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

Deliverable no.: D5.9

Title of Deliverable: Tool Components and Tool Integration

Contractual Date of Delivery to the CEC:	**M30**
Actual Date of Delivery to the CEC:	**June 3, 2011**
Organisation name of lead contractor for this deliverable:	**AAU**
Author(s):	**Kim G. Larsen, Alexandre David** **Holger Hermanns, Joost-Peter Katoen** **Brian Nielsen, Axel Belinfante**
Participants(s):	**AAU CFV CNRS ESI RWTH SU**
Work package contributing to the deliverable:	**WP 1–4**
Nature:	**R**
Version:	**1.1**
Total number of pages:	**35**
Start date of project:	1 Jan. 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 describes the development of the quantitative validations tools by individual partners within the two categories: tools for validation of probabilistic and stochastic systems, and tools for validation of real-time systems. Also, we provide a short status on the progress on interaction between tools (including external and commercial tools).

Keyword list: CoDeMoC, UPPAAL SMC, SCOOP, OPAAL, JTorX, UPPAAL Tron, Tool Components, Tool Integration, Simulink, UML State-Charts, Framework for Symbolic Testing

Contents

1 Introduction 4

2 Development of Tools by Individual Partners 5

2.1 CoDeMoC – (Co)ntinuous Time Markov Chain against (De)terministic Timed Automata (Mo)del (C)hecker 5
2.2 UPPAAL SMC 7
2.3 SCOOP: A Tool for SymboliC Optimization Of Probabilistic Processes 10
2.4 OPAAL: A Lattice Model Checker 13
2.5 JToRX 15
2.6 UPPAAL Tron 17

3 Tool Components 20

4 Integration between Tools 25

4.1 Linking Probabilistic Tools 25
4.2 Linking to Matlab/Simulink 26
4.3 Linking to UML State-Charts 26
4.4 Towards a Unifying Framework for Symbolic Model-Based Testing 28

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

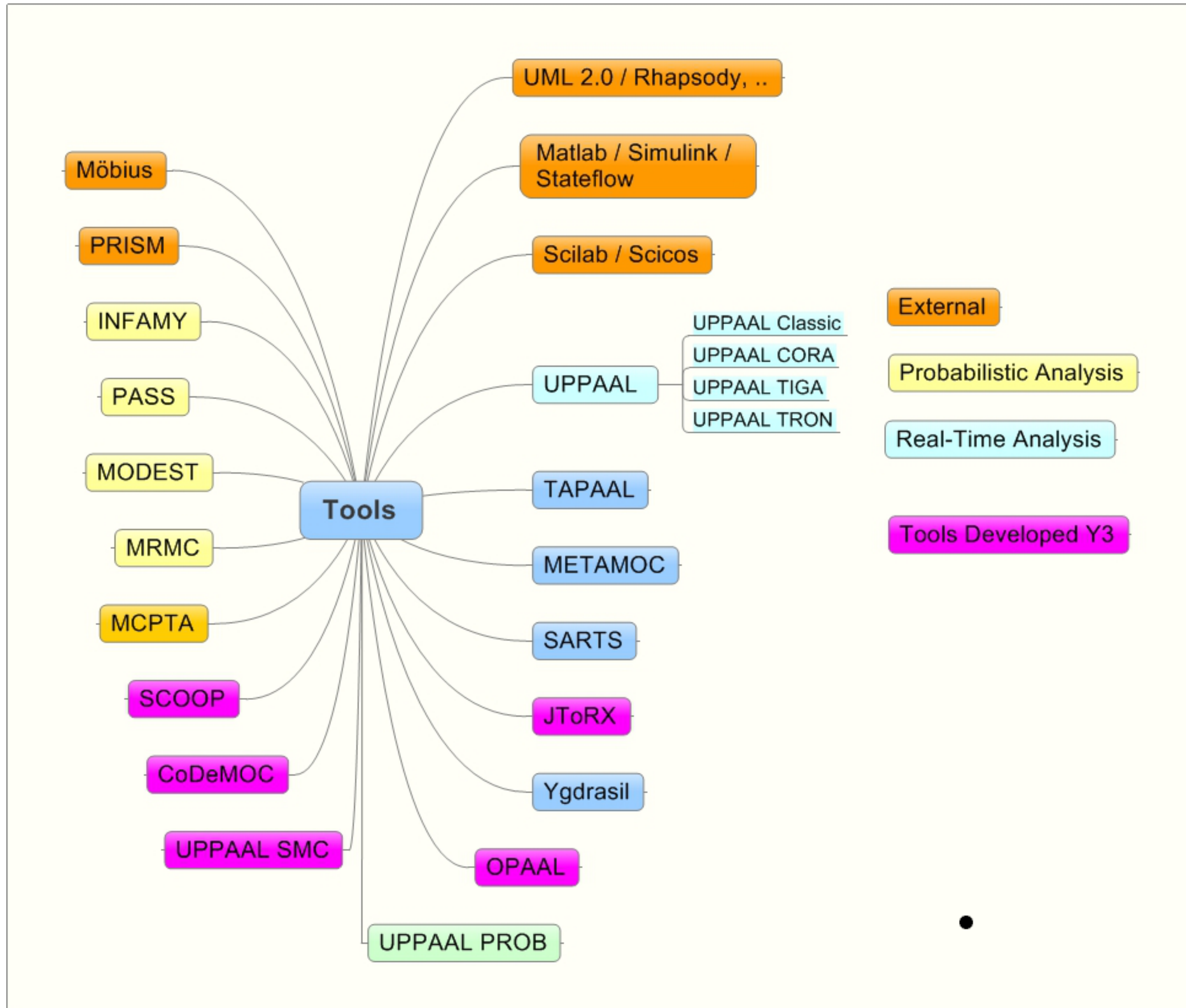

Figure 1: Internal and External Tools in Quasimodo (purple indicate tools developed or extended during year 3).

In this deliverable we describe the third year effort in Quasimodo on development of tools by individual partners (see Fig. 1). We also give the final status on the development and realization of frameworks for exchange and interaction between internal tools and external (commercial) tools. The tools that have been developed or extended during year three will be detailed in the following section. For detailed description of the other Quaosimodo tools mentioned in Figure 1 we refer to last years deliverable D5.8.

2 Development of Tools by Individual Partners

2.1 CoDeMoC – (Co)ntinuous Time Markov Chain against (De)terministic Timed Automata (Mo)del (C)hecker

Participants

- Benoit Barbot, Joost-Pieter Katoen, Tingting Han, Alexandru Mereacre; RWTH Aachen; Germany
- Taolue Chen; Deparment of Computer Science, Oxford University

Contribution CoDeMoC is a model checker for Continuous-time Markov chains (CTMCs) against linear real-time specifications described as Deterministic timed automata (DTA). This tool computes the probability of the set of timed paths accepted by the DTA A in the Markov chain C. It supports finite acceptance conditions. The central question that it addresses is: what is the probability of the set of paths of C that are accepted by A, i.e., the likelihood that C satisfies A? Under finite acceptance criteria this equals the reachability probability in a finite piecewise deterministic Markov process (PDP). Qualitative verification amounts to a graph analysis of the PDP, i.e., in order to decide whether the probability that C satisfies A exceeds zero, or equals one, a graph analysis suffices. Reachability probabilities in our PDPs are then characterized as the least solution of a system of Volterra integral equations of the second type and are shown to be approximated by the solution of a system of partial differential equations.

For single-clock DTA, this integral equation system can be transformed into a system of linear equations where the coefficients are solutions of ordinary differential equations. As the coefficients are in fact transient probabilities in CTMCs, this result implies that standard algorithms for CTMC analysis suffice to verify single-clock DTA specifications.

The tool supports bisimulation minimisation and parallelisation. The latter exploits the transient analysis of the several sub-CTMCs in the product of the CTMC and the single-clock DTA.

Three case studies from different application fields have been used to show the feasibility of the tool component. The first case study has been taken from Donatelli et al. which considers 1-clock DTA as time constraints of until modalities. Although using a quite different approach, our verification results coincide. The running time of our implementation is about three orders of magnitude faster. Other considered case studies are a randomly moving robot, and a real case study from systems biology.

Bisimulation quotienting (i.e., lumping) yields state space reduction of up to one order of magnitude, whereas parallelizing transient analysis yields speedups of up to a factor 13 on 20 cores, depending on the number of subgraphs in the decomposition.

The discretization approach for multi-clock DTA may give rise to large models: checking a robot example (up to 5,000 states) against a two-clock DTA yields a 40- million state DTMC for which simple reachability probabilities are to be determined. The tool component can be downloaded from
`http://www-i2.informatik.rwth-aachen.de/CoDeMoC`.

