Project no.:	**ICT-FP7-STREP-214755**
Project full title:	**Quantitative System Properties in Model-Driven Design**
Project Acronym:	**QUASIMODO**
Deliverable no.:	**D4.5**

Title of Deliverable: Final Algorithms and Evaluation

Contractual Date of Delivery to the CEC:	**Month 39**
Actual Date of Delivery to the CEC:	**June 7, 2011**
Organisation name of lead contractor for this deliverable:	
Author(s):	
Brian Nielsen	
Mariëlle Stoelinga,Mark Timmer	
Participants(s):	**AAU, ESI, RWTH)**
Work package contributing to the deliverable:	**WP 4**
Nature:	**R**
Version:	**1**
Total number of pages:	**20**
Start date of project:	1 Jan. 2008 **Duration:** 40 months

Project co-funded by the EC 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 consortium members (including the Commission Services)	

Abstract:

Thus deliverable summarizes our evaluation of our quantitative test generation algorithms and tools. Overall, our application experiences on real case-studies show that the Quasimodo test tools are very promising, and includes unique features not found in commercial tools; especially handling quantities like real-time is based on solid theory.

Keyword list: Test Generation, conformance, evaluation, IOCO, tool integration

Contents

Abbreviations

AAU:	Aalborg University, DK
CFV:	Centre Fèdèrè en Vèrification, B
CNRS:	National Center for Scientific Research, FR
ESI:	Embedded Systems Institute, NL
ESI/RU:	Radboud University Nijmegen, NL
RWTH:	RWTH Aachen University, D
SU:	Saarland University, D

1 Introduction

Testing is the most important practical technique for the validation of software systems. Moreover, even if techniques like model checking will perhaps one day lead to the automated verification of software systems, testing remains an indispensible tool to assess the correctness of the concrete physical operation of software systems on given hardware platforms and in the context of larger, embedding systems. The ultimate reliability of critical software systems that we now depend on for vital applications in everyday life (driving a car, flying a plane, transferring money, operating on patients, etc.) can only be ascertained by testing the final implementations of the hardware and software combinations involved.

Quasimodo has developed several algorithms and tools for firmly based testing of embedded systems with an emphasis of handling quantiative constraints like complex data, time, continuous behavior, probabilities, etc.

The latest version of these algorithms and tool components are already described in the Deliverables D4.3, D4.4, D4.6, and D5.9. Hence, here we focus on some overall evaluations, and future directions.

A distinguishing feature for Quasimodo algorithms and tools is that they are rooted in well defined and well understood testing theories. This is not only of theoretical interst, but of great practical importance as they determine 1) what implementations are considered correct, and thus provides sound verdict assignment, and 2) what observations a test generation algorithm must do and how it should interpret these wrt. the specification.

We evaluate the major implemented tool components based on our experiences and their application to case studies. We also discuss and compare these with external test generation tools.

2 Ioco-based Testing

Participants

- Mark Timmer, University of Twente, NL;
- Ed Brinksma, University of Twente, NL;
- Mariëlle Stoelinga, University of Twente, NL;

Challenge

In spite of the important status of testing as a tool for reliable engineering, the consideration of testing as subject for serious academic study is comparatively late in the development of computer science, i.e., since the 1990s, as before that time most studies concerning correctness were focussed on the development of theories for program and system verification. Nevertheless, nowadays there is a considerable body of knowledge concerning testing theories and tools, most notably as applications of formal methods for concurrent systems and automata theory for dynamic system properties, and the theory of abstract data types for static properties of data structures and operations on them.

The use of formal methods in the context of testing offers the instruments for addressing the following important issues:

- The unambiguous specification of models that capture the allowed behaviours of implementations under test;
- The precise definition of the criteria for conformance, i.e., the formal definition of when the behaviour of an implementation can be considered correct with respect to the specification. Such criteria are often referred to as implementation relations;
- The precise definition of relevant concepts such as test cases, test suites, test runs, the validity of tests, etc;
- A well-defined basis for the development of algorithms for the derivation of valid tests from specifications and the evaluation of test runs, and their implementation in tools for test generation, execution and evaluation.

Results

To contribute to the field, in [11] we gave a comprehensive introduction to a framework for testing based on formal modelling by labelled transition systems and theories of observable behaviour that can be traced back to the process-algebraic approach to concurrency, and process calculi such as CCS [9] and CSP [6]. This work was used as reading material for the well-known Marktoberdorf Summer School.

What we presented is essentially an extension and reformulation of the ioco theory first presented by Jan Tretmans [12, 14], which applies ideas first formulated by Brinksma for synchronously communicating systems [4], to the much more practical setting of input/output systems. The work by Brinksma, in turn, was inspired by the seminal paper of De Nicola and Hennessy that first introduced a formalised notion of testing in process algebra [5].

A central concept in the ioco theory is the notion of quiescence, which characterises system states that will not produce any output response without the provision of a new input stimulus. In the setting of input/output systems one generally assumes the systems to be input-enabled: all input actions are always possible in all system states, i.e., input can never be refused. This means that an input/output system is never formally deadlocked, since one can always execute further (input) actions. In this context quiescence becomes the meaningful representation of unproductive behaviour, comparable to deadlocked behaviour in the case of synchronously communicating systems.

