martProject no.: **FP7-ICT-STREP-214755**

Project full title: **Quantitative System Properties in Model-Driven Design**

Project Acronym: **QUASIMODO**

Deliverable no.: **D4.2**

Title of the deliverable: Algorithms for off- and online quantitative testing

Contractual Date of Delivery to the CEC:	**M24**
Actual Date of Delivery to the CEC:	**M24**
Organisation name of lead contractor for this deliverable:	**AAU**
Author(s): Brian Nielsen	
Participants(s): AAU, RWTH, ESI, ESI/Twente, RWTH	
Work package contributing to the deliverable:	**WP4: Testing**
Nature:	**R+P**
Version:	**0.1**
Total number of pages:	**15**
Start date of project:	1st January 2008 **Duration:** 36 month

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)
Dissemination Level

PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract:

This deliverable presents the results on algorithms for quantitative offline and online testing of real-time systems.

Keyword list: real-time systems, blackbox testing, conformance, model based development, non-determinism.

Table of Contents

Abbreviations

- **SUT** System Under Test, includes implementation under test and test interface / test context
- **IUT** Implementation Under Test
- **UML** Unified Modelling Language
- **ACTL** action computation tree logic

Executive Summary

Model-based testing is an innovative approach that has high potential to improve the quality and efficiency of the testing process. This deliverable presents the results on algorithms for offline and online testing of real-time systems. We show how model-checking tools and model-checking algorithms can be used to generate timed test cases, but also that these techniques are insufficient to deal with the non-determinism, observation uncertainties, approximations, and infinite states required by real-time systems. We show how timed games may be a better metaphor and present our results on using timed game solving algorithms for timed test case generation. Alternatively online testing can be used. Our online testing algorithm use symbolic data-structures and algorithms to represent and explore the current set of possible reachable states given the current observed timed trace. These algorithms are an non-trivial extension of the symbolic algorithms in the Uppaal-model-checker. We conclude that online real-time testing is feasible for many real-time systems.

1 Introduction

1.1 Context

Testing aims at checking whether the behaviour of a (physical) system under test is correct with respect to (conforms-to) its specification. Formal model-based testing [1,2,3] is an innovative approach that has high potential to improve the quality and efficiency of the testing process. It allows for the generation of large amounts of correct and effective/covering test cases completely automatically from a model of required behavior. The test engineer can thus focus on specifying (via the model) *what* should be tested at a high level of abstraction, rather than at *how* by laboriously creating and maintaining test scripts (or worse as still is frequently the case in practice: manually executing these). In particular when requirements change, the effort of updating test cases/test scripts is reduced by automatically re-generated these to reflect the change.

As described in Annex-I, a goal of Quasimodo is to extend and improve the existing model-based testing techniques towards quantitative models. Quantitative testing is particularly challenging because it must deal with non-determinism, observation uncertainties, approximations, and infinite states.

A previous Quasimodo deliverable (D4.1) dealt with formalizing suitable notions of correctness (conformance relations) for quantitative systems. These are important not only from a theoretical viewpoint, but are also practically important because test generation algorithms and tools are directly based on those. A main challenge is to transform the theory and the test generation algorithms into implementable and efficient algorithms; in particular, this involves the development of symbolic methods to efficiently represent and manipulate the involved infinite domains (time, continuous data, and probabilities).

This deliverable focuses on presenting the results on algorithms for offline and online black-box conformance testing of real-time systems. Further advances will be reported in the future deliverables D4.3 (Test selection and coverage) and D4.5 (Final algorithms and evaluation), as well as in deliverables related to case studies. Our results for testing hybrid and stochastic models will be reported in the future deliverable D4.6, and similarly, results on approximate testing will be reported deliverable D4.4.