Papers

1. Taolue Chen, Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre. Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications. Logical Methods in Computer Science, 7(1-2):134, 2011.

2. Benot Barbot, Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Efficient CTMC Model Checking of Linear Real-Time Objectives. TACAS 2011: 128-142

3. Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications. LICS 2009: 309-318

2.2 UPPAAL SMC

Participants:

- Alexandre David, Kim G. Larsen, Marius Mikucionis, Danny Poulsen, Jonas V. Vliet, Aalborg University, DK
- Axel Legay; IRISA/Rennes, F
- Wang Zhen, East China Normal University, Shanghai, China

Contribution UPPAAL SMC extends the UPPAAL toolset with the ability to perform statistical model checking of networks of timed automata. The tool monitors several runs of the system, and then applies estimation and hypothesis testing to estimate the correctness of the system with respect to probabilistic guarantees of time- and cost-bounded properties. The engine for generating runs is based on a natural semanteics of (networks of priced) timed automata based on races between components.

One of the major differences with classical Uppaal is the introduction of a new user interface that allows to specify networks of constant slope timed automata (CSTA) with respect to a stochastic semantic such semantic is naturally needed to apply SMC. Another contribution is the implementation of several versions of the sequential hypothesis testing algorithm of Wald. Contrary to other implementations of SMC, we also consider those tests that can compare two probabilities without computing them. The tool has been applied to a number of case studies including the IEEE 1394 High Performance Serial Buss ("Firewire"), Firewire Protocol, Lightweight Media Access Control Protocol as well as stochastic job-shop scheduling modelled using Duration Probabilistic Automata. On the later class of case-studies, the performance of UPPAAL SMC is order or magnitude faster than the estimation and statistical model checking engines of PRISM.

The UPPAAL SMC supports the rich modeling formalism of UPPAAL with the addition of allowing clocks to have different rates (possibly even negative) in different locations. Rational expression attached to locations are used to define the exponential distribtions of delays. We add branching edges and associated weights for the probabilistic extension as shown in Fig. 2. We also generalize rates on clocks to be expressions that take value over integers (even negative) compared to just 0 and 1 for UPPAAL.

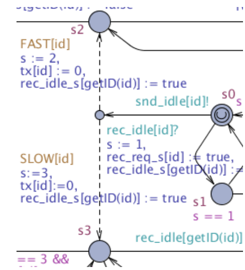

Figure 2: Branching edges (from firewire case-study).

Properties are evaluated on bounded runs by `time`, `steps` or by a clock, or by a cost constraint. The bound is a constant value. The expressions `expr` are state predicates. The properties are of the following type:

- A qualititive check: `Pr[time<=bound](<> expr) >= p`.
- A quantitative check: `Pr[time<=bound](<> expr) ?`.
- A comparison check `Pr[time1<=bound1](<> expr1) >= Pr[time2<=bound2](<> expr2)`.

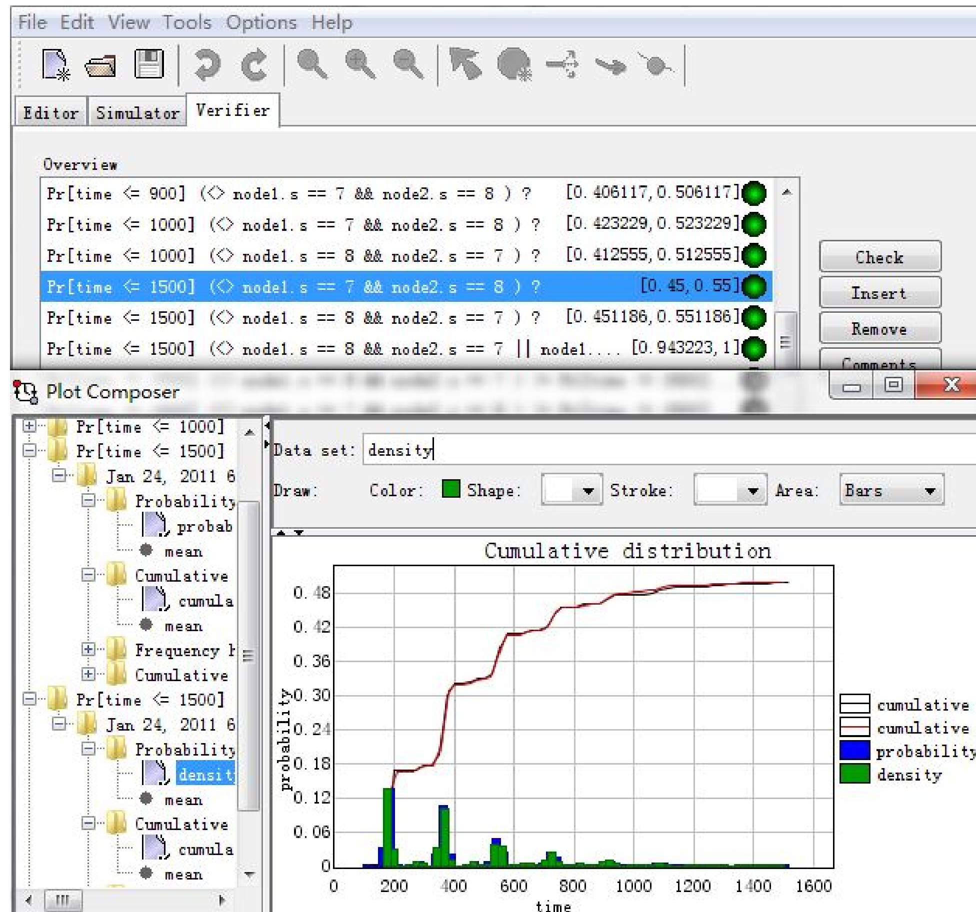

Figure 3: The verifier shows the results of the different probabilistic queries and the plot composer shows probability distributions.

The first formula is to check if the probability of satisfying some property is at least p. The second one estimates this probability within an interval. The last one compares two probabilities without evaluating them and gives the result for all bounds up to bound1 if both bounds are the same. All these checks rely on some statistical algorithm to estimate the correctness by observing runs of the system. The qualitative check applies a variant of the sequential hypothesis testing of Wald, the quantitative checks estimates the probability with a Monte-Carlo based approach, finally the comparison check is an extension of sequential hypothesis testing that allows to compare probabilities without computing them. The algorithms are precise up to a certain value that can be chosen by the user.

The verifier shows the estimated intervals of probabilities and provides a plot composer to visualize and compare different results. Figure 3 shows a screenshot with the verifier (above) and the plot composer (below). The verifier provides additional results in a form of plotted data which are accessible via a popup-menu by right-clicking over the property. Any plot can be exported to a number of graphical formats and data saved in a textual format.

In addition, a custom plot can be created in plot composer accessible via Tools menu. On the

left side of the plot composer, the data sets are grouped and displayed in a tree structure. Any data set can be selected for the composite plot by double clicking on its node and the same way de-selected. The details of the plot can be customized on the right side above the plot.

Papers

1. Alexandre David, Axel Legay, Zheng Wang, Kim G. Larsen and Marius Mikucionis, Time for Statistical Model Checking of Real-time Systems, in: Proceedings of the 23rd International Conference on Computer Aided Verification (CAV), pages xxxx, Springer Verlag, 2011.

2. Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, Danny B. Poulsen, Jonas V. Vliet and Zheng Wang, Statistical Model Checking for Networks of Priced Timed Automata, 2011.

2.3 SCOOP: A Tool for SymboliC Optimization Of Probabilistic Processes

Participants

- Jaco van de Pol, Marielle Stoelinga, Mark Timmer; Twente, The Netherlands
- Joost-Pieter Katoen, RWTH, Germany

Challenge Several algorithms and tools exist for model checking qualitative and quantitative properties for a wide range of probabilistic models, modelling for instance randomised protocols or biological processes. Although these techniques are promising, their applicability is limited by the well-known *state-space explosion* and the restricted treatment of *data*.

Probabilistic process algebras typically only allow a random choice over a fixed distribution, and input languages for probabilistic model checkers such as the PRISM language [22] or the probabilistic variant of Promela [3] only support basic data types. However, to model realistic systems, more elaborate and convenient means for data modelling are indispensable.

The incorporation of data yields a significant increase of state space size. However, most of today's probabilistic minimisation techniques are not well-suited to be applied in the presence of data. Although several reduction techniques can be applied at the model level, this requires the complete model to be constructed first.

Results Our approach is to *symbolically* reduce at the process-algebraic level, minimising state spaces prior to their generation by means of syntactic transformations.