Particular technical elegance of the proposed framework is achieved by representing quiescence in a state by a special output action, representing the absence of 'real' outputs in that state. This allows us to model the relevant implementation relations by the inclusion relation over sets of traces of actions, including quiescence. Such sets of generalised traces then capture the relevant notion of observable behaviour.

An important difference between our presentation and that of Tretmans is that we formulated the whole theory completely in terms of (enriched) traces of labelled transition systems without resorting to process algebraic constructs. Also, there are some subtler differences, viz.:

- Our definition of quiescent transitions has been altered slightly, such that they are preserved under determinisation of the transition systems;

- We do not need the assumption that the transition systems are strongly convergent, i.e. we do allow τ-loops in the implementations under test. In our set-up diverging test runs simply do not affect the set of completed test runs, and therefore also do not affect the test evaluations. If diverging test runs must be excluded to avoid infinite internal computations at the test execution level, one must resort to standard fairness assumptions;
- Our presentation does, in principle, allow for uncountable numbers of states and actions, for which the framework remains intact. This is only useful, however, in the presence of formalisms in which (test) processes over such uncountable sets can be effectively characterised.
- We introduced a novel notion of consistency for test suites, requiring them to fail any implementation that exhibits erroneous behaviour.

Over the years, the ioco framework has established itself as the robust core for a considerable number of theories and tools for conformance testing in different settings, and well-tested, real-life applications. The work in [11] contains the hard core of that successful framework that represents our by now well-established understanding of the desired relation between useful implementation relations for dynamic behaviour on the one hand, and test generation and evaluation on the other hand. Specific results for applying the framework to case studies in Quasimodo is described in Deliverable 5.10.

3 Implementation Relations for Real-Time Model-Based Testing

Participants: Julien Schmaltz (ESI/RU)
Jan Tretmans (ESI)

Testing embedded systems requires not only to test the relation between inputs and outputs, but also to test the timing between these events. A formal theory for model-based testing of untimed systems is the **ioco**-theory [13]. The implementation relation **ioco** uses labelled transition systems with inputs and outputs. A peculiar aspect of the **ioco**-theory is to consider the absence of outputs, called *quiescence*, as a special observable event.

In real-time testing with Timed Automata, in addition to inputs and outputs, there is also time that is observable: a tester can observe the passing of time, or can decide not to provide an input for a certain period of time. This means that time passing, with specific duration, occurs as an observable action in traces and observations. An issue with time, from a testing perspective, is that it is neither input nor output, but a bit of both: a system and its environment synchronize on time, and both can decide to let time pass, or to perform an action first. Time might be called ‘semi-controllable’ and ‘semi-observable’ by the system as well as its environment.

Quiescence is the absence of outputs, now and in the (unbounded) future. Once we have time in our tests, absence of outputs is always for a particular period of time. This is observed by having time pass without any action occurring. This has triggered many discussions whether in timed testing the concept of quiescence is still necessary or desirable.

Several extensions of **ioco** for real-time model-based testing, with and without quiescence, have been proposed in the literature. In Deliverable D4.1: *Quantitative Testing Theory*, which was based on [10], we have presented, discussed, and compared a couple of them.

In an extension of [10] we defined a new real-time implementation relation for timed transition systems: $\mathbf{tioco}_\eta$. This relation, in addition to inputs and outputs, has as an observable event the observation of a fixed delay denoted with η, which may occur repetitively, but quiescence is no longer included.

Further investigations have shown that the $\mathbf{tioco}_\eta$-relation, on the one hand, is very intuitive. Non-conformance according to $\mathbf{tioco}_\eta$ is easily explained using realistic and intuitive timed experiments, and the relation exactly formalizes what is being done in real-time model-based testing tools and case studies – see Deliverable 5:10: *Final report: case studies and tool integration*. On the other hand, the formal definition of $\mathbf{tioco}_\eta$ is straightforward, and it can be shown under which conditions it relates to the other timed implementation relations. In particular, it coincides with e-relativized timed input/output conformance $\mathbf{rtioco}_e$ under the most-general environment e, which is the implementation relation of UPPAAL-TRON; see Figure 1.

We conclude that $\mathbf{tioco}_\eta$ captures the intuitive meaning of real-time testing as well as provides a strong basis for the further development of timed model-based testing. The investigations on $\mathbf{tioco}_\eta$ are currently wrapped up, and prepared for publication.

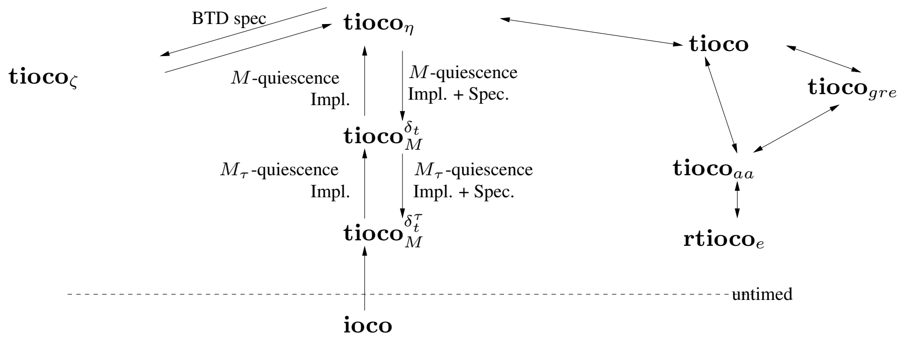

Figure 1: Overview of various timed implementation relations.