1.2 Testing Real-time systems

Timed conformance testing must evaluate the timeliness of the SUT, i.e. it must thus execute input stimuli with different timings and loads, and evaluate the correctness of the timed responses (as defined by the conformance relation). Thus, interactions and observations are timed i/o sequences. Testing real-time systems is challenging because

- real-time systems typically exhibit a lot of non-determinism in both specification models and in the actual implementations. The implemented systems are inherently concurrent because they need to control multiple simultaneous activities and detect and react to a multitude of events. Non-determinism also arise from the underlying hardware (pipelines, caches, bus-arbitration, etc) and internal scheduling and resource allocation decisions. At the model level, non-determinism is used to abstract away unknown implementation details or aspects, or are too complex to fully model. Test models thus use non-determinism to capture the permissible behaviour including the permissible time bounds on outputs. Finally, models of large and complex systems are constructed as parallel composition of models of its components.

- the tester makes imperfect observations. This concerns both limitations on the precision with which it measures/produces physical quantities (especially time), and the states/system actions that are observable/controllable to the tester in the first place.
- the tester must itself execute in real-time on physical hardware that rarely is fully predictable. Furthermore, the tester's and SUTs time base need to be synchronized.

This means that the tester in general cannot infer in which exact state it or the SUT is in. As consequence, the outputs, their exact order and timing are indeterminate and unpredictable. Therefore, test cases must branch depending on the actual outputs and their timing.

In the special case where the implementation is (or when it in practice can be viewed as) completely deterministic, it is fairly easy to use a model-checker like Uppaal to generate (using the counter-example mechanism) test cases of pre-computed fixed timed i/o sequences, that even exhibit some a-priory guarantees like model-coverage or time-optimality. We have done this in past works as summarized in [4].

1.3 Off-line and on-line testing

In offline test generation the test suite is pre-computed completely from the specification before it is executed (potentially automatically) on the implementation under test.

Another testing approach, see Figure 1, is online (on-the-fly) testing that interleaves test generation and execution. Here the test generator interactively interprets the model, and stimulates and observes the SUT. Only a single test input is generated from the model at a time which is then immediately executed on the SUT. Then the produced output (if any) by the SUT as well as its time of occurrence are checked against the specification, a new input is produced and so forth until it is decided to end the test, or an error is detected. Typically, the inputs and delays are chosen randomly from the model. An observed test run is a trace consisting of an alternating sequence of (input or output) actions and time delays.

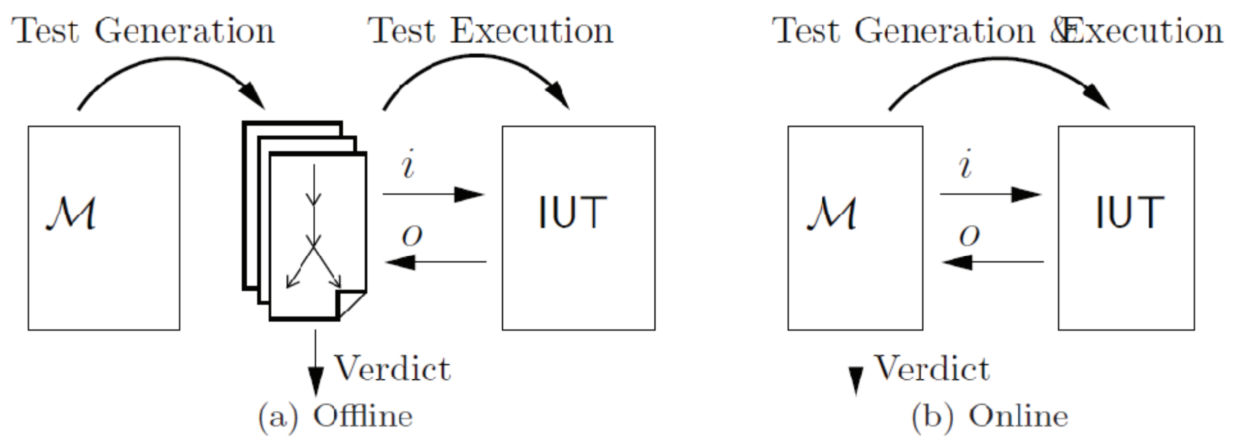


Figure 1 Offline and online testing

Table 1 summarizes the characteristics of online and offline testing. The advantages of offline test generation are that test cases are easier, cheaper, and faster to execute because all time constraints and choices in the specification have been resolved at test generation time, and in addition, that the test suite can be generated with some guarantees and optimization a-priori, e.g., that the specification is structurally covered with as few communications or resets as possible, or that a given set of test-purposes (observation objectives) are met as fast or with as few resources as possible.