We developed the probabilistic process-algebraic language prCRL [29, 30], generalising the μCRL language [20] by adding a probabilistic choice operator. We also defined a restricted variant of prCRL, called the LPPE (linear probabilistic process equation). Any prCRL specification fulfilling some mild conditions can be transformed (linearised) into an LPPE, which then allows symbolic reductions and easy state space generation (in the form of a probabilistic automaton). On the LPPE format, reductions can be applied. Most of these transform an LPPE into an equivalent LPPE. Figure 4 illustrates the approach.

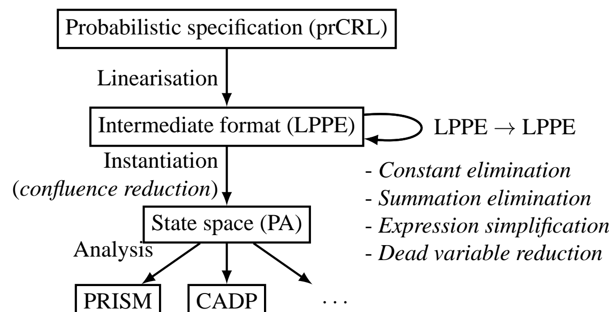

Figure 4: The LPPE-based verification approach

The SCOOP tool [36] we implemented takes a prCRL specification, linearises it, applies reduction techniques, and generates the (often significantly) reduced state space, without having

to construct the complete state space first. The tool is publicly available, open source and has a user-friendly web interface[1].

SCOOP implements three LPPE simplification techniques, which do not change the actual state space, but improve readability and speed up state space generation. (1) *Constant elimination* detects if a parameter of an LPPE never changes its value. Then, the parameter is omitted and every reference to it is replaced by its initial value. (2) *Summation elimination* simplifies nondeterministic choices for which only one of the possible alternatives enables real behaviour. This kind of construction often occurs when composing parallel components that communicate using message passing. (3) *Expression simplification* rewrites conditions, action parameters and next state parameters using for instance basic logical identities and the evaluation of functions, where possible using heuristics.

SCOOP implements two state space reduction techniques, that do change the LPPE or instantiation in such a way that the resulting state space will be smaller. (1) *Dead variable reduction* was generalised easily from the non-probabilistic variant, introduced in [40]. It reduces state spaces while preserving strong bisimulation, by resetting irrelevant variables based on the control flow of an L(P)PE. (2) *Confluence reduction* was introduced in [7] for LPEs, to reduce state spaces while preserving branching bisimulation. Basically, it detects internal τ-transitions that do not influence a system's behaviour, and uses this information when generating the state space. A generalisation to the probabilistic setting was presented in [37].

All these techniques work on the syntactic level, i.e., they do not unfold the data types at all, or only locally to avoid a data explosion. Hence, a smaller state space is obtained without first having to generate the original one.

SCOOP was developed in Haskell (6640 lines of code excluding comments), based on a simple data language to allow the modelling of several kinds of protocols and systems. A web-based interface makes it convenient to use; it provides the user with 30 seconds of server-time per request. Alternatively, SCOOP can be downloaded to run locally on any platform.

The tool automatically applies the LPPE simplification techniques, and allows the user to choose whether or not to apply dead variable reduction and/or confluence reduction.

After generating and optimising an LPPE, SCOOP can also generate its state space and display it in several ways. It can export to the AUT format for analysis with the CADP toolset [19], or to a transition matrix for analysis using PRISM [22]. Alternatively, under some conditions SCOOP can translate the optimised LPPE directly to the PRISM language, allowing the use of this probabilistic model checker to symbolically compute (quantitative) properties of the model.

To illustrate the strengths of SCOOP, we applied it to some case studies. Both state space reduction techniques were applied, decreasing the number of states by 90% – 95%. The number of transitions decreased even more. The time needed to generate the reduced state spaces was always at most one fourth of the time to generate the original ones (up until the point where swapping was needed; then, the generation of the smaller state spaces is even faster, relatively).

[1] The implementation, including the web-based interface, can be found at `http://fmt.cs.utwente.nl/~timmer/scoop/`.

Papers:

1. Joost-Pieter Katoen, Jaco van de Pol, Marielle Stoelinga and Mark Timmer, A Linear Process Algebraic Format for Probabilistic Systems with Data, in: Applications of Concurrency to System Design (ACSD), IEEE CS Press, 2010

2. Joost-Pieter Katoen, J. van de Pol, Marielle Stoelinga and Mark Timmer, A linear process-algebraic format with data for probabilistic automata (2011), in: Theoretical Computer Science

2.4 OPAAL: A Lattice Model Checker

Participants

- Mads Chr. Olesen, Kenneth Y. Jorgensen, Kim G. Larsen; Aalborg University, DK

Contribution Common to almost all applications of model checking is the notion of an underlying concrete system with a very large or sometimes even infnite concrete state space. In order to enable model checking of such systems, it is necessary to construct an abstract model of the concrete system, where some system features are only modelled approximately and system features that are irrelevant for given verifcation purposes are abstracted away.

The model checker OPAAL allows for such abstractions to be specifid through user-defined lattices that are part of the model. We call them lattice automata. Lattice automata are synchronising extended finite state machines which may include lattices as variable types. The lattice elements are ordered by the amount of behaviour they induce on the system, that is, larger lattice elements introduce more behaviour. We call this the monotonicity property.

Lattice automata, as implemented in OPAAL, are a subclass of well-structured transition systems. The tool can exploit the ordering relation to reduce the explored state space by not re-exploring a state if its behaviour is covered by an already explored state. In addition to the ordering relation, lattices can have a join operator (the least upper bound), joining two lattice elements into one, thereby potentially overapproximating the behaviour, with the gain of a reduced state space. The overapproximated model checking can however be inconclusive. We introduce the notion of a joining strategy, by specifying which lattice elements are joinable, allowing the user to specify the amount of overapproximation with much more control. This allows for a form of user-controlled CEGAR (Counter-Example Guided Abstraction Refinement). The CEGAR approach can easily be automated by the user, by exploiting application-specific knowledge to automatically derive more fine-grained joining strategies given a spurious error trace. Using joining strategies can thus, for some systems and properties, provide very efficient model checking and conclusive answers at the same time.

The OPAAL tool is released under an open source licence, and can be freely downloaded at `www.opaal-modelchecker.com`. The OPAAL tool is implemented in Python and is a stand-alone model checking engine. Models are specfied using the UPPAAL XML format, extended with some specialised lattice features. Using an interpreted language has the advantage that it is easy to develop and integrate new lattice implementations in the core model checking algorithm. Our experiments indicate that although OPAAL uses an interpreted language, it is still sufficiently fast to be useful. Users can create a new lattice by implementing a simple Python class interface. The new class can then be used directly in the model (including all user-defined methods). Joining strategies are defined as a Python function.

An overview of the OPAAL architecture is given in Fig. **??**, showing the five main components of OPAAL. The Successor Generator" is responsible for generating a transition function for the transition system based on the semantics of UPPAAL automata. The transition function is combined with one or more lattice implementations from the Lattice Library", which can be easily extended with further user-defined lattices.

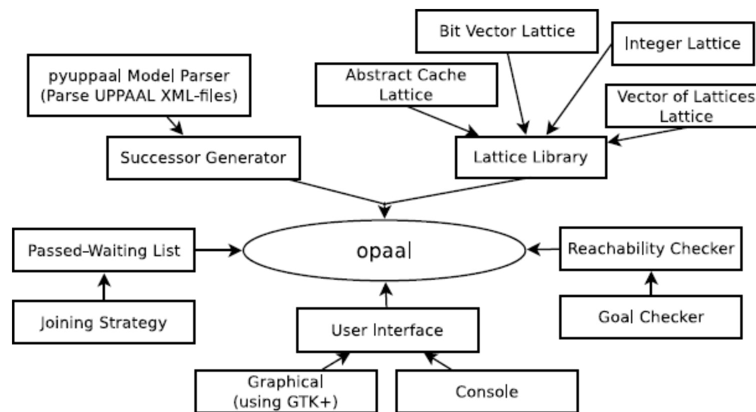

Figure 5: Overview of OPAAL's architecture.

The Successor Generator" exposes an interface that the Reachability Checker" can use to perform the actual verification. During this process a Passed-Waiting List" is used to save explored and to-be explored states. The Passed-Waiting List" uses a user-provided Joining Strategy" on the lattice elements of states, before they are added to the list.

