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

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

Project Acronym: **QUASIMODO**

Deliverable no.: D4.6

Title of Deliverable: On-line hybrid/stochastic testing

Contractual Date of Delivery to the CEC:	**Month 30**
Actual Date of Delivery to the CEC:	**May, 2011**
Organisation name of lead contractor for this deliverable:	
Author(s):	
Brian Nielsen, Marius Mikucionis	
Holger Hermanns, Hernán Baró Graf	
Participants(s):	**P04 RWTH**
Work package contributing to the deliverable:	**WP 4**
Nature:	**R**
Version:	**1**
Total number of pages:	**22**
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:

This report documents progress on stochastic and hybrid online model-based testing. We extend our online testing tool UPPAAL-TRON to testing from models with mixed continuous and discrete (time dependent) signals by exploiting existing UPPAAL facilities, and by integrating with external tools for signal generation and monitoring. In context of the online testing tool TorX we examine how to perform scheduler based probabilistic selection of tests from usage and implemenation profile models.

Keyword list: Test generation, hybrid automata, Probabilistic testing, Simulink, PhaVer, Uppaal-Tron.

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

Model-based testing aims at checking whether the behavior of a (physical) system under test (SUT) is correct with respect to (conforms-to) its specification by executing the SUT, thus complementing verification and model-checking of models.

In previous Deliverable D4.1 we extended the underlying theory for testing quantitative systems, and in Deliverable D4.2 we presented our contributions to on- and offline testing to systems where real-time is a key property. In D4.3 we presented coverage based test generation and coverage metrics.

This deliverable reports on our results towards enabling (online) testing based on models that have, or require, probabilistic properties or hybrid (mixed discrete and continuous signals).

The deliverable focuses on online testing where tests events are generated, executed and evaluated event-by-event. This is opposed to offline testing where a test suite is first generated and then executed. An online algorithm dynamically computes the state-set that the model (and in the case of relativized conformance, the system model combined with its environment model) could possibly occupy after the (timed) observations made so far. As execution progresses outputs are checked against the legal actions in the current state set, based on a precise formal notion of correctness. Similarly, the next input test input event is typically chosen randomly (with equal probability) from the inputs enabled in the current state set. A more detailed discussion and comparison of the properties of the two methods were presented in Deliverable 4.3.

2 Testing Hybrid Systems

Participants

- Marius Mikucionis,AAU
- Kim G. Larsen, AAU
- Brian Nielsen, AAU

The goal is to advance techniques for (online) testing of hybrid systems. By definition, an important characteristic is that these systems have a mix of (time sensitive) discrete signals and evolutions of continuous variables (e.g., temperature, pressure, speed and physical position etc.).

2.1 Challenges

Consider an example of the water level in two water containers, each from which water is being consumed with a specific rate. The system also has a pump which can supply water to one container at a time (with a certain rate). The water level must be greater than a critical level. The water levels (v_1, v_2) may be captured by the hybrid automaton in Figure 1.

In a hybrid automaton the evolution of its continuous variables may be described by *any* continuous function/differential equation. In the restricted class of linear hybrid automata (LHA), the (first) derivative is (per location) bounded by an interval by two integers (e.g. $temperature' \in [2,3]$). Thus a timed automaton can be seen as a LHA where clocks are continuous variables with constant derivative of precisely one.

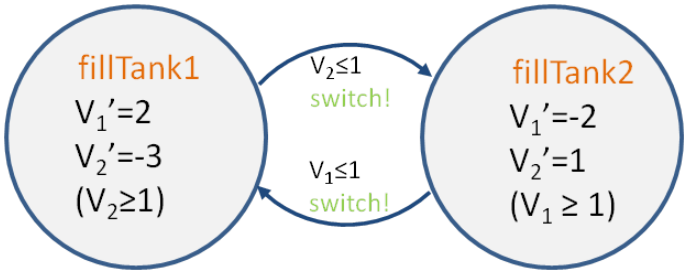

Figure 1: Simple (Linear) Hybrid Automaton.