There are two main disadvantages of offline test generation. One is that the specification must be analyzed in its entirety, which often results in a state-explosion which limits the size of the specification that can be handled. Another problem is non-deterministic implementations and specifications. In this case, the output (and output timing) cannot be predicted, and the test case must be adaptive. Typically, the test case takes the form of a test-tree that branches for all possible outcomes. This may lead to very large test cases. In particular for real-time systems the test case may need to branch for all time instances where an output could arrive.

Offline test generators therefore often limit the expressiveness and amount of non-determinism of the specification language. This has been a particular problem for offline test generation from timed automata specifications, because the technique of determinizing the specification cannot be directly applied.

There are several advantages of online testing. Testing may potentially continue for a long time (hours or even days), and therefore long, intricate test cases that stress the SUT may be executed. The state-space-explosion problem experienced by many offline test generation tools is reduced because only a limited part of the state-space needs to be stored at any point in time. Further, online test generators often allow more expressive specification languages, especially wrt. allowed non-determinism in real-time models: Since they are generated event-by-event they are automatically adaptive to the non-determinism of the specification and implementation. A disadvantage is that the specification must be analyzed online and in real-time which require very efficient test generation algorithms to keep up with the implementation and specified real-time requirements. Also the test runs are typically long, and consequently the cause of a test failure may be difficult to diagnose. Although some guidance is possible, test generation is typically randomized which means that satisfaction of coverage criteria cannot be a priory guaranteed, but must instead be evaluated post mortem.

Offline Test Generation	Online Testing
Fast and easy execution	State-exploration overhead at runtime
A-priory coverage guarantee possible	Coverage measured post-mortem
Optimized test suite possible	Deep testing, high-volume randomized testing
Costly/difficult generation computation	Larger models (State space explosion problem less frequent)
Restricted model expressiveness	Fewer restrictions of the model expressiveness
	Diagnostics/localizing error is difficult

Table 1 Characteristics of offline and online testing

2 Offline test generation using model-checking

2.1 *Testing Real-Time Systems Using UPPAAL*

Participants. Anders Hessel, Jacom Illum Rasmussen, Kim G. Larsen, Marius Mikučionis, Brian Nielsen, Paul Pettersson, Arne Skou

- Anders Hessel and Kim G. Larsen and Marius Mikučionis and Brian Nielsen and Paul Pettersson and Arne Skou**. Automated Model-Based Conformance Testing of Real-Time Systems**. Chapter in (Eds.) Jonathan Bowen, Mark Harman, and Rob Hierons: Formal Methods and Testing, Springer Verlag, LNCS 4949, 2008

In this work [4] we identify a restricted class of deterministic and output urgent timed automata (DOUTA) and we show how it is possible to use the unmodified Uppaal model-checker to generate timed test sequences. These may even be optimal (either shortest possible or taking least possible time to execute) or satisfy a given (user defined) coverage criteria of the model. We also define a language for defining test purposes and coverage criteria, and present an efficient test generation algorithm.

AAU with external collaborators have put same underlying approach into commercial industrial use. UML diagrams (of a subset of UML2) are translated to networks of timed automata. Uppaal-CORA is then used to generate test sequences. The cost-based branch-and-bound algorithms together with other dedicated search heuristics implemented in CORA have been the key to enable (guaranteed) coverage based test generation for large models. This approach has successfully been applied to commercial GUI-based testing of administrative systems as well as medical devices.