Papers:

1. Andreas E. Dalsgaard, Rene R. Hansen, Kenneth Y. Jrgensen, Kim G. Larsen, Mads C. Olesen, Petur Olsen and Jiri Srba, OPAAL: A Lattice Model Checker. In Proceedings of NASA Formal Methods, Lecture Notes in Computer Science, 6617. 2011

2.5 JToRX

Participants

- Axel Belinfante, University of Twente, NL;
- Lars Frantzen, ESI, NL;

Challenge JTorX is a tool that supports automatic test case generation, execution and evaluation from a requirements model [4, 1, 35]. As reported in Quasimodo Deliverable 5.8, JTorX has been extended during the Quasimodo project, to be able to handle transitions with parameterized transition labels. In that deliverable we reported how the extension enabled the use of JTorX for timed testing (by allowing JTorX to use the ta2torx tool to access timed models)

We have further integrated JTorX with other tools, and a connection has been made made between Lars Frantzen's STS library (STSimulator), such that JTorX can use STS models. Moreover, also during the Quasimodo project, an XML interchange format for STSes has been defined, and STSimulator has been extended to support this XML interchange format for STSes.

STSimulator is a prototype Java library to simulate Symbolic Transition Systems (STSs) [35, 18, 17]. STSs are transition systems with an explicit notion of data, and data-dependent control flow (somewhat similar to UML state machines). They are a well studied formalism for modeling and testing reactive systems. Using this library, common on-the-fly testing algorithms for transition systems can be easily implemented. There are two ways to specify an STS model for STSimulator. The first way is to specify it in Java using an API that STSimulator provides for this purpose. This has the disadvantage that a program that uses the STSimulator has to be recompiled for each new model. The second way uses the above-mentioned support for the XML interchange format for STSes. This allows us to have a single (model-independent) program that reads an STS model from a given (XML) file.

Results The connection between JTorX and the STSimulator has been made as follows. JTorX contains several interfaces that allow it to use external tool components to perform certain functions. The so-called torx-explorer interface allows it to use an external component to access the LTS, STS or timed transition system of a given model. The so-called torx-instantiator interface allows it to use an external component to obtain concrete instances for parameterized transition labels that are to be used as stimulus. Both ta2torx and STSimulator are connected to JTorX via the torx-explorer interface. Because STSimulator provides two functions: providing access to an STS, and providing concrete instances for parameterized transition labels, the torx-explorer interface was extended during the work on integrating JTorX and STSimulator, such that both these functions can be accessed via a single interface.

The integration of JTorX with STSimulator took place towards the end of the Quasimodo project. Therefore, with the JTorX+STSimulator combination only a proof-of-concept experiment has been done in one of the case studies in Quasimodo: the Hydac case study. One component of the tool-setup for this case study—a component that had specifically been written for the case study—offers functionality that is very similar to the functionality that is offered

by the JTorX+STSimulator combination. We thus explored wether JTorX+STSimulator can be used as a replacement for this component, and this turned out to be very well possible, although at first try JTorX performed much slower than the tool component that it replaced. The slow performance was caused by visualizations that JTorX runs by default, and by the performance penalty caused by outputting diagnostics and debug information. Disabling the visualizations and the diagnostics and debug output made JTorX+STSimulator perform with similar speed as the tool that it replaced.

2.6 UPPAAL Tron

Participants

- Marius Mikucionis, Brian Nielsen, AAU, DK.

Contribution: Uppaal-TRON has hitherto functioned as a stand-alone command-line based tool. To make it more accessible and easier to use we have developed a GUI and plugged this into the Uppaal GUI to create an integrated environment for modelling, simulation, verification as well as testing.

The procedure for using Uppaal-TRON is:

1. Model the system to be tested as a Uppaal timed automata network (closed system)
2. Partition the processes in the network into a part modelling the environment assumptions and another part modelling the SUT requirement. These two parts communicate via channels declared observable (possibly having variables attached for value passing). Both parts are required and assumed to be input enabled.
3. Develop the SUT adaptor that translates abstract model events to and from physical system stimuli. This also informs Uppaal-TRON about the observable actions that it understands (hence provide the information for the partitioning), the mapping of model-time-units to physical time units, and desired maximum test duration.
4. Run the test, by invoking Uppaal-TRON from the command-line giving it model-file and adaptor as arguments, as well as an extensive number of other options for controlling delay strategy, the amount of information in log-files and diagnostic information, and other engine parameters. The output is a textual log file with the interaction (and state set-information) between the tester and SUT, and a verdict
5. Analyse the log files; particularly when the verdict is fail, it must be determined what cause the failure which may be located in the model, SUT, or in the adaptor (an in principle also tin Uppaal-TRON).

The Uppaal-TRON GUI is added to Uppaal as another tab along with the existing editor, simulator and verifier tabs. When activating the tron tab, the user is presented with a new screen with one of three sub-panels for respectively partitioning, configuration, and test execution (See Fig. 6).

Partitioning Panel: Here (See Fig. 7) the user interactively partitions the processes of the model, and is helped to create a consistent partitioning (there is a few semantic rules that the partitioning of the model into an environment and SUT part must satisfy). The user is presented with a list of the processes in the system. By selecting one, the GUI presents the channels (inputs and outputs) used by the selected process. From this list the user can select a channel that should be an observable input or output, and add it to the corresponding list. While doing so a graphical

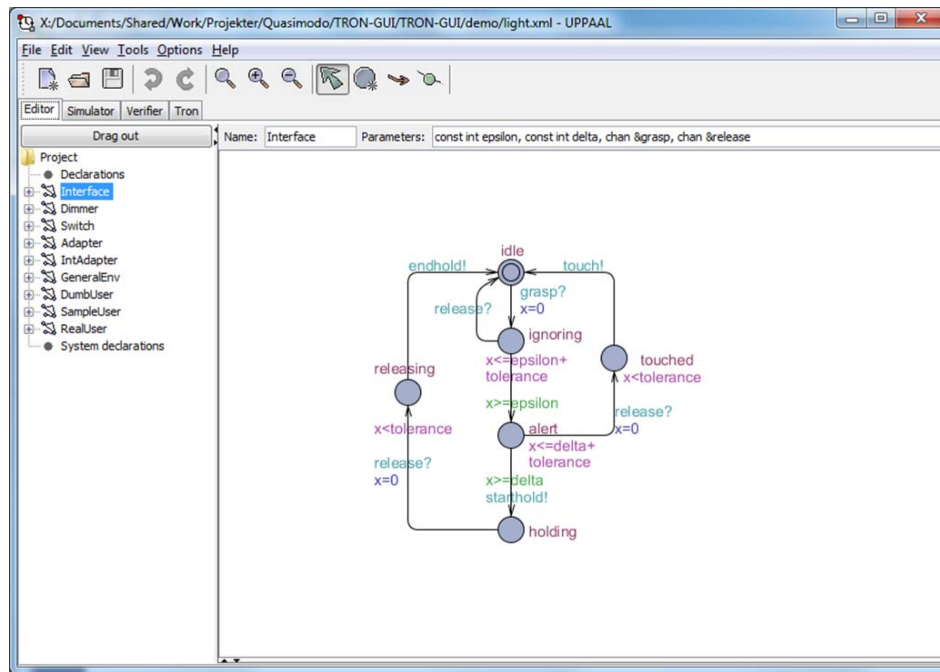

Figure 6: Main UPPAAL Window

overview of the partitioning that shows the environment processes, SUT model processes, and the observable actions is updated. The user may select an observable action and attach model variables to the actions (for value passing). A list of variables is shown in a (searchable) popup-menu. When the user has selected the desired set of observable actions, he can auto complete the partitioning. The GUI (via the engine) inspects of the dependencies and data flow in the model, such that each process either belongs to the environment part, or the SUT part. If it cannot be completed consistently, the GUI gives an error message.

Configuration Panel: This panel presents an overview of the various configuration settings that the user may and must set. These include different timing (in particular test duration and a concretization of model-time-units) and delay selection options, engine and adaptor options, and log options, See Fig. 8.