4 Timed Test Generation

Participants

- Brian Nielsen, Aalborg University, DK;
- Marius Mikucionis, Aalborg University, DK;
- Kim G. Larsen, Aalborg University, DK;

4.1 UPPAAL-TRON

UPPAAL-TRON is a model-based testing tool for (online) testing of real-time systems. It implements real-time testing based on the concepts of timed automata specifications and timed (environment relativised) input/output conformance, thus having precise semantics and a strong theoretical foundation. The tool implementation supports the full UPPAAL Timed automata language, and uses its successful graphical editor/simulator. The UPPAAL-TRON engine use and extend the symbolic reachability algorithms from the UPPAAL verifier, thus leveraging from its efficient analysis algorithms, to compute symbolic state-set enabling efficient handling of non-determinism and timing uncertainty. Finally, it has a well defined API for programming customized adapters.

An evaluation of the experiences obtained from its development and application to case studies (the two previous Danfoss refrigeration controller case studies, and the Myrianet WSN protocol, Poosl models, Autotrust cases described in Deliverable 5.10) indicate that UPPAAL-TRON is a strong tool for detecting behavioral differences between the model and an implementation. Especially it is found strong on timing compared with other tools.

The case studies also point to areas of improvements:

It may be difficult to apply for people that are not familiar with its concepts, both with respect to setting up, configuring, and running the tool, and visualizing and interpreting the results. In particular the adapter protocols require expert knowledge.

The virtual clock frame work is an excellent and applicable idea, but it must be generalized and easier to use.

Given that we have a sound core, we have already started to address these usability issues. One effort is the development of the UPPAAL-TRON Graphical User Interface described in Deliverable D5.9.

We are also re-thinking the adapter framework, and in particular the virtual clock framework such that they are easier and more flexible to use. In particular the opening of the virtial clock framework towards semi-black box implementations and model simulators. Also the adaptor framework should open up towards allowing the user to choose between using different clocks as time base; the SUT may have its own clock, the test host may have its, or there may be a common independent clock (virtual or physical). Currently UPPAAL-TRON uses the test hosts clock as time base.

4.2 A probabilistic UPPAAL-TRON

The envisioned stochastic extension for (real-time) use-profile testing has not yet materialized. Originally it was envisioned that this extension would be based on the algorithms of the probabilistic version of the UPPAAL model checker; however, the Ph.D. Student working on this task stopped.

The UPPAAL-TRON algorithm is randomized in three ways: In a given moment it may

- Decide to stimulate (give an input) the SUT, or wait for outputs
  - If the decision is to stimulate the system, it must decide what input to give; to achieve this it computes the (input) actions currently enabled in the environment model

  - If the decision is to delay, then it has a choice of how much to wait; to achieve this it computes the (maximum) delay permitted by the environment model.

Currently, these choices are resolved randomly uniformly distributed.

A requirement for use-profile based testing (in particular if used for reliability and performance evaluation) is that the test sequences adhere to statistically sound samples of the model, which must then have a well-defined timed stochastic semantics. Hence more or less ad hoc extensions of the above based on weights on edges and distributions of delays may not satisfy this requirement.

We are therefore pursuing to use the results from statistical model-checking of real-time systems (See Deliverable 2.5) as basis for our future implementation which has recently re-gained momentum from the implementation of statistical model checking in UPPAAL.

5 Review of Model-Based Testing Tools in the Case Studies

Participants: Jan Tretmans (ESI)

Context

In Quasimodo we have worked on different model-based testing (MBT) case studies, in which we used various MBT tools; see Deliverable 5.10: "Final report: case studies and tool integration". In this section we will briefly review the use of the different tools in the case studies.

The MBT tools that were used were the ones under development in Quasimodo: UPPAAL-TRON, JTORX, and TORXAKIS; two additional ones: GAST and Conformiq, and two test tools specifically made for the case study at hand.

Contribution

We first review the main conclusions of the MBT case studies:

1. *Testing the Myrianed WSN gMAC protocol*: The most elaborate case study was testing the WSN gMAC protocol. Several models have been developed, and the three Quasimodo MBT tools: UPPAAL-TRON, JTORX, and TORXAKIS have been used. Model-based testing turned out to be feasible with all three tools, many and long tests were run on the SUT, it increased understanding of the behaviour of the protocol, and some behaviours that did not correspond with the written documentation were detected.

   The behaviour of a WSN node is very non-deterministic and time-critical, e.g., the node can determine autonomously in which slot it will send, and reception of a message can lead to completely different subsequent behaviour based only on the time when it was received.