2.2 *An Alternative Approach*

The DOUTA assumption in [4] is frequently impractical or unrealistic for embedded real-time systems. However, offline test generation using a real-time model-checking engine for unrestricted timed automata models is still possible using the following procedure that separate test input generation and test evaluation in two different runs:

1. **Generate a preset timed input sequence** by using the model-checker to generate a concrete timed trace satisfying the test purpose, and project away outputs, internal actions, and sum up adjacent delays
2. **Execute the preset-timed input sequence on the SUT**, and record the resulting timed input/output sequence
3. **Check whether the observed trace is included in the specification model** (our rt-ioco conformance relation requires timed trace inclusion). This can most easily be done using the TRON engine. Alternatively the trace can be represented as a timed automata model which can be composed in parallel with the specification model (with some automated rewriting needed to turn observable channels into broadcasts) and then model–check that the final state of the trace automaton is reachable.
4. **Check that the test purpose is satisfied** (or sufficient coverage has been reached**)** by a similar mechanism as the previous step**.**

This procedure is practical and operational. However, in our view it is imperfect because it cannot guarantee a-priory that the test purpose will be satisfied on a correct implementation due to the non-adaptiveness of the preset input sequence. It many cases the procedure can be effective, but the extent of this problem depends on how much output uncertainty exists in the model and implementation.

3 Offline Testing using Timed Games

Participants: Alexandre David, Kim G. Larsen, Shuhao Li, Brian Nielsen

As previously discussed, an effective test case for real-time systems testing need to be adaptive to the system non-determinism, lack of observation precision, and partial observability and controllability. As a simple example consider Figure 2 that specifies the behavior of a Light Controller. In e.g., location L5 the Lamp is required to produce either a bright or a dimmed light, and moreover required to do so within 2 time units. Which output and at which time this occurs is unpredictable from the tester's point of view.

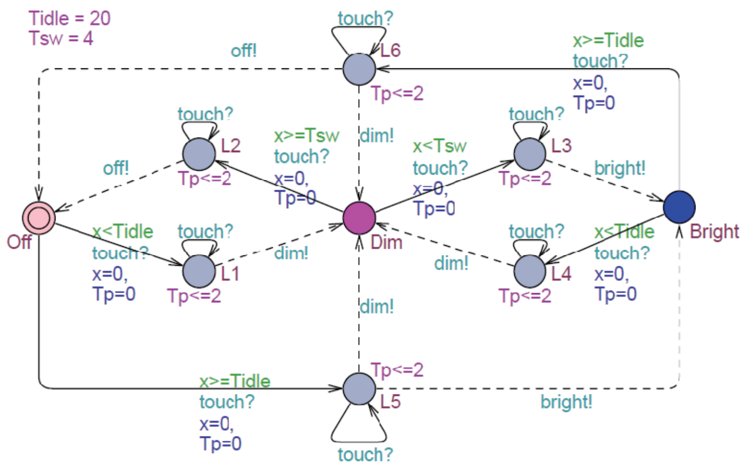


Figure 2 Timed Game Automaton specifying the behaviour of a Lamp Controller. Dashed edges are uncontrollable, solid ones are controllable

Generating a test case under these circumstances that satisfy a given test purpose thus resembles synthesizing an control strategy for an embedded controller, only now the SUT has the role of opponent, and the (winning) strategy guides the tester as what action to take on a given observation (system output) such that the tester will observe the test purpose on a correct implementation, see Figure 3. The idea is to formulate the specification as a timed game automaton where the actions that can be controlled by the tester (typically inputs) are mapped to controllable transitions; similarly outputs are mapped to uncontrollable transitions. Given a test-purpose, a timed game solver like Uppaal-TIGA can then be used to generate a winning strategy when one exists that can be used as the core of a test case.

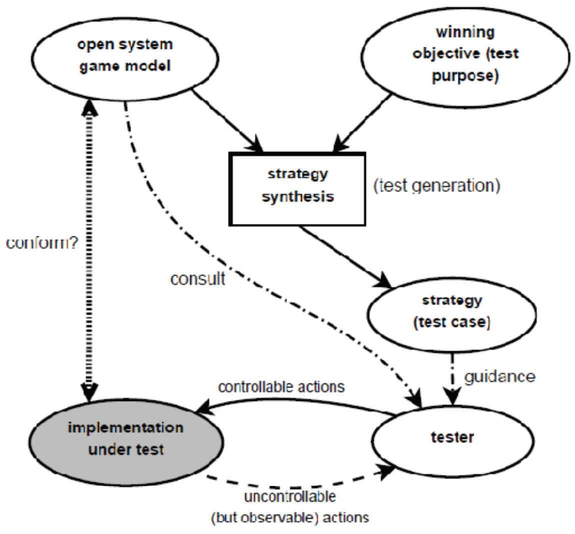


Figure 3 Using games for testing

3.1 A Game-Theoretic Approach to Real-Time System Testing

- Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen. A Game_Theoretic Approach to Real-Time System Testing In: Proc. 11th Conference on Design, Automation and Test in Europe (DATE'08).