Test Execution Panel: The execution panel presents the user with an overview of test execution as this progress. The user may start or interrupt the test, and inspect the tool outputs and error/explanatory messages. Importantly, it displays the test execution as a message sequence chart (MSC) enabling much easier comprehension of the test run compared to inspection of a textual log file. The layout of the MSC can be controlled in a number of ways. First, it can either be displayed using either a real-time scale (where events are separated by their real-time occurrence difference as in Fig. 9), or an event-scale (where events are listed using a uniform distance, and the axis contains the time stamp as in Fig. 10). Further, the time axis may be shown

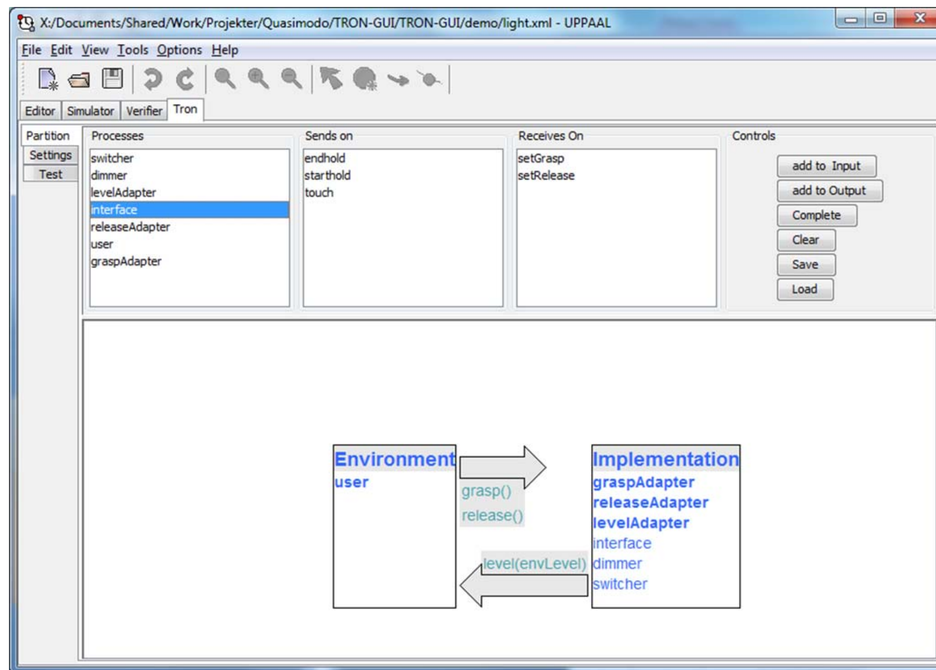

Figure 7: Partitioning Panel

in terms of model-time units or physical time units at various scales. Tool-tips are used to display detailed timing information about events.

Configuration File: Previously, configuration data was distributed in several places. Now, it is presented in the GUI in a systematic way, and all parameters and settings related to a test is collected and stored in a single (XML formatted) file that may be loaded and saved using the GUI. This makes it easier to manage and run test campaigns and variations thereof.

In conclusion, the GUI improves the accessibility and usability of Uppaal-TRON that assists model partitioning, test configuration, visualization of test execution, and management of test campaigns.

Future work includes a diagnostics panel aiming a visualising why a test failed. However, this is not straightforward because 1) point of failure may not coincide with point of fault, 2) because Uppaal-TRON works in states sets (potentially large sets of symbolic states) it is a challenge to find a good way of visualizing these, and the actions enabled/disabled in these, and online executions may be quite long. An interactive animation of the model with the user serving as SUT would be helpfull. Finally, it does currently not check the input enabledness assumption.

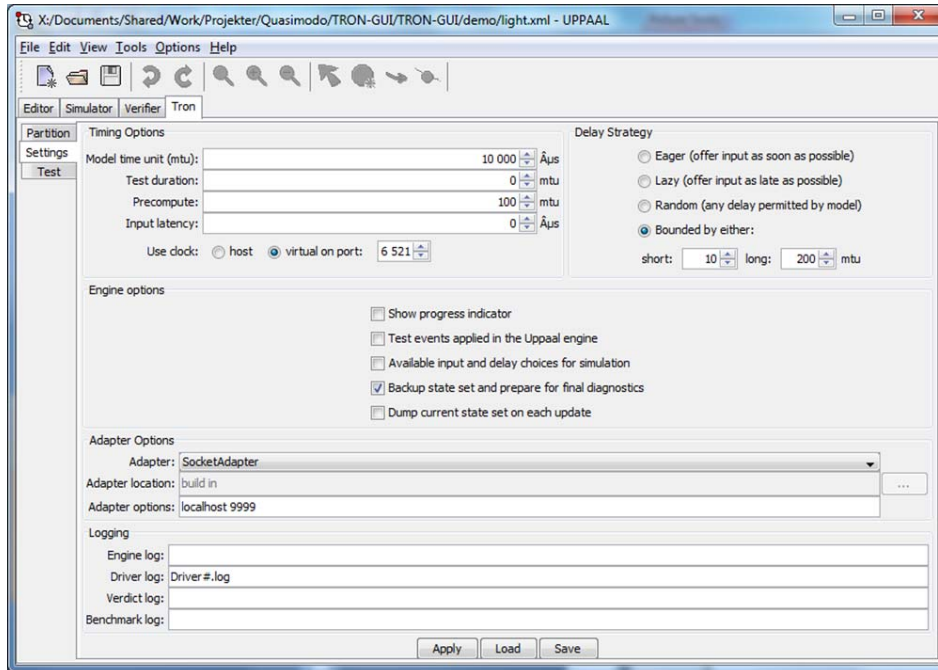

Figure 8: Configuration Panel

3 Tool Components

A goal of Quasimodo has been to contribute with a number of unique tool components for quantitative analysis and validation of aspects relevant for the various phases of the development process ranging from requirement capturing and design to implementation and testing. The Quasimodo tool components are available as plug-ins allowing for exploitation from existing commercial embedded tools, e.g. SimuLink, Rhapsody, Scilab/Scicos.

The development of tool components envisaged in the Quasimodo DoW (Description of Work) is given in Fig. 11. As indicated, all component promissed has been completed. Below we provide a detailed account on the progress made during the three years.

Quantitative refinement checking: During the first year algorithmic support for probabilistic and stochastic simulation has been given and evaluated in [8, 41]. In the setting of timed automata, encodings of a variety of refinements into timed game problems has been given in [9]. During the second year this encoding has been replaced with a direct, on-the-fly implementation in UPPAA-TIGA, allowing for direct checks of simulations between timed automata and timed game automata. Also a specification theory based on timed I/O automata has been developed with tool support for refinement and consistency checking implemented in UPPAAL-TIGA.

Worst-case analysis component The tool METAMOC applies value-analysis to (C-) programs to construct an abstract TA model. The program model is combined with suitable TA

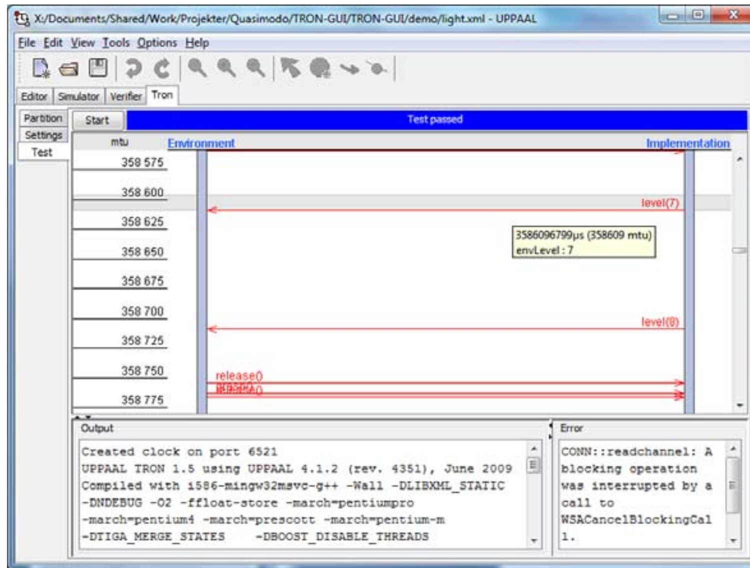

Figure 9: Execution panels (real-time layout of events, time in model time units)

models of the execution platform's pipeline, caches and main memory allowing modeling checking with UPPAAL to determine the worst-case execution time.

Schedulability analysis A modelling framework for schedulability analysis of embedded applications on multi-processor architectures has been made in [14]. The framework includes a rich collection of attributes for task (best- and worst-case execution times, minimum arrival times, deadlines and priorities), dependencies between tasks, assignment of resources, a variety of scheduling policies (including FIFO, EDTa dn FPS), and possibilities of preemption. The framework for schedulability analysis [14] has been extended with accurate modeling of resources [28] leading to more accurate estimates of worst-case blocking-times than classical worst-case response-time analysis.

The tool SARTS allow for schedulability analysis of safety-critical Java programs executing on the JOP Java processor.

Quantitative discrete simulation The discrete time simulation implemented in UPPAAL-TIGA provides the basis on which an upcoming discrete simulation engine for timed automata will be based. Furthermore, MPCTA is currently being extended with a discrete event simulation component that is specifically tailored to a sound handling of non-determinism.

Code generation Substantial work towards the P2J compiler translating from Promela (Promela being the modeling language of the finite-state model checker SPIN) as source language to Java as target language. Emphasis has been on a semantic preserving translation. More details can be found in the deliverable D3.6.