Models were loosely specified, i.e., they were partial and left details open,, through nondeterminism, because these details were unknown and could not be derived from the documentation.

The SUT consisted of the original 'C'-code compiled and executed on a standard PC in a test environment that redirected inputs, outputs, and clock ticks through a socket connection to the model-based test tool. Due to the use of simulated time the progress of time was much slower than 'real' real-time, thus making long-lasting test runs.

When comparing the three tools there are differences. TORXAKIS cannot directly deal with real-time – clock ticks must be explicitly added to the model and this is cumbersome – but it can deal with data in models: manipulations on all kind of data, also for counting clock ticks, are easily performed. UPPAAL-TRON and JTORX (real-time version [3]) can deal with real-time directly which leads to much more compact and readable models.

Whereas synchronization in simulated time between TORXAKIS and the SUT was easy and straightforward because all clock ticks were represented as explicit actions in the model, this turned out to be more tricky for UPPAAL-TRON. A special adapter, i.e. glue-code, between UPPAAL-TRON and the SUT was developed with a synchronization protocol so that the timing delays of UPPAAL-TRON were correctly mapped to the clock ticks of the SUT, and vice versa.

2. *Model-based testing of the Hydac case*: This case study turned out to be so specific that no general MBT tool was used. First, the connection to he Simulink/Matlab environment (the SUT) required special programming. Second, the testing involved generation of consumption curves, which are sequences of data values satisfying complex sets of constraints. This was implemented directly in Java. The Quasimodo tools cannot deal with such constraints, yet. Only TORXAKIS can deal with data of such complexity, but it cannot deal yet in a systematic way with complex constraints. Moreover, TORXAKIS, being implemented in Haskell, in which also the models have to written, has a steep learning curve, which was considered not worthwhile for this case study. For future research, a solution with a constraint- or SMT solver might be investigated.

3. *Testing of Herschel/Planck*: This project has just started. Models have been made for use with UPPAAL-TRON, and the adapter is under construction, so far without problems.

4. *Model-based testing of electronic passports*: The electronic passport was tested with TORXAKIS, which was successful. A model was written in Haskell, with (restricted) use of data values.

5. *Model-based testing of a software bus at Neopost*: A model was written in mCRL2, validated through simulation, and the implementation wast tested with JTORX. Modelling payed off, and subtle bugs, not found with traditional testing, were detected with JTORX.

6. *Model-based testing POOSL*–UPPAAL: Model-based testing was used to compare two models in different languages. This was successful but in its initial phase.

7. *Testing automated trust anchor updating in Autotrust*: An implementation of an Internet protocol was successfully real-time tested with UPPAAL-TRON with respect to a model constructed from the corresponding RFC.

8. *Testing a printer controller: control part*: The state-based part of a printer controller was tested with TORXAKIS, GAST, a specialized PYTHON tool, and tried with Conformiq. The main features of the SUT are parallelism and resulting nondeterminism. TORXAKIS can in principle deal with nondeterminism but with too much of it it results in a classical state-space explosion due to inefficient internal data structures. GAST can deal with nondeterminism and a couple of previously undetected bugs were detected with it. The disadvantage is that the model (in the functional language Clean) is cryptic and tricky to write. The PYTHON-based tool was written from scratch for this case, and worked well, but could only deal with very restricted parallelism. The commercial tool Conformiq cannot deal with this kind of SUTs because it does not support nondeterminism and only very restricted kinds of parallelism.

9. *Testing a printer controller: data part*: The data part was completely stateless but involved many data variables with constraints. This led to a data-space explosion of possibilities. Equivalence partitioning combined with combinatorial testing (pairwise testing) was used to keep this explosion a little bit under control.

Discussion

We discuss a couple of aspects: real-time, nondeterminism, state-space explosion, data-space explosion, environment modelling, connection to SUT, usability, and integration.

Real-time Real-time behaviour is an important aspect of embedded systems. Real-time testing must be supported in the modelling notation, and it should be built-in in the MBT tool, preferably both 'real' real-time and in simulated time. The most sophisticated tool in this respect is currently UPPAAL-TRON.

Nondeterminism Concurrency, parallelism, abstraction, and partial specifications are important aspects when making models of real-time systems. This means that an MBT tool must support nondeterminism. Surprisingly, commercial tools like Conformiq, and also *Smartesting* [15] do not support that. This implies these tools are not able to deal with testing of systems like WSN. Also specifically developed MBT tools, e.g. programmed in PYTHON, usually have difficulties in dealing with nondeterminism.

Having nondeterminism implies that a well-defined implementation relation must be the basis of test generation, such as $\mathbf{rtioco}_\epsilon$ for UPPAAL-TRON, and **ioco** for JTORX and TORXAKIS.

State-space explosion In on-line testing (see Deliverable D4.2: “Algorithms for off- and on-line quantitative testing”) state-space explosion is much less of an issue then in off-line tools with explicit state representation, yet, the experiments with TorXakis show that it does not completely disappear in case nondeterminism plays an important role.