In [5] we present a game-theoretic approach to the testing of real-time embedded systems whose models may have output uncertainty and timing uncertainty of outputs. By modeling a system in question using timed game automata (TGA) and specifying the test purpose as an ACTL formula, we employ a recently developed timed game solver Uppaal-Tiga to synthesize winning strategies, and then use these strategies to conduct black-box conformance testing of the system. We show how to execute winning strategies as test cases in the context of real-time conformance testing. The testing process is proved to be sound and complete with respect to the given test purpose. Case study and preliminary experimental results indicate that this is a viable approach to real-time embedded system testing.

3.2 Cooperative Testing of Timed Systems.

- Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen. **Cooperative Testing of Timed Systems.** In: Proc. 4th Workshop on Model-Based Testing (MBT'08), Budapest, Hungary, March 2008, ENTCS 200(1): 79-92.

This paper [6] deals with test purpose based testing of real-time embedded systems. The testing activity is viewed as a game between the tester and the system under test (SUT) towards a given test purpose (winning objective). The SUT is modeled using timed game automata (TGA) and the test purpose is specified as an ACTL formula. We employ a timed game solver Uppaal-Tiga to check if the timed game is solvable, and if yes, to generate a winning strategy and use it for blackbox conformance testing of the SUT.

Specifically, we show that in case the game solving yields a negative result, we can still possibly test the SUT against the test purpose. In this case, we use Uppaal-Tiga to generate a cooperative winning strategy. The testing process will continue as long as the SUT reacts to the tester stimuli in a cooperative manner. In this way, we can hopefully arrive at a certain state in the “surely winning" zone of the game state space, from which cooperation from SUT is no longer needed. We present an operational framework of cooperative winning strategy generation, test case derivation and test execution. The test method is proved to be sound and complete. Preliminary experimental results indicate that this approach is applicable to non-trivial timed systems.

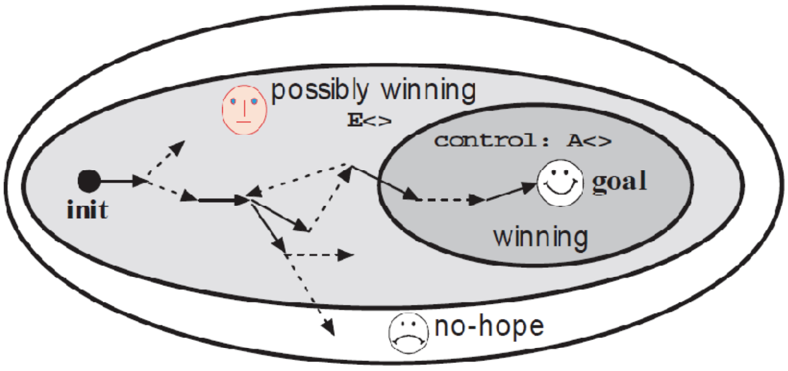


Figure 4 Testing using cooperative winning games:
state space partitioned into winning, loosing and possibly winning states.

To enable generation of cooperative strategies the core algorithms of Uppaal-TiGa has been extended with capabilities for generating the largest set of winning states.

This work also shows an interesting link between offline and online testing, as the cooperative part of the strategy can be executed in an online randomized fashion.

3.3 Timed Testing under Partial Observability.

- Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen: **Timed Testing under Partial Observability**. In: Proc. 2nd International Conference on Software Testing, Verification, and Validation (ICST'09), Denver, Colorado, April 2009.

This paper [7] studies the problem of model-based conformance testing of partially observable timed systems. We model the system under test (SUT) using timed game automaton (TGA) that has internal actions, output uncertainty and timing uncertainty of outputs. We define the partial observability of a SUT using a set of observable predicates over the TGA semantic state space, and specify the test purposes as ACTL logic formulas. A partially observable timed game solver Uppaal-Tiga is used to generate winning strategies, which are then used as test cases. We propose a conformance testing framework for this particular setting, define a partial observation-based conformance relation, present the test execution algorithms, and prove the soundness and completeness of this test method.