Code generation for timed specification is obtained by the tool chain described in deliver-

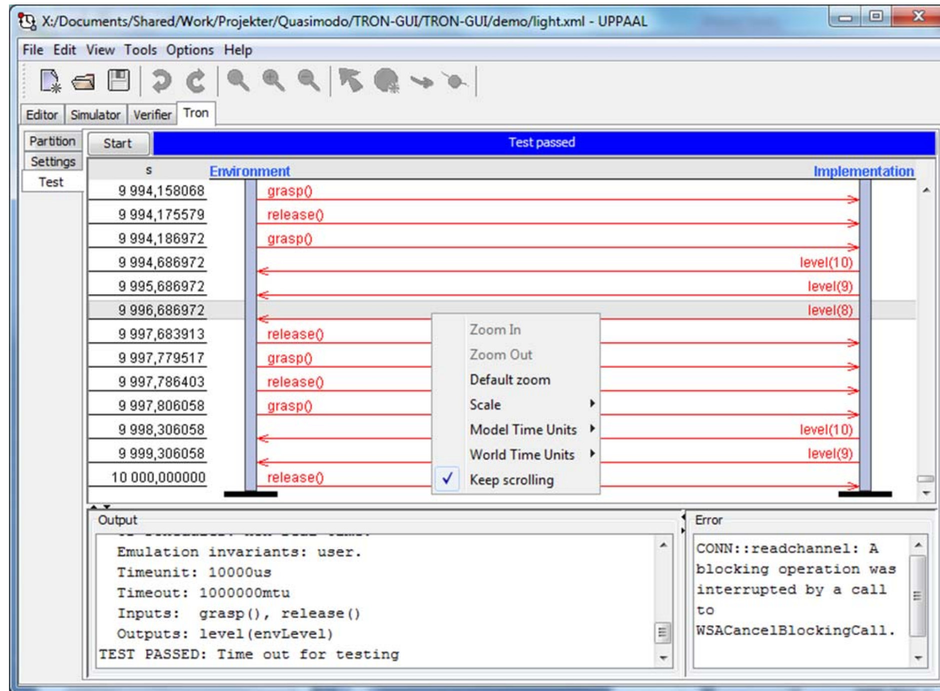

Figure 10: Execution panel (event-based layout, time in seconds)

able D3.7 where strategies obtained from timed games and given control objectives using UPPAAL Tiga are automatically translated into S-functions allowing for subsequent simulation and code generation using Simulink.

Controller synthesis under partial observability: An implementation in UPPAAL-TIGA of the algorithm in [10] for synthesis of control strategies under partial observabilities has been made in [12].

Implementability checking: In [11] an implementation in UPPAAL of the algorithm for robustness analysis of timed automata from [15] has been made. That is, the correctness of the model is analysed in the presence of small drifts on the clocks.

Optimal scheduling: Optimal scheduling has been performed substantially using UPPAAL-CORA. In particular, the application of agent-based search engines in [34] has proven particular succesfull. In a similar manner [31] provides heuristic-guided search which leads to designated goal-states substantially faster than the standard UPPAAL verification engine.

On-line real-time testing: Both UPPAAL-TRON and TORX provide algorithmic support for on-line real-time testing [21].

Off-line real-time testing Using UPPAAL and UPPAAL-TIGA off-line generation of strategies for conformance testing of real-time systems has been pursued for increasingly expressive classes of timed automata specifications [13, 32].

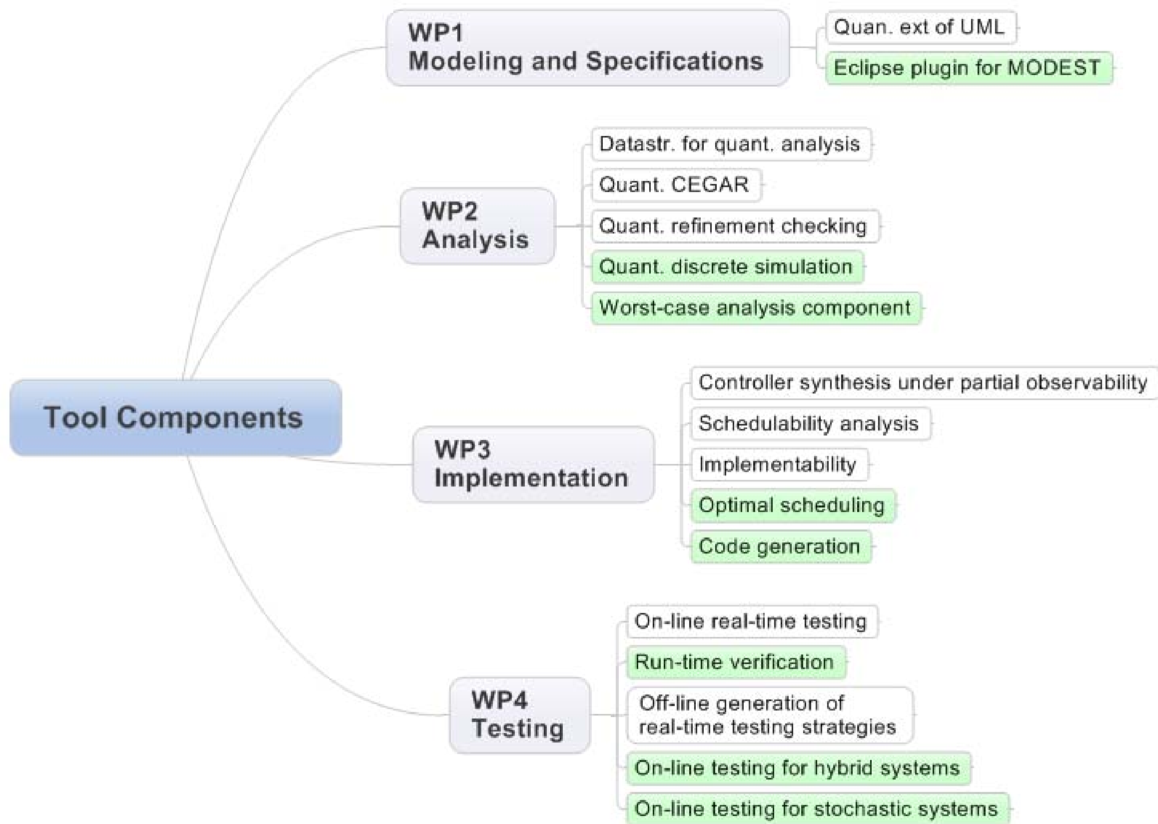

Figure 11: Quantitative Tool Components. Light Components: developed during first two year. Green Components: developed during the third year.

Run-time Verification The on-line engines of UPAAL Tron and JToRX may be used for run-time verification of an implementation as this corresponds to the simplied setting where the tools are used as simple monitors.

On-line testing for hybrid systems: As detailed in deliverable D4.6 UPPAAL Tron may be used for conformance testing Simulink models (e.g. non-linear dynamical systems) against timed automata specifications. The framework requires exchange of integer variable values of UPPAAL with real-valued signals of Simuling, synchronization of distrete timed automata events with continuous signals, as well as time synchronization between the tools.

Complementatry to conformance testing Simulink models, co-simulation of UPPAAL Tron with Simulink as well as the tool Phaver/SpaceEX are supported.

On-line testing for stochastic systems: Extensions of the theory of ioco testing has been made in which the non-deterministic selection between alternatives are replaced by an explicit probabilistic selection.

The support of statistical model checking – as implemented in UPPAAL SMC – may be

used for generating test input sequences in accordance with an assumed distribution expected of the environment of a systems, thus providing the basis for performance evaluation and reliability testing.

4 Integration between Tools

We give in this section the final status on the progress made towards interaction and integration between tools of the consortium and between tools of the consortium and external/commercial tools, e.g. PRISM, Möbius, Matlab/Simulink, Scilab/Scicot, and UML/Rhapsody.

4.1 Linking Probabilistic Tools

PRISM[2] is a leading (external) tool for model checking and validation of probabilistic systems with numerous applications ranging from correctness of security protocols, efficiency of wireless protocols, reliability of nanotechnology designs, to the analysis of signalling pathways. Obviously, a number of links for interaction between PRISM and Quasimodo tools has been made. In addition, given the support of probabilistic timed automata in MODEST and the stochastic semantics introduced for timed automata in UPPAAL SMC translations between MODEST and UPPAAL models have been made.

In addition to a number of links for interaction between PRISM and Quasimodo tools translation bet addition to these a

MODEST, MCPTA:] The modeling formalism of MODEST is extremely expressive and translations from models in (a subset of) MODEST to PRISM. This translation has during year 2 been exploited to allow for the analysis of the fragment of MODEST corresponding to PTA using PRISM. In a similar manner MODEST models may be translated to Möbius allowing for performance analysis using simulation.

MRMC and INFAMY:] Interfaces between PRISM and MRMC as well as INFAMY have been made allowing PRISM models to be fed into and analyzed by the novel engines of these tools.

