2 provides an overall idea on how Farkle works.

C. Brake-by-Wire System

The Brake-by-Wire (BBW) is a braking system in which mechanical parts are replaced by electronic sensors and actuators and thus removing the hydraulic connections between the brake pedal and each wheel brake. Anti-lock Braking System (ABS) is usually an inherent functionality provided by BBW systems [11]. The purpose with the ABS subsystem is to prevent locking of the wheels by controlling braking based on calculated *slip rate* value. Slip rate is calculated according to the following formula (where *r* is the radius of the wheel):

$$s = (vehicleSpeed - wheelSpeed * r)/vechileSpeed$$

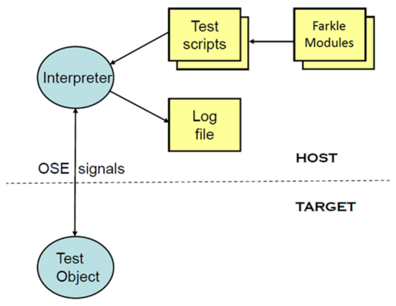


Fig. 2. Farkle test execution environment.

If *s* is greater than a certrain limit, then the brake actuator is released and no brake is applied, otherwise the requested brake torque is used. The Electronic Control Unit (ECU) for each wheel will have three application software components: one is a sensor to measure the wheel speed, one is an actuator for the brake, and the third one implements the ABS controller for the wheel. A schematic view of the BBW system components is shown in Figure 3 considering only one wheel for brevity.

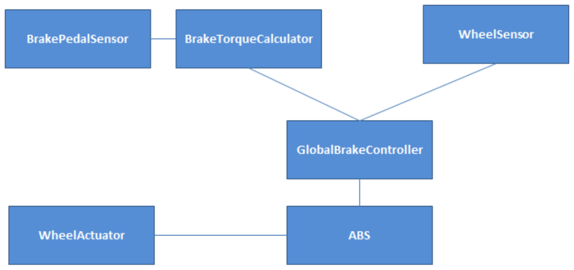


Fig. 3. Components composing a BBW system

There are several timing requirements in BBW systems. For example, the total brake reaction delay could be specified not to be more than 200ms (in a sample implementation of the BBW system), or requirements on the periods of samplings done by sensors. Generally, BBW is an example of safety-critical, distributed and real-time systems in which meeting timing requirements has also direct impacts on the safety of the system.

A Timed Automata (TA) model, designed in UPPAAL tool [8], describing the internal behavior of the ABS component of BBW system is shown in Figure 4. In this model, *y* is a clock whose specification on the states indicates the amount of time units that can be spent in each state (non-deterministically, between 0 and the specified value) before a transition has to be made to the next state. These timing specifications are naturally derived from high level timing requirements of the BBW system and its components. The values in the TA model here are just samples, and the exact values for each implementation of the BBW system might be different.

III. Proposed Approach

In this section we describe our proposed approach to generate test cases in order to verify that the timing properties of the system at runtime match the clock constraints specified in the timed automata models of the system. This is made possible basically by annotating the code so that state changes can be determined and tracked at runtime, and measuring the time difference between the state changes. The following are the steps that constitute the approach:

1) Based on the automata models, C/C++ enumerations (enum) that represent each state machine and their internal states are generated. These enumeration structures are stored in a C/C++ file along with a helper function called `set_state_wtime(StateMachine,State)`. The file is then included in the implementation code of the target application (i.e., to be tested).
2) The states in the timed automata model are mapped to the code using the above helper function. This is done by adding calls to the `set_state_wtime()` helper function at places where a state change occurs in the code. The helper function basically logs the new state belonging to the specified state machine along with the time stamp at which the transition and change to the new state has occurred.
3) According to the timing specifications in the timed automata model, a test script is generated which verifies the measured time difference between (pairs of) consecutive states against the model. In other words, if the time difference and also the order of state changes match the model, then the result of the test is determined as *pass*, otherwise a *fail* verdict is decided.