Experiments on some non-trivial examples show that this method yields encouraging results.

4 Online Real-Time Testing

Participants: Kim G. Larsen, Marius Mikučionis, Brian Nielsen

4.1 Symbolic Online Testing

Participants. Kim G. Larsen, Marius Mikučionis, Brian Nielsen

- Anders Hessel and Kim G. Larsen and Marius Mikučionis and Brian Nielsen and Paul Pettersson and Arne Skou**. Automated Model-Based Conformance Testing of Real-Time Systems**. Chapter in (Eds.) Jonathan Bowen, Mark Harman, and Rob Hierons: Formal Methods and Testing, Springer Verlag, LNCS 4949, 2008
- Marius Mikûcionis: Online testing of real-time systems. Phd Thesis. To appear 2010.

The details of this work is described in [4,8]. The system specification consists of a closed network of Uppaal timed automata that can be partitioned into an environment model and a SUT model, see Figure 5. The correctness relation is (environment) relativised real-time input/output conformance. Online (on-the-fly) testing interleaves test generation and execution such that computation of the next test event to be given to the SUT is performed while the SUT is running, and at the same time checking whether the outputs and timings of the SUT are permissible. The main idea is to dynamically and incrementally compute the set of possible states that the specification may occupy given the current observed timed input/output trace. If this state-set becomes empty, the trace is not included in the specification, and hence the implementation is not conforming.

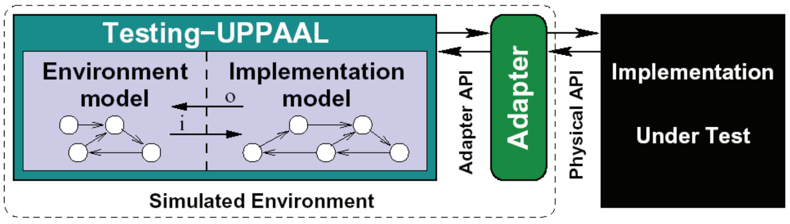
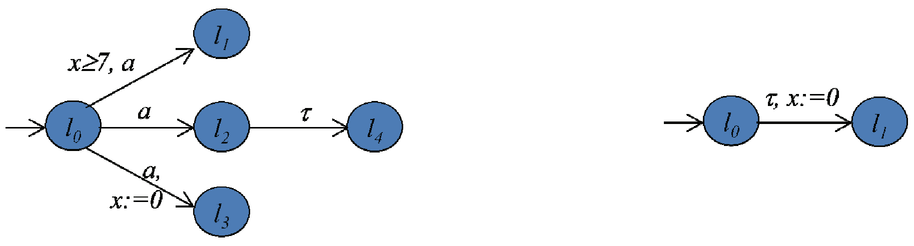


Figure 5 Uppaal-TRON testing setup

For timed automata with a dense model of time, this set of concrete states is infinite. A main challenge for real-time online testing is to represent this set of states by symbolic states and to compute and explore this set of symbolic states sufficiently efficiently to allow real-time performance. To see that a set of symbolic states is necessary consider Figure 6 that illustrates the states reachable after a given action and after a given delay, given a symbolic initial state.



$\{ \langle l_0, x=3 \rangle \}$ **after** a = $\{ \langle l_2, x=3 \rangle, \langle l_4, x=3 \rangle, \langle l_3, x=0 \rangle \}$

$\{ \langle l_0, x=0 \rangle \}$ **after** 4 = $\{ \langle l_0, x=4 \rangle, \langle l_1, 0 \leq x \leq 4 \rangle \}$

Figure 6 Timed automata models and symbolic state sets representing the possible reachable states after an observable action or (accumulated) delay.