UPPAAL and MODEST: The major obstacle is how to overcome the differences in synchronization mehcanisms and the limited support for discrete datastructures and user-defined functions in MODEST. The synchronisations in UPPAAL are either shake-hand between two processes or broadcast with one designed sender and several receivers. In addition, the semantics of changing states when taking a transition follows the sequential semantics of procedural programming languages where statements are evaluated sequentially and modify states accordingly.

MODEST is a high-level modelling language for stochastic timed systems. It has a formal semantics in terms of stochastic timed automata, which include, as special cases, several other widely-used models such as timed or probabilistic automata. Modest processes in parallel compositions synchronise CSP-style over the actions in their shared alphabets, and assignments when changing states are treated as update functions and thus executed atomically. Support for model-checking and simulation of Modest models is provided by the Modest Toolset.

To link Modest and UPPAAL, we have faced some major obstacles concerning the semantics of computing new states and the way synchronisations are made. One first step we have taken is

[2]http://www.prismmodelchecker.org/

to implement an alternate semantics in UPPAAL to support CSP-like synchronisations associated with a different type of update(parallel instead of sequential). This change in semantics still allows us to use the full expressive power of the UPPAAL language, except that we have a different way to synchronise.

In the other direction, our first step has been to define and implement binary shake-hand and broadcast synchronisation in Modest and the Modest Toolset, including the sequential update semantics for these kinds of synchronisation. These changes are extensions, not modifications, to the Modest language.

With the above extensions allowing to bridge between the differences in synchronisation and update semantics models in both formalisms can be translated into the other, extending the available analysis options for both tools. A prototype of a MODEST-to-UPPAAL translator and a draft description of the translation procedure from UPPAAL to Modest are being made.

4.2 Linking to Matlab/Simulink

Matlab/Simulink being the de facto tool used by European industries a main goal of Quasimodo is to seek integration with this tool to increase impact. Despite the leave of the industrial partner Incron Gmbh, and unsuccessful attempt of direct collaboration with Mathworks, Quasimodo pursuse this direction fully and aims at providing several links to Matlab/Simulink. Below we give a status on the results obtained with references to deliverables D3.7 an D4.6 for more detailed information:

UPPAAL Tiga and Simulink: Extending the succesfull application of controller synthesis carried out on the HYDAC case study we have developed framework and implemented a tool chain, which – given a user defined timed game model in UPPAAL Tiga – allow for winning strategies to be automatically imported to Simulink as an S-finction allowing for simulation, validation and automatic generation of code. We refer to deliverable D3.7 section 2 gives a detailed description.

UPPAAL Tron and Simulink: We have developed an implemented a framework allowing for linking UPPAAL Tron and Simulink models. The framework can be used in several ways. In particular, in testing conformance of a real-time system with respect to a timed automata model one may augment the model with dynamic behavior using co-simulation by Simulink for environment emulation purposes (see Fig. 12. Also, the framework can be used to test conformance of Simulink models against timed automata specifications. We refer to deliverable D4.6 Appendix A for a detailed description.

4.3 Linking to UML State-Charts

As part of the YGDRASIL tool chain AAU has implemented a translation from UML state-charts to timed automata involving the tools Rational Systems Developer and UPPAAL. The translation is done by first exporting the UML state-charts into the standard XMI format, and then translating

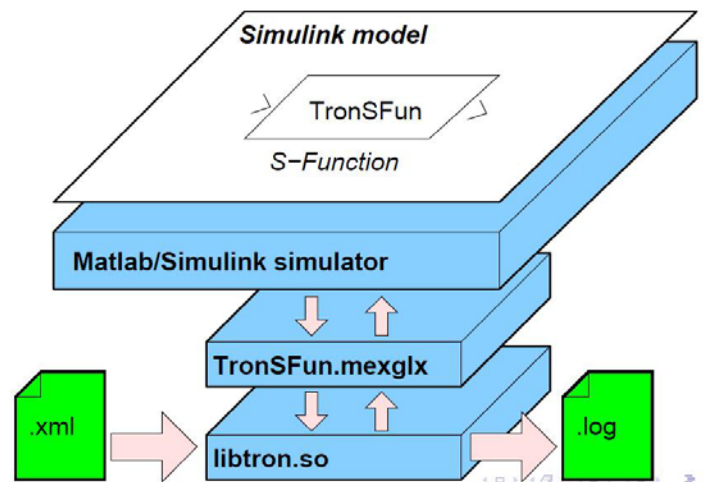

Figure 12: Co-simulation of Simulink and UPPAAL-TRON.

into the UPPAAL XML format. The translation will permit verification as well as automatic test generation from state-chart models, and is foreseen to be applied in the industrial case-study of Terma.

4.4 Towards a Unifying Framework for Symbolic Model-Based Testing

Participants

- Lars Franzen, ESI, Eindhoven, NL.

Current Situation and Goal

When doing model-based testing one is confronted with a vast variety of tools and modeling languages. This makes it hard to switch to a new tool or language, or to compare different approaches. Furthermore one has to always learn a new language and tool philosophy.

Ideally there would be just one modeling language within a unifying tool framework. To reuse the existing tools they must be integrated into the framework, and the language used must be translated into the language of each tool being part of it. As a consequence, the language must be able to subsume all features of the languages to which it translates. There is a broad variety of such features. Looking at the characteristics a modeling language can specify we can identify for instance:

- functionality,
- time,
- continuous behavior.

Looking at the modeling features the language offers we can identify for instance:

- graphical or textual,
- state-based or process algebraic,
- synchronous or asynchronous,
- single process or communicating processes,
- flat or hierarchical modeling,
- deterministic or nondeterministic.

This partial lists already show that it is a more than ambitious goal to come up with one language and tool environment which is capable of dealing with all these aspects in an accessible manner.

Goal and Setup To make some first realistic steps towards such a unifying framework we started by what we already have had developed, both theoretically and implementation-wise. Three tools we had at hand:

- JTorX [6],
- TorXakis,and

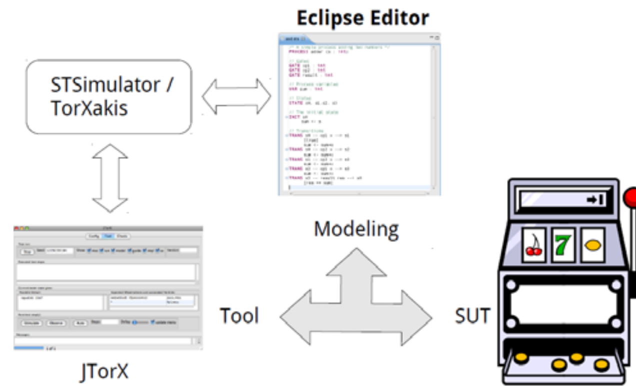

Figure 13: The Tool Setup

- STSimulator [25].

All these tools are based on the *ioco* testing theory [38]. JTorX does offer a state-of-the-art graphical user interface to initiate and control the testing process. Furthermore it implements a sound and complete testing algorithm for *ioco* and well defined interfaces interact with other components. TorXakis and STSimulator are both simulators for symbolic models called Symbolic Transition Systems (STS) [18], which can serve as the specification model within *ioco*. Such an STS specifies the functionality of a system in a state-based, asynchronous, and potentially nondeterministic manner. The first goal was to connect TorXakis and the STSimulator to JTorX, such that JTorX can be used as the graphical frontend to do model-based testing via STS specifications.

One drawback of both TorXakis and the STSimulator is that they use very special modeling languages—each has a modeling API in the lanuage the tool is implemented in, i.e. Haskell resp. Java—which are not very accessible in general. Thus, a more common and convenient modeling language became a second goal. We chose the Xtext [27] language development framework, which is based on the Eclipse Modeling Framework [23]. With Xtext one can define the grammar of a Domain Specific Language (DSL), and Xtext generates a full blown editor for it. Furthermore, Xtext can be used to interpret or transform the DSL model. Figure 13 shows the setup. Summarizing, we intended to start with modeling and testing functional aspects via symbolic, state-based models. The GUI for the testing process itself is provided by JTorX. As model simulator and data selector either TorXakis or the STSimulator can be used. The model itself is written in an editor generated by Xtext.

The Tools

In this section we give some more detail for the three tools JTorX, TorXakis, and STSimulator. For Xtext we refer to the website [27].

JTorX For an overview of JTorX we refer to [5].

TorXakis TorXakis is based on the model-based testing tool TorX [39] extending it with symbolic test generation capabilities [17]. TorXakis is implemented in Haskell[3].

The STSimulator Library [25] is implemented in Java. Firstly, it allows to construct STS objects. In its current version, the following simple data types are supported: `Boolean`, `Integer`, `String`, `Money`, `Date` and `Enumeration`. Simple types can be combined to yield lists or complex types, but not in a recursive manner. To express guards, a straightforward language can be used, which offers the common operators known from other programming languages. To visualize the STS, the simulator can produce the input format for the DOT [24] tool.