The generation of executable test scripts described in Step 3 is done in the following way:

- Minimum number of paths covering all clock constraints in the timed automata model are identified (clock constraint coverage). For example, in case of the TA model of the ABS component shown in Figure 4, the following paths are selected: $Entry \rightarrow CheckSpeed \rightarrow AsigT1 \rightarrow Exit$ and $Entry \rightarrow CalcSlipRate \rightarrow AsigT2 \rightarrow AsigT3 \rightarrow Exit$.
- For each of the identified paths a test script is generated. The script basically provides and sends necessary input(s) causing the state machine to start from the *Entry* state, taking the states and transitions constituting the selected path until reaching the *Exit* state. Considering the example in Figure 4, this means providing a value

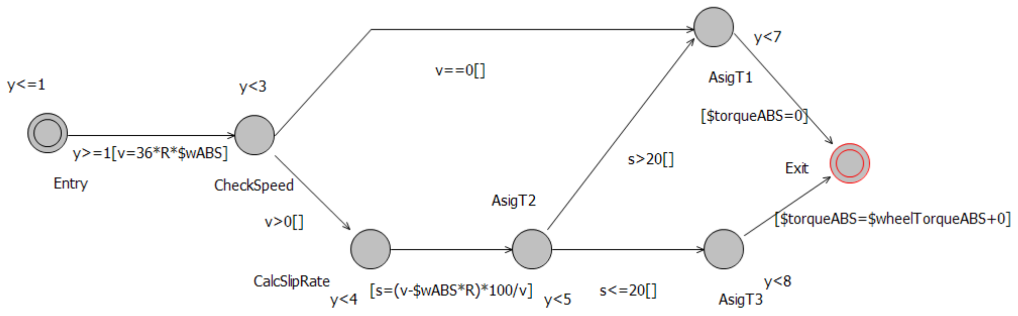


Fig. 4. Timed automata model of the ABS component

for *wABS* variable resulting in a value for *V* causing the desired transition and path to be taken. Based on the log information generated by the helper function, the script checks whether the order of the states and the time spent in each state with a clock constraint match the extracted path from the timed automata or not.

IV. Application & Implementation of the Approach

We have implemented the BBW system on OSE 5.5.1. An OSE process is created and developed for each component shown in Figure 3, and the communication between them is implemented by defining and passing appropriate OSE signals. For example, the signal definition for passing wheel speed information between processes is shown in Listing 1.

```
#define WHEEL_SPEED_SIG 1026
typedef struct WheelSpeedSignal{
  SIGSELECT sigNo;
  float WheelSpeed;
} WheelSpeedSignal;
```

Listing 1. Signal definition for wheel speed

To test and verify the clock constraints in the running system, the approach described in the previous section is followed. First information about different states is extracted from the timed automata models and enumeration structures (C/C++ enums) representing them are automatically generated:

```
enum StateMachines {BrakePedalSensor, BrakeTorqueCalculator
     , GlobalBrakeController, WheelActuator, ABS,
     WheelSensor};
enum States {Entry, CheckSpeed, CalcSlipRate, AsigT1,
     AsigT2, AsigT3, Exit, Brake, NoBrake, Reac, ...};
```

The result of this step is a C/C++ file containing the generated data structures along with the aforementioned helper function (that has a fixed implementation), which is then included in the implementation code of the BBW-ABS system. The next step is to map the states in the timed automata models to the code by adding calls to the helper function. The result is depicted in Figure 5 in case of the ABS process.