The basic algorithm is shown in Figure 7. The algorithm is implemented by extending the Uppaal model-checking engine with a new zone operation for computing the absolute time bounded future, and with special filters for manipulating state-sets:

- `ActionChoices` computes the "menu" of enabled set of observable actions in a given state set.
- `Tau-closure` computes all states reachable via internal actions and a silent delay (interval).
- `AfterAction` computes all states reachable after a given observable action occurring at a specific time (interval).
- `AfterDelay` computes all states reachable after a given specific time delay.
- `MaxDelay` computes the maximum allowable delay before an input event must occur.

Although the basic algorithm was designed and implemented prior to Quasimodo several improvements have been made during the course of Quasimodo, such that it is now more efficient, correct and robust (timing wise):

- The state set exploration algorithms have been optimized, especially state set can now be represented using federations of zones. Further, the time horizon when computing the maximum delay may be bounded.
- Great care has been taking when events are time stamped to ensure that that the mapping of real-time occurrence to computer time stamp to model time is correct and robust.
- Time stamping of input and output actions is now done by an interval (a single time instant is supported as a special case). Thus the state-set is slightly over approximated reflecting the possible uncertainty of time stamping. This is of practical importance, especially for long running tests.
- The algorithms is "self monitoring" by checking whether test events were actually issued at the expected times as dictated by the model.
- The algorithm may also take into account the properties of the platform (e.g., reaction time when supplying a test input event).

- A virtual time clock framework has been developed. This is a kind of a clock synchronization algorithm that keep model and SUT time synchronized and that allow time to “fast forward”. It assumes that a test component on the SUT has some control over the clocks and timers of the SUT. We have developed test components for applications that use Posix threads synchronization or Java monitor primitives to manage time. When possible, this is highly useful in practice because it reduces the timing uncertainty and speeds up testing. Similarly, it is indispensible in an educational or demonstration situation where the test tool and SUT execute on the same host (thus competing for the same CPU making predictable real-time behavior problematic for both).
- An algorithm has been developed to automatically analyze the last known state set to provide diagnostic information, e.g., whether the error is caused by an output or timing violation, whether the assumptions of input enabledness of the environment or SUT models have been violated, or whether a real-time execution failure occurred that violated the environment model. For instance, if an output (or timing) violation occurred the algorithm lists the set of acceptable output actions (or timing delay).
- Several improvements have been made to the adaptor framework. A Users Manual has been written.

Algorithm 2. Test generation and execution: $TestGenExe(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T)$.

```
$\mathcal{Z} := \{(s_0, e_0)\}$;            // initialize the state set with initial state
while $\mathcal{Z} \neq \emptyset \wedge \sharp iterations \leq T$ do
    switch between action, delay and restart randomly do
        case action:                                        // offer an input
            if $\mathsf{EnvOutput}(\mathcal{Z}) \neq \emptyset$ then
                randomly choose $i \in \mathsf{EnvOutput}(\mathcal{Z})$;
                send $i$ to IUT,;
                $\mathcal{Z} := \mathcal{Z}$ After $i$;
        case delay:                                         // wait for an output
            randomly choose $d \in \mathsf{Delays}(\mathcal{Z})$;
            sleep for $d$ time units or wake up on output $o$ at $d' \leq d$;
            if $o$ occurs then
                $\mathcal{Z} := \mathcal{Z}$ After $d'$;
                if $o \notin \mathsf{ImpOutput}(\mathcal{Z})$ then return fail ;
                else $\mathcal{Z} := \mathcal{Z}$ After $o$
            else $\mathcal{Z} := \mathcal{Z}$ After $d$;             // no output within $d$ delay
        case restart: $\mathcal{Z} := \{(s_0, e_0)\}$; reset IUT;        // reset and restart
if $\mathcal{Z} = \emptyset$ then return fail else return pass;
```