Data-space explosion Apart from state-space explosion, also the data space may explode, with exponentially many possibilities of combining data parameters. Current Quasimodo MBT tools cannot deal with this in a satisfactory manner. TORXAKIS can deal with data and with all kinds of manipulations and calculations on data, but not yet with huge data-explosion or complex constraint solving. Combinations with tools from these domain, such as combinatorial testing tools and constraint solvers, must be further elaborated.

Environment modelling UPPAAL-TRON makes it possible, or actually requires, that the environment of the SUT is modelled too, so that unrealistic or physically impossible test sequences are excluded. In JTORX and TORXAKIS the more general mechanism of test purposes can be used for the same goal, but environment modelling is not that far incorporated as in UPPAAL-TRON. On the other hand, models in UPPAAL-TRON must be input-enabled which makes that partial specifications cannot be be directly used, but must be made explicit.

Connection to SUT Connecting the MBT tool to the SUT requires development of an adapter. Although some work has been done here, more is needed, in particular in case of real-time behaviour: the synchronization of time between MBT tool and the SUT. Current solutions in UPPAAL-TRON show that this is not trivial, and currently it requires quite some specialized programming if it does not completely fit with the templates with are currently provided in UPPAAL-TRON.

Usability While doing these case studies, JTORX had the nicest user interface of the three, with logging and replay facilities, but in the mean time a GUI has been developed for UPPAAL-TRON and plugged into the UPPAAL GUI to create an integrated environment for modelling, simulation, verification as well as testing; see Deliverable D5.9: "Tool Components and Tool Integration".

The printer controller case studies had as explicit goal that industry engineers would use the MBT tools. However, the usability, documentation, and support of the MBT tools are currently insufficient, so that this turned out to be not feasible.

Integration The Neopost case study shows that it is important to support a whole process of model-based design, validation, simulation, and testing. JTORX with mCRL2 and UPPAAL-TRON with its GUI integration in the UPPAAL tool make steps forward, but more is required, e.g., to linking to existing, industrial tools in the model-based area.

Perspective

Model-based testing has the potential to improve the testing process: a model is a precise and unambiguous basis for testing, design errors are found during validation of mode, longer, cheaper, more flexible, and provably correct tests are automatically generated, test maintenance and regression testing are easier, and the extra effort of modelling is compensated by better tests.

Good MBT tools are indispensable for the success of model-based testing. The current Quasimodo tools are in a good position to become such tools. In various aspects they already outperform currently available commercial tools. But there are also areas for improvement: combining the strengths of the three tools, and avoiding the weaknesses, combining with external techniques and tools, e.g., constraint solving and combinatorial testing, increase usability both in making models and in doing the testing, better linking to other model-based and model-driven tools, and integration within existing industrial testing processes.

6 A Comparison of some Model-Based Testing Tools

Participants: Fides Aarts (ESI/RU)
Jan Tretmans (ESI)

Introduction

Many MBT tools exist, both commercial and academic, tools with different modeling notations, different test generation algorithms, on-line and off-line tools, different ways of connecting to the SUT, etc. Without the intention of being complete, a list of MBT tools would include:

> AETG, Agatha, Agedis, Autolink, Conformiq Qtronic, Uppaal-Cover, GAST, Gotcha, JTORX, MaTeLo, ParTeG, Phact/The Kit, QuickCheck, Reactis, RT-Tester, SaMsTaG, Smartesting Test Designer, Spec Explorer, STG, TestGen, TestComposer, TGV, TORX, TORXAKIS, T-Vec, UPPAAL-TRON, Tveda.

Given the abundance of MBT tools, it is difficult to select the most appropriate one that satisfies the desired needs. In a little investigation, more intended to get insight into what other MBT tools have to offer, than to do a fair and deep comparison, we had a look at two Quasimodo MBT tools: JTORX and UPPAAL-TRON, and three external MBT tools: GAST, MODELJUNIT, and GRAPHWALKER. We tried these tools on an implementation of the simple toy-case study of the game " Crossing the River, with a Wolf, a Goat, and a Cabbage" (see, e.g.,[1]), also called FWGC puzzle (Farmer-Wolf-Goat-Cabbage), which served as SUT. The selection of tools was based on practical criteria, like free availability, and perceived initial ease of use.

We compared on the following criteria:

- does the tool support input generation and output checking;
- ease of connection between the SUT and the MBT tool (the adapter);
- code coverage of Java code of the SUT implementation; we used Cobertura[2], an open-source coverage tool for measuring and visualizing coverage of Java programs.
- supported implementation relation(s), as far as specified;

[1]http://www.mathcats.com/explore/river/crossing.html
[2]http://cobertura.sourceforge.net/

- supported platforms;
- ease of use and documentation.

Description of the MBT Tools

JTorX JTorX is a Java-based tool developed to test whether an SUT is **ioco**-conforming to a given specification [1]. It is based on the core functionality of TorX [2], but it contains additional features, e.g. besides **ioco**, testing for **uioco** (extension of **ioco** that can cope with underspecified traces) is possible. Another extension is the simulation feature that is included in JTorX. The simulator allows guided explorations, e.g., by using the log of a test run. In JTorX specifications are defined as Labelled Transitions Systems (LTS), which can be represented in Aldebaran (.aut) or GraphML (.graphml) format. Implementations can be given in the same format (then they are simulated) or as a real program.