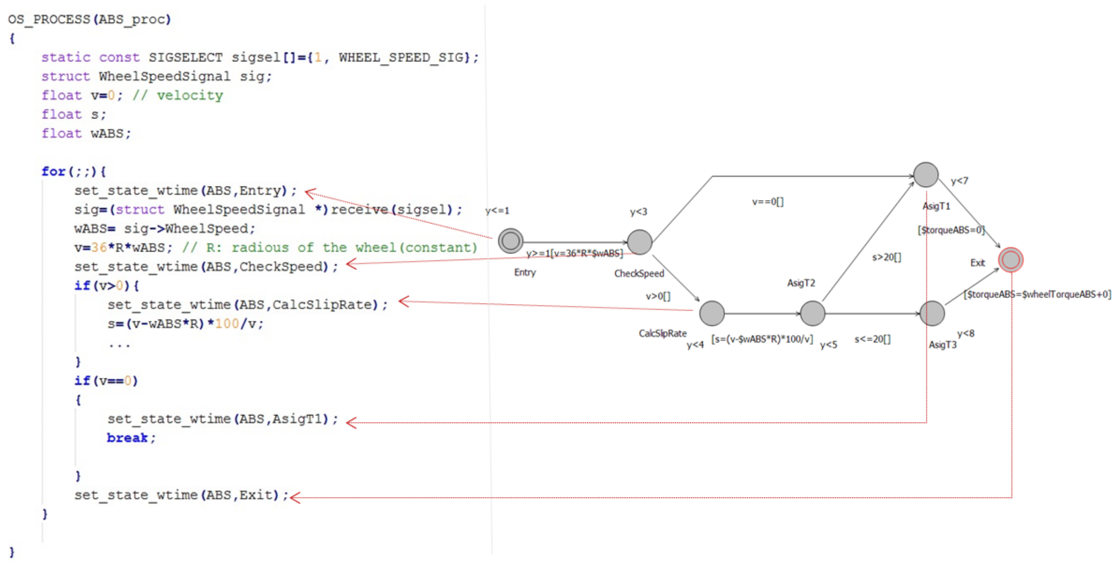
This is the only manual step in the whole approach. This step may also be automated if in a model-driven development methodology, for example, the code is generated taking into account the timed automata models of the system and thus aware of different states and transitions.

The final step is the generation of executable test scripts. For each identified path for covering the clocks constraints in the model, a test script is generated which, by investigating the generated log information about state changes, verifies that:

1) the order of visited states is in accordance with the timed automata model,
2) the amount of time spent in each state matches the timing specifications in the timed automata model. This is done by consulting the time stamps associated with and logged for each state change.

These details (order of states and timing specifications) are inherent and present in the timed automata models and are extracted from them in generating test scripts.

The test scripts are generated in the form of Python scripts which are then executed by the Farkle test execution framework (described in Section II-B). Farkle which runs on the host machine enables test scripts to communicate with the target system. The scripts send signals to the BBW processes running on the target. These signals (e.g., wheel speed signal in Listing 1) contain input values which are received, extracted and acted upon by the recipient process. In terms of state machines in the models, passing these input values invoke transitions in the recipient process's internal states. Whenever during the execution of a process the `set_state_wtime(StateMachine,State)` helper function (added during the mapping step) is called, a log record is created for tracking the states and the time point that a state change has occurred. This is used by test scripts to determine test verdicts. Finally, the *pass* or *fail* results



```
OS_PROCESS(ABS_proc)
{
    static const SIGSELECT sigsel[]={1, WHEEL_SPEED_SIG};
    struct WheelSpeedSignal sig;
    float v=0; // velocity
    float s;
    float wABS;

    for(;;){
        set_state_wtime(ABS,Entry);
        sig=(struct WheelSpeedSignal *)receive(sigsel);
        wABS= sig->WheelSpeed;
        v=36*R*wABS; // R: radious of the wheel(constant)
        set_state_wtime(ABS,CheckSpeed);
        if(v>0){
            set_state_wtime(ABS,CalcSlipRate);
            s=(v-wABS*R)*100/v;
            ...
        }
        if(v==0)
        {
            set_state_wtime(ABS,AsigT1);
            break;

        }
        set_state_wtime(ABS,Exit);
    }

}
```

Fig. 5. Mapping of states to the code (ABS function)

returned by executing the test scripts show whether the actual behavior of the system at runtime has been in accordance with and respecting the constraints specified in the timed automata models or not.

V. Related Work

In [12], we have previously introduced an approach for monitoring of timing properties of real-time tasks at runtime by enriching schedulers with the necessary monitoring mechanisms. The approach does not only enable provision of information about timing properties such as actual execution time (vs. estimated WCET), response time, deadline misses, and observed period and Minimum Inter-Arrival Time (MIAT) of tasks, but also helps to *preserve* and *enforce* such properties at runtime. In other words, if, for example, a task is taking too much time than its allowed time budget, this time budget is enforced by preempting the task, to let other tasks in the system perform as expected and pre-determined. Although in [12], we have not explicitly discussed and dealt with testing of such timing properties, the approach and the monitoring mechanisms introduced there can be regarded as the core functionality needed for dynamic testing of properties such as execution time of tasks, determining occurrence of deadline misses in the system, violation of period and MIAT values, and so on. In this paper, however, we more directly addressed testing of real-time systems. Moreover, we focused mainly on timing properties specifiable in the form of timed automata describing the internal behavior of real-time tasks, as well as the system in general (e.g., end-to-end deadlines). Extending the work done in [12] for testing purposes, and combining it with the approach suggested here to provide a comprehensive testing framework for timing properties is left as future directions of this work to investigate. Beside the monitoring method we have introduced in [12], we have also provided a survey of different available methods and tools for monitoring of timing constraints in [13].