Once an STS object is created, it is passed to a simulator object, which sets it into its initial state. Now one can give inputs or outputs to the simulator, which computes the next set of possible states (due to nondeterminism, there may be several). A typical computational task here is deciding if a guard is true for the current valuation of the STS-variables. This can be done by the underlying interpreter using simple evaluation. The simulator can also be used to generate feasible inputs or outputs for the current state on its own. Here solutions to guards have to be computed, which is achieved via the consultation of an external constraint solver.

Modeling Symbolic Transition Systems in XML

To have a simple format to interchange STSs between tools an XML schema for STSs has been defined, see [2]. The current Version 1.0 does has a few limitations with respect to the STSimulator library:

- no custom initial value of location variables
- no definition of complex types
- no definition of lists

For an example of how to model an STS in XML we refer to [16].

Connecting JTorX with STSimulator

Results

Within Quasimodo we have achieved the following results:

- Definition of an XML interchange format for STSs
- Definition of an Xtext DSL for single STSs
- Mapping of the Xtext STSs to the XML format
- Extending JTorX to understand the STS XML format

[3] `http://www.haskell.org`

- Interfacing the STSimulator with JTorX

Together we have achieved a first complete development chain:

1. Model the specification as an STS in Xtext
2. Automatically transform the model from Xtext to XML
3. Use JTorX to test the SUT via the STS in XML format, using the STSimulator for data selection

Application

We have applied the toolchain at the Hydac case.

Outlook

We have a first working chain of tools which are loosely coupled into a framework which is based on Xtext on the modeling end, and JTorX on the test execution end. To interchange data the XML format has proven beneficial.

To make this proof of concept into an industrial strength framework several improvements can be conceived. From a modeling point of view not just a single STS, but several communicating STSs are desirable in many situations. The *ioco* theory and the tools are already prepared to do so, just the Xtext modeling part needs to be extended in this direction.

Another, often mandatory modeling feature is time. Here the *TRON* [26] tool is a promising candidate to be integrated into the framework. To make the whole framework a bit less loosely coupled by defining more rigorous interfaces, or unifying the tools under the EMF, might be another direction to go. Finally, hierarchical modeling is often handy to structure the models. Here approaches known from the Unified Modeling Language (UML) [33] are interesting as an extension point for the current setting, especially when also using graphical models.

References

[1] JTorX webpage, August 2009.

[2] STSchema Version 111110. `http://www.frantzen.info/uploads/schemas/STSchema111110.xsd`.

[3] C. Baier, F. Ciesinski, and M. Größer. PROBMELA: a modeling language for communicating probabilistic processes. In *Proc. of the 2nd ACM/IEEE Int. Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 57–66. IEEE, 2004.

[4] A. Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference (TACAS 2010)*, volume 6015 of *LNCS*, pages 266–270. Springer, March 2010.

[5] A. F. E. Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Paphos, Cyprus*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270, Berlin, March 2010. Springer Verlag.

[6] Axel Belinfante. Jtorx: a tool for on-line model-driven test derivation and execution. In *To appear in TACAS 2010*, 2010.

[7] S. C. C. Blom and J. C. van de Pol. State space reduction by proving confluence. In *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 596–609. Springer, 2002.

[8] Jonathan Bogdoll, Holger Hermanns, and Lijun Zhang. An experimental evaluation of probabilistic simulation. In *28th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 5048 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 2008.

[9] Peter Boulychev, Tomas Chatain, Alexandre David, and Kim G. Larsen. Playing games with timed games. Under submission, 2009.

[10] Franck Cassez, Alexandre David, Kim Guldstrand Larsen, Didier Lime, and Jean-François Raskin. Timed control with observation based and stuttering invariant strategies. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *ATVA*, volume 4762 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2007.

[11] Alexandre David, Piotr Kordy, Kim G. Larsen, and Jan Willen Polderman. Practical robustness analysis of timed automata. Under submission, 2008.

[12] Alexandre David, Kim G. Larsen, and Didier Lime. Uppaal-tiga 2009: Towards realizable strategies. Under submission, 2009.

[13] Alexandre David, Shuhao Li, Brian Nielsen, and Kim G. Larsen. A game-theoretic approach to real-time system testing. In *DATE*, pages 486–491. IEEE, 2008.

[14] Alexandre David, Jacob I. Rasmussen, Kim G. Larsen, and Arne Skou. *Model-based Framework for Schedulability Analysis Using UPPAAL 4.1*. Taylor ad Francis, 2009. To appear in CRC Press Book on "Model-Based Design of Heterogeneous Embedded Systems".

[15] Conrado Daws and Piotr Kordy. Symbolic robustness analysis of timed automata. In Eugene Asarin and Patricia Bouyer, editors, *FORMATS*, volume 4202 of *Lecture Notes in Computer Science*, pages 143–155. Springer, 2006.

[16] L. Frantzen. Modeling symbolic transition systems in XML. `http://www.frantzen.info/archives/7-Modeling-Symbolic-Transition-Systems-in-XML.html`.

[17] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *FATES 2004*, number 3395 in LNCS, pages 1–15. Springer-Verlag, 2005.

[18] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A symbolic framework for model-based testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *FATES/RV 2006*, number 4262 in LNCS, pages 40–54. Springer-Verlag, 2006.

[19] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *Proc. of the 17th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6605 of *LNCS*, pages 372–387. Springer, 2011.

[20] J. F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Proc. of Algebra of Communicating Processes*, Workshops in Computing, pages 26–62. Springer, 1995.

[21] Anders Hessel, Marius Mikucionis, Brian Nielsen, Paul Pettersson, Arne Skou, and Kim G. Larsen. *Testing Real-Time Systems Using UPPAAL*, volume 4949 of *LNCS*. 2008.

[22] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Proc. of the 12th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[23] Eclipse Modeling Framework homepage. `http://www.eclipse.org/modeling/emf/`.

[24] Graphviz homepage. `http://www.graphviz.org/`.

[25] STSimulator homepage. `https://stsimulator.dev.java.net/`.

[26] TRON homepage. `http://www.cs.aau.dk/~marius/tron/`.

[27] Xtext homepage. `http://www.eclipse.org/Xtext/`.

[28] Jacob Illum, Kim G. Larsen, Marius Mikucionis, and Steen Palm. Model-based approach for schedulability analysis. Intern Report.

[29] J.-P. Katoen, J. C. van de Pol, M. I. A. Stoelinga, and M. Timmer. A linear process-algebraic format for probabilistic systems with data. In *Proc. of the 10th Int. Conf. on Application of Concurrency to System Design (ACSD)*, pages 213–222. IEEE, 2010.

[30] J.-P. Katoen, J.C. van de Pol, M.I.A. Stoelinga, and M. Timmer. A linear process-algebraic format with data for probabilistic automata. *Theoretical Computer Science*, 2011 (to appear).

[31] Sebastian Kupferschmid, Jörg Hoffmann, and Kim G. Larsen. Fast directed model checking via russian doll abstraction. In *Proceedings of TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, 2008.

[32] Shuhao Li, Alexandre David, Kim G. Larsen, and Brian Nielsen. Cooperative testing of uncontrollable timed systems. In *Fourth Workshop on Model-Based Testing (MBT'08)*, March 2008.

[33] Object Management Group. *UML 2.0 Superstructure Specification*, ptc/03-08-02 edition. Adopted Specification.

[34] Jacob Illum Rasmussen, Gerd Behrmann, and Kim Guldstrand Larsen. Complexity in simplicity: Flexible agent-based state space exploration. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2007.

[35] STSimulator webpage, August 2009.

[36] M. Timmer. Scoop: A tool for symbolic optimisations of probabilistic processes. In *Proc. of the 8th International Conference on Quantitative Evaluation of SysTems (QEST)*, 2011, to appear.

[37] M. Timmer, M. I. A. Stoelinga, and J. C. van de Pol. Confluence reduction for probabilistic systems. In *Proc. of the 17th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6605 of *LNCS*, pages 311–325. Springer, 2011.

[38] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[39] J. Tretmans and E. Brinksma. TorX : Automated Model Based Testing. In A. Hartman and K. Dussa-Zieger, editors, *First European Conference on Model-Driven Software Engineering*. Imbuss, Möhrendorf, Germany, 2003.

[40] J. C. van de Pol and M. Timmer. State space reduction of linear processes using control flow reconstruction. In *Proc. of the 7th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, volume 5799 of *LNCS*, pages 54–68. Springer, 2009.

[41] Lijun Zhang, Holger Hermanns, Friedrich Eisenbrand, and David N. Jansen. Flow faster: Efficient decision algorithms for probabilistic simulations. *Special Issue on TACAS 2007, Logical Method in Computer Science (LMCS)*, 2008.