The communication to the SUT is accomplished via standard in- and output, a TCP connection, or with the TorX adapter protocol. For the FGCW puzzle we used standard input and standard output to connect the SUT to JTorX. In general, the connection is straightforward and easy to achieve.

The test cases generated from the model covered 96% of the lines of code and 95% of the branches in the FWGC Java implementation.

As mentioned before, JTorX supports the implementation relations **ioco** and **uioco**. JTorX generates inputs and checks outputs, and it does on-line MBT.

JTorX is available for Linux, Mac and Windows, but others can be added. For our experiments we installed it on Windows and Linux, which was easy. The user interface of JTorX is user-friendly and simple. The tabs are clearly arranged and it is straightforward to get the tests running. For this purpose only a few instructions are required. However, the documentation included is only moderate. For the SUT used, i.e. the implementation of the FWGC puzzle, JTorX was a very suitable MBT tool. All tests could be carried out as desired and the results were as expected. When a SUT-mutant of the imp was added, the tool was able to find it.

Uppaal-Tron Uppaal-Tron is an MBT tool based on the Uppaal engine [7]. It can be used for black-box conformance testing of timed systems and it mainly targets embedded software. It tests systems in an on-line fashion: tests are derived, executed and checked while maintaining the connection to the system in real-time.

Uppaal-Tron can be connected to the SUT in several ways. The communication can either be done using a TraceAdapter or a SocketAdapter. The TraceAdapter uses standard input and standard output for communication. In our experiments we used the SocketAdapter to achieve communication with Uppaal-Tron. The SocketAdapter communicates using TCP sockets. The Uppaal-Tron package comes with pre-made Java classes that implement the protocol it uses. In order to use the SocketAdapter, these classes and some additional setup code have to be put in place around the SUT. In order to test a program with Uppaal Uppaal-Tron, first a model and an environment need to be created in Uppaal. For our experiments we created a detailed model of the FWGC puzzle in Uppaal. The environment used does not contain any logic; it simply consists of one state that has a transition leading to itself for every available channel. When such a model and environment have been created, it is possible to use UPPAAL-TRON, to load the model, and generate test cases for the SUT. In order to communicate, it uses sockets which require implementation of a SocketAdapter for the SUT. After the necessary adapter construction, we compared the SUT and the model using UPPAAL-TRON. After a few iterations of the model, all tests runs succeeded and a pass verdict was returned.

The test cases generated from the model covered 92% of the lines of code and 88% of the branches in the FWGC implementation.

UPPAAL-TRON is available for Linux and Windows. It supports the implementation relation $\mathbf{rtioco}_e$ (relativized timed input-output conformance) [8]. This relation extends $\mathbf{ioco}$ with explicit environment assumptions and takes timing into account, which is not applicable to our example FWGC that does not have any real-time characteristics. UPPAAL-TRON generates input data and checks output data.

Based on its specifications, Uppaal UPPAAL-TRON seems to be a good tool for real-time embedded systems. The tool also has comprehensive documentation that allows starting quickly. However, compared to JTORX, UPPAAL-TRON is slightly more difficult to use.

GAST GAST is a generic automatic software test-system that is implemented in the functional programming language Clean Kooetal03. Its primary goal is to test software written in Clean. However, GAST is not restricted to software written in Clean. In GAST, there are two kinds of functions that need to be defined: properties and the SUT. Properties, which are expressed as functions in Clean, are used in the tests to check whether they hold for the SUT. The SUT is a function which accepts a certain amount of arguments defined with specific types. The type information is used by GAST to generate test-data automatically. Also the other steps in the functional testing process, i.e. test execution and test result analysis, can be performed fully automatically by GAST.

In GAST, the implementation and specification are defined by Clean-functions. The specification function takes a state and an input and results in a list of possible states combined with output. The implementation function takes an implementation state as an input and results in a collection of possible output implementation states. We accomplished communication with the Java implementation through the implementation function using stdin and stdout and linking those to Java. Because inputs and outputs had to be written and read alternately, a quiescence output from Java to GAST has been added, so that every input resulted in one output. Alternatively, we connected Java and GAST using pipes, which also worked fine after solving a buffering problem of Clean. To allow GAST to perform multiple test runs, we extended the Java program with a reset input.

The test cases generated from the model covered 98% of the lines of code and 95% of the branches in the FWGC implementation.

To use GAST, you need a Clean 2.0 system with generics, which is available for Linux and Windows. The implementation relation supported by GAST is a version of $\mathbf{ioco}$ for Mealy Machines. GAST generates input data and check output data; it works in an on-line fashion.

There is no graphical modelling tool, therefore models, inputs, and outputs must be defined in the Clean programming language. GAST seems to be a very powerful tool with Clean as a very expressive language, but Clean is not easy to learn, which makes using GAST a challenging task. Unfortunately, documentation is hardly available, which could have solved this problem to some extent. Moreover, it has no standard options to communicate with external SUT. Thus, compared to the other MBT tools, adapter construction and connecting the SUT is more complicated.