UPPAAL is a tool suite for modeling and verification of real-time systems modeled as networks of timed automata [8], [14]. Three main parts that constitute UPPAAL are a description language, a simulator and a model-checker. UPPAAL is an example of prominent tools and methods for verification of real-time systems at the modeling level and does not concern implementation code and actual execution of the system. UPPAAL TRON [15], [16] is a testing tool based on the UPPAAL engine that is designed for black-box conformance testing of real-time systems. The testing approach provided by TRON is similar to ours in the sense that both communicate with and execute test cases against the running system, and also both use timed automata models. UPPAAL TRON derives, executes, and evaluates test cases against the implementation of the system in real-time. However, the main difference between the testing method of UPPAAL TRON and our approach is that UPPAAL TRON addresses online generation and execution of test cases based on the results of previous executed test cases, and focuses mainly on observable input and output actions considering the Implementation Under Test (IUT) as black-box and assuming that its internal states are not observable [16]. While in our approach, test cases are only generated offline, and also the IUT is considered as white-box whose internal state changes are fully tracked and compared against the TA models. The mapping of states to code in our approach is thus needed and introduced to enable this feature. Also what is tested in our approach is not the output from the IUT and the time at which it is produced, but the order of states and the time spent in each one which has a time constraint. From this perspective, there seems to be potentials for combining our testing approach with UPPAAL TRON. Shin, Drusinsky and Cook present in [17] an approach for white-box testing of timing properties. They introduce assertion state charts as a way to specify and keep track of timing constraints. From these state charts, some test cases are derived manually (e.g., testing single use case scenarios) and some are automatically generated (e.g., testing the state chart under test itself). Their approach is however closer to UPPAAL TRON with respect to what is actually testable and tested in the running system.

In [18], we have discussed the general idea of combining static analysis and testing. Its potentials and different combination scenarios along with an example are also provided in that work.

VI. CONCLUSION

In this paper, we introduced an approach for dynamic testing of timing properties in real-time systems. By tracking state changes at runtime, the approach allows for more detailed testing of timing properties of real-time systems and their tasks whose internal behaviors are represented in the form of timed automata. Testing and runtime verification of timing properties becomes especially important in cases where static analysis methods are hard to apply in practice or can be invalidated at runtime. In general, however, both approaches (static analysis methods and testing) can be considered complementary and used together to gain more confidence in temporal correctness of real-time systems.

Timed automata models are used in our approach as a representation and source for timing requirements and constraints from which test cases are automatically generated. We demonstrated the applicability of our approach on the ABS subsystem of Brake-By-Wire system, verifying the timing constraints specified on different states constituting its behavioral model. Also, as for the implementation of our approach, we used OSE as the core platform and real-time operating system, C/C++ for implementing the BBW application, and Python as the language for generating executable test scripts. Using OSE along with its test execution environment, Farkle, enables to test an embedded system in its target environment. This is a valuable feature considering that resource constraints originating from the execution environment (e.g., battery/power, layout and size, heat generation, available memory, CPU, etc.) affect and dictate, to a great degree, how an embedded system should be designed and perform. It should, however, be noted that the approach is not necessarily dependent on these technological choices and may be well implemented differently.

As mentioned in the paper, the mapping step of the approach, in which state changes are annotated and marked in the code using the helper function, is the only step which needs manual intervention. As a future work, we are working on solutions for automating this step, and thus automating the whole testing approach. Also, in this work, since we were only interested in the time difference between state changes, the execution time of the added calls to the helper function (to log and timestamp state changes) had no effect on the test result and was simply ignored. However, if, for example, end-to-end response times are to be tested, it is important to take into account the added timing-cost of the helper function. Considering that the helper function has a fixed execution time, though, its impacts on the execution time of a task to which it is added may be easily predicted and reduced from the task's total execution time. Also, it might be possible to claim that in some systems, if tests are passed while having helper function calls in the code, they might also work fine in terms of timing properties when the helper function calls are removed, e.g., in the release and final version of a product. Careful investigation of such claims and scenarios, and side effects of adding helper functions to enable testing, as well as different ways to mitigate them are left as other future directions of this work.