Figure 7 Basic online testing algorithm.

```
Algorithm 3: Symbolic online test, $OnlineTestImp(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T)$.
Input: $future := 1\text{mtu},\ outLatency := 0,\ inpLatency := 0$
$\mathcal{Z} := \{\langle s_0, e_0 \rangle\}$;                     // let the set contain an initial state
while $\mathcal{Z} \neq \varnothing \wedge \mathsf{GetTime}() \leq T$ do
    while not $buffer.isEmpty$ do                        // consume the output buffer
        $e{:=}\text{buffer.poll}();$ ;                              // dequeue first event
        $\mathcal{Z} := \mathcal{Z}$ after $(\![e.from, e.till]\!) e.channel$; // apply it to the state set
        if $\mathcal{Z} = \varnothing$ then return $fail$;                     // check if it's OK
    $now := \mathsf{GetTime}()$;
    $\mathcal{Z} := \mathcal{Z}$ after $(\![now - outLatency, now + future]\!)$;
    if $\mathcal{Z} = \varnothing$ then return $fail$;                        // is it OK to delay?
    $\mathcal{C} := \mathcal{Z}$ after $(\![now + inpLatency, now + future]\!)$;        // copy for choices
    $c := \mathsf{Random}(\mathsf{Events}(\mathcal{C}, A_{inp}) \cup \{[0, \mathsf{MaxDelay}(\mathcal{C}, now + future)]\tau\})$;
    if $buffer.isEmpty$ then
        $t = \mathsf{Random}((\![c.from, c.till]\!))$;
        sleep until $t$ or wakeup on output at $t' \leq t$;
        if $buffer.isEmpty$ and $e.channel \neq \tau$ then
            $from := \mathsf{GetTime}()$;
            send $c.action$ to $\mathsf{IUT}$;
            $till := \mathsf{GetTime}()$;
            $\mathcal{Z} := \mathcal{Z}$ after $(\![from, till]\!) c.action$;
if $\mathcal{Z} = \varnothing$ then return $fail$ else return $pass$
```

Figure 8 Improved and Implemented Online testing algorithm.

These algorithms and features have been implemented in the real-time testing tool Uppaal-TRON.

5 Future Work

For the work direction on offline testing using timed games, a tool that can convert the generated strategies into executable test cases (possibly in a standardized test case notation language like TTCN3) remains to be developed. We plan to leverage on the work on code generation in controller synthesis to achieve this. A tool for the execution of cooperative strategies is also missing, although Uppaal-TRON is the obvious candidate for this. Test generation will also benefit from general performance improvement of underlying algorithms for solving timed games.

For online testing, the ultimate vision is to allow automated model-based testing for combined real-time, hybrid (mixed discrete and continuous signals) and stochastic models. However, more realistic short-term goals are a test component for probabilistic real-time systems based on the semantic framework of probabilistic Timed Automata as implemented in Uppaal-PROB, and a test component that generates hybrid behavior using co-simulation with Simulink. The existing Uppaal-TRON needs to be challenged by further advanced case-studies.

6 Bibliography

1. M. Utting and B. Legard: **Practical Model-Basted Testing**. Morgan Kaufmann, ISBN: 0123725011, 2007
2. Bouyssounouse, Bruno; Sifakis, Joseph (Eds**.) Embedded Systems Design - The ARTIST Roadmap for Research and Development.** LNCS 3436, ISBN: 978-3-540-25107-1, 2005
3. J. Tretmans. **Model Based Testing with Labelled Transition Systems**., In: Formal Methods and Testing. Edited by R.M. Hierons, J.P. Bowen, M. Harman, 2008. pp. 1-38.
4. Anders Hessel and Kim G. Larsen and Marius Mikučionis and Brian Nielsen and Paul Pettersson and Arne Skou**. Automated Model-Based Conformance Testing of Real-Time Systems**. Chapter in (Eds.) Jonathan Bowen, Mark Harman, and Rob Hierons: Formal Methods and Testing, Springer Verlag, LNCS 4949, 2008
5. Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen. **A Game-Theoretic Approach to Real-Time System Testing** In: Proc. 11th Conference on Design, Automation and Test in Europe (DATE'08).
6. Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen. **Cooperative Testing of Timed Systems.** In: Proc. 4th Workshop on Model-Based Testing (MBT'08), Budapest, Hungary, March 2008, ENTCS 200(1): 79-92.
7. Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen: **Timed Testing under Partial Observability**. In: Proc. 2nd International Conference on Software Testing, Verification, and Validation (ICST'09), Denver, Colorado, April 2009.
8. Marius Mikučionis: **Online testing of real-time systems**. Phd Thesis. To appear 2010.