MODELJUNIT MODELJUNIT is a Java library that extends JUnit to support model-based testing[3]. The models in MODELJUNIT are finite state machines (FSM) or extended finite state machines (EFSM) written in Java by using the given framework. Models are classes with methods that implement the states and, if necessary, the guards to allow the EFSM to enter that state. The internal state of the model is saved in the state variable in form of an integer enumerating the states. The implemented models are fed to a Tester instance (random, greedy and look-ahead testers are available already). After connecting listeners to the Tester instance, a couple of test cases can be run.

To allow automatic test runs, the model has to communicate with the SUT. The class that implements the SUT is passed to the SUT in MODELJUNIT in form of a command line string which is executed (i.e., a process instance is created) on instantiation or on reset of the model. If the model wants to emit an input string to the SUT, the `message()` method is called with this string and it is sent to the standard input of the SUT. The same method can be used to read a string from the SUT. The last character of the parameter of the `message()` method is used to determine whether the message is and input (?) or output (!).

Coverage In order to obtain coverage of the model, certain coverage metric instances can be added to the Tester instance as well (as of this writing the API documentation includes action, state, transition and transition-pair coverage). After implementing the model, we instantiated the Tester and ran it. Unfortunately, we did not see a way to obtain understandable output from the test runs by using the verbose listener. No other listener worked, and thus running a couple of test cases is possible but not useful. MODELJUNIT offers classes to obtain coverage information on test runs. This gave some results, but this value depended strongly on how many test cases were run and according to this the results are not reliable.

MODELJUNIT is OS independent. In the documentation we could not find any information on supported implementation relations. Actually, the documentation of MODELJUNIT is non-existent in terms of usable documentation: the API is documented by JavaDoc and there is one example on how to implement a model available on the project homepage. Yet there is no handbook available apart from [15], which supposedly has at least one chapter about MODELJUNIT. Maybe due to the lack of documentation, it was difficult to use the tool. For example, we were not able to obtain a clear error message in case the SUT behaved differently with respect to the model. Moreover, we had problems with modeling the specification, which could have been circumvented with a clear description of how to use the guards and how to update the current state.

GRAPHWALKER According to their website GRAPHWALKER is an open source tool for model-based testing. However, their interpretation of MBT is different from how we understand the term. GRAPHWALKER is a tool for generating off-line and on-line test

[3]http://www.cs.waikato.ac.nz/ marku/mbt/modeljunit/

sequences from finite state machines and extended finite state machines. When using the tool, it is possible to create long, unpredictable test sequences with high variation and randomness, which will result in better test coverage of the system under test. In our experiments, we reached 96% statement coverage and 94% branch coverage of the FWGC implementation. As a result, GRAPHWALKER is a powerful tool for test data generation and test execution. However, it does not support test result analysis. We did not find any means to produce a verdict or to check whether the output generated by the SUT is correct. Because GRAPHWALKER has less functionality than the other tools concerning the entire MBT process, we do not include it in our overall comparison.

Summary

The table below summarizes the main results of our investigation. Based on our experiences with the tools, we assigned points at a scale from one (bad) to five (good) for connection to SUT, coverage, supported implementation relation(s), platforms, ease of use and documentation. GRAPHWALKER is not included in the table due to its different focus.

	JTORX	UPPAAL-TRON	GAST	MODELJUNIT
connect. to SUT	5	4	2	4
coverage (st./br.)	96%/95%	92%/88%	98%/95%	N/A
impl. rel.	ioco,uioco	ioco,rtioco	ioco (FSM)	N/A
platforms	Lin.,Mac,Win.	Lin.,Win.	Lin.,Wind.	indep.
ease of use	4	3	2	2
documentation	3	4	2	2

Altogether, we can conclude that JTORX seems to be the most capable tool for the system under test and model we used. JTORX gives good results, it is very user-friendly, and easy to use. However, our comparison is very limited, we did not use all functionality of the different testing tools, and we were perhaps not completely without bias,, being already familiar with JTORX. In case of real-time embedded systems, UPPAAL-TRON will probably outperform JTORX (there a real-time variant of JTORX [3] which we did not consider here).

Of course, a more elaborate comparison is necessary, involving more tools, using more criteria, and applying the tools to other SUTs, in order draw hard conclusions, but this first investigation gives already first insights in other model-based testing tools. Another comparison is e..g, [16].

Discussion

A couple of points need mentioning when looking at the results of this short comparison:

- Some tools only support the generation of test-input data; checking the correctness of outcomes is not supported, and consequently must be arranged in a different way. All Quasimodo tools support both test-input data generation and checking of outcomes.