VII. ACKNOWLEDGEMENTS

This work has been supported by the MBAT European Project [19] and Xdin Stockholm AB (formerly Enea Services Stockholm AB) [20] through the ITS-EASY industrial research school [21]. The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no 269335 (see Article II.9. of the JU Grant Agreement) and from the Swedish Governmental Agency for Innovation Systems (VINNOVA). We would also like to thank Raluca Marinescu and Dr. Cristina Seceleanu for their technical tips and support for this work.

REFERENCES

[1] J. Wegener and M. Grochtmann, "Verifying timing constraints of real-time systems by means of evolutionary testing," *Real-Time Syst.*, vol. 15, no. 3, pp. 275–298, Nov. 1998. [Online]. Available: http://dx.doi.org/10.1023/A:1008096431840

[2] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/1978802.1978814

[3] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem-overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1347375.1347389

[4] M. Saadatmand, A. Cicchetti, and M. Sjödin, "Design of adaptive security mechanisms for real-time embedded systems," in *Proceedings of the 4th international conference on Engineering Secure Software and Systems*, ser. ESSoS'12. Eindhoven, The Netherlands: Springer-Verlag, 2012, pp. 121–134.

[5] S. Chodrow, F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," in *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, dec 1991, pp. 74 –83.

[6] M. Saadatmand, A. Cicchetti, and M. Sjödin, "Uml-based modeling of non-functional requirements in telecommunication systems," in *The Sixth International Conference on Software Engineering Advances (ICSEA)*, October 2011.

[7] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994.

[8] G. Behrmann, R. David, and K. G. Larsen, "A tutorial on Uppaal 4.0," http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf, November 2006.

[9] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *In Lecture Notes on Concurrency and Petri Nets*, ser. Lecture Notes in Computer Science vol 3098, W. Reisig and G. Rozenberg, Eds. Springer–Verlag, 2004.

[10] Enea, http://www.enea.com, Last Accessed: June 2013.

[11] S. Anwar, "An anti-lock braking control system for a hybrid electromagnetic/electrohydraulic brake-by-wire system," in *American Control Conference, 2004. Proceedings of the 2004*, vol. 3, 2004, pp. 2699–2704 vol.3.

[12] M. Saadatmand, M. Sjodin, and N. Mustafa, "Monitoring capabilities of schedulers in model-driven development of real-time systems," in *17th IEEE Conference on Emerging Technologies Factory Automation (ETFA)*, Krakow, Poland, 2012, pp. 1–10.

[13] N. Asadi, M. Saadatmand, and M. Sjödin, "Run-time monitoring of timing constraints: A survey of methods and tools," in *The Eighth International Conference on Software Engineering Advances (ICSEA)*, Venice, Italy, October 2013.

[14] Uppaal, http://www.uppaal.org/, Accessed: August 2013.

[15] Uppaal for Testing Real-Time Systems Online (TRON), http://people.cs.aau.dk/~marius/tron/, Accessed: August 2013.

[16] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using uppaal-tron: an industrial case study," in *Proceedings of the 5th ACM international conference on Embedded software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 299–306.

[17] M.-T. Shing, D. Drusinsky, and T. Cook, "Quality assurance of the timing properties of real-time, reactive system-of-systems," in *2006 IEEE/SMC International Conference on System of Systems Engineering*, 2006, pp. 6 pp.–.

[18] M. Saadatmand and M. Sjödin, "On combining model-based analysis and testing," in *10th International Conference on Information Technology : New Generations (ITNG 2013)*, Las Vegas, NV, USA, April 2013.

[19] MBAT Project: Combined Model-based Analysis and Testing of Embedded Systems, http://www.mbat-artemis.eu/home/, Accessed: June 2013.

[20] XDIN AB, http://xdin.com/en/about-xdin/enea-experts/, Accessed: June 2013.

[21] ITS-EASY post graduate industrial research school for embedded software and systems, http://www.mrtc.mdh.se/projects/itseasy/, Accessed: June 2013.