- For connection between the SUT and the MBT tool (the adapter) JTORX seems to be the winner: the connection can be made without programming in the MBT tool. All other tools need some kind of programming in the adapter on in the tool itself.
- Code coverage differences are not significant. Moreover, for real comparison, more information is needed, such as length of test cases, duration of test campaign, etc.
- MODELJUNIT and GRAPHWALKER do not say anything about an implementation relation, which makes it impossible to predict what kind of errors they will detect. It looks like these tools only support deterministic models, (although they are not explicit about that). For fully deterministic models most most implementation relations coincide anyhow, but as the Quasimodo case studies show, testing embedded systems requires dealing with nondeterminism.
- All tools have at least an on-line testing option. The tools with explicit implementation relation are only on-line.
- Most tools run at least on Linux and Windows.
- For most tools, except for UPPAAL-TRON, documentation seems to be an afterthought. Yet, JTORX is considered more user friendly.
- These tools are very interesting research and experimental vehicles, but not yet full industrial strength tools.

7 Towards Integration of Hybrid, complex data, and other aspects

Quasimodo has developed strong tool components for testing control, real-time, complex data, and stochastic models.

However, they are not fully integrated in the sense that one tool component supports all aspects. As discussed, handling time, data intensive, hybrid, costs and stochastics integrated into the same omni-potent notation and algorithmic core is (at best) challenging, it is a question whether a more feasible approach (especially in the short term) would be to integrate specialized tool components.

We developed an approach for testing hybrid systems based on an integration of UPPAAL-TRON and Simulink as described in Deliverable 4.6. Moreover, as specifically reported in D5.9, a unifying framework for symbolic model-based testing (aimed at handling complex data structures and data dependencies) is being carved out and prototyped implemented for the JTorX/STSSimulator tools. The challenge of hiding the underlying diversity is handled through Domain Specific Languages for modeling.

A future vision may therefore be to integrate the specialized test tool components to exploit their individual strengths via a a common "test-tool-bridge" framework, as illustrated in Figure 2 However, developing such a framework that enables tool components to synchronize and exchange information in a well-defined semantically sound manner will require further research beyond Quasimodo.

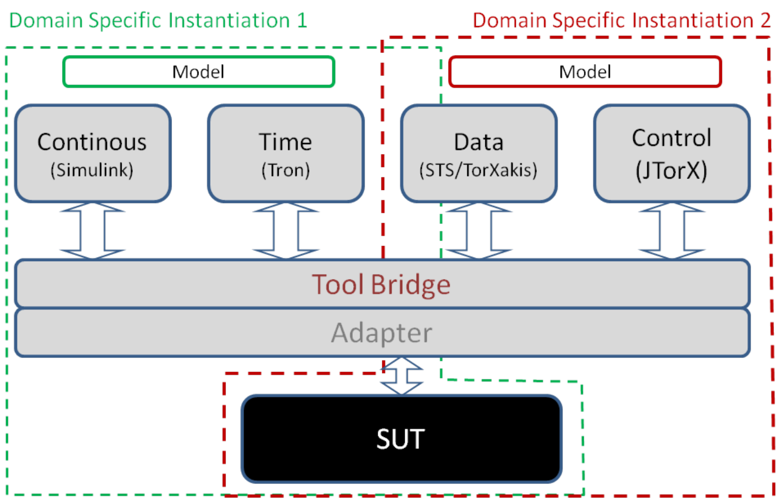

Figure 2: A possible future tool integration framework.

References

[1] A.F.E. Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer, 2010.

[2] A.F.E. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L.M.G. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems (IWTCS '99)*, volume 147 of *IFIP Conference Proceedings*, pages 179–196. Kluwer, 1999.

[3] H. Bohnenkamp and A. Belinfante. Timed Testing with TorX. In J. Fitzgerald, I.J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods: Int. Symposium of Formal Methods Europe*, number 3582 in Lecture Notes in Computer Science, pages 173–188. Springer-Verlag, 2005.

[4] H. Brinksma. A theory for the derivation of tests. In *Proceedings of the 8th International Workshop on Protocol Specification, Testing, and Verification (PSTV '88)*, pages 63–74, 1988.

[5] R. de Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1–2):83–133, 1984.

[6] C.A.R. Hoare. *Communicating sequential processes*. Prentice Hall, 1985.

[7] K.G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing Real-Time Embedded Software using UPPAAL-TRON: An Industrial Case Study. In W. Wolf, editor, *EMSOFT 2005 – ACM Int. Conf. On Embedded Software*, pages 299–306. ACM, 2005.

[8] M Mikucionis. *Online Testing of Real-Time Systems*. PhD thesis, Dept. of Computer Science, Aalborg University, Aalborg, Denmark, 2010.

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

[10] J. Schmaltz and J. Tretmans. On Conformance Testing for Timed Systems. In F. Cassez and C. Jard, editors, *Formal Modeling and Analysis of Timed Systems – FORMATS 2008*, volume 5215 of *Lecture Notes in Computer Science*, pages 250–264. Springer-Verlag, 2008.

[11] M. Timmer, H. Brinksma, and M. I. A. Stoelinga. Model-based testing. In *Software and Systems Safety: Specification and Verification*, NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2011, to appear.

[12] G. J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[13] J. Tretmans. Model Based Testing with Labelled Transition Systems. In R.M. Hierons, J.P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer-Verlag, 2008.

[14] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[15] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.

[16] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-Based Testing. Working Paper Series 04/2006, The University of Waikato, Department of Computer Science, Hamilton, New Zealand, 2006.