Given a model that describes the hybrid behaviors and especially the evolution of continuous variables (typically through differential equations), and a suitable notion of conformance, there are two sub-problems from a testing perspective:

- Generating input signals (esp. trajectories) to be fed to the SUT
- Monitoring and evaluating outputs (esp. trajectories) by comparing those of the SUT with those permitted by the model

Whilst simulation of (deterministic) hybrid automata is computationally feasible (still requiring advanced (possibly imprecise) numerical computation algorithms), their formal analysis is difficult. In full generality, reachability analysis is undecidable. For linear hybrid automata it is semi-decidable (termination not always guaranteed), and requires use of comparatively costly polyhedra based data-structures algorithms.

Hence testing hybrid systems is challenging; note that also for online testing (esp. the monitoring aspect) where computing the set of reachable set of states (of possibly non-deterministic models and with timing-uncertainty) after a given history of observable actions, is a core operation (both in ioco/TorX and rtioco/Uppaal-TRON), makes in general use of a reachability sub-procedure.

2.2 Results

As noted, analysis of hybrid automata requires fairly advanced and specialized algorithms that cannot be achieved though a straightforward extension of those already in Uppaal. The required algorithms and data structures are more general and also much more costly, and don't fit easily in the existing Uppaal implementation.

Except the first 2 points listed below, research developments and collaborations have taken us in another direction emphasizing exploitation with external tools, where Uppaal-TRON is efficiently handling the discrete and timed signals and the continuous evolutions are computed and monitored by dedicated simulators and model-checkers. Hence, separate models are required for each tool/aspect. Such an integration may at first seem simple, but synchronization of time and of the models must be done carefully; the behavior of the

models are rarely completely independent, as they have to react to certain events/threshold crossings in the other model. Thus some communication and state tracking must be facilitated.

2.2.1 Encoding LHA using Stopwatches

UPPAAL has recently been extended with so-called stopwatches which are continuous clock variables that can be stopped (by setting its derivative to zero) and resumed (setting its derivative back to one), thus providing a means for integration over time. UPPAAL is able to perform an (*over-approximate*) reachability analysis for such timed automata with stopwatches. Earlier work [9] has shown that reachability analysis of linear hybrid automata can be reduced (via a translation) to reachability analysis of timed automata with stop watches. At the disadvantage of the translation reachability analysis may potentially be more efficient than the general polyhedra-based algorithms for LHA[9].

Since stopwatches are also supported in UPPAAL-TRON (resulting in a slightly over approximated state-set) this gives a path to testing LHA with the existing tool. It does require a small extension that it is possible to read the value of a clock (encoding a continuous variable) and send this to the SUT (and reverse); in the current implementation, this can only be done for discrete variables. This path has not been further implemented or evaluated.

2.2.2 Tracking signals using integer variables

UPPAAL-TRON supports discrete (integer) variables may be updated with regular time intervals defined by the clocks of the timed automaton models.

These values may be passed to and from the SUT (via its adapter) using channel synchronization with value passing. Further, UPPAAL-TRON supports non-deterministic choice, and especially integer-computation using the normal (simple) arithmetic operators like $\{+, -, *, /\}$, that may be composed into relatively complex functions using iteration and function abstractions using a C-like syntax. Hence, a continuous time varying signal may be approximated using discrete variables. These time-varying data values are recomputed on demand based on the clocks of the model, thus being in synchrony with the remaining timed automata model.

We have identified a number of modeling techniques and patterns [13, 12] that exploit these features to approximate a continuous time varying signal. These patterns include both generating input stimuli, and the more difficult evaluation of whether the output signal is contained in an envelope (over approximation) of the desired signal. The techniques have been successfully applied and demonstrated on an industrial case study of supermarket refrigeration controller where different environment temperature curves were generated, and the value of a PID-regulator was monitored. Figure 2 illustrates tracking of the temperature signal where the over-approximated values are connected to the TA model (capturing discrete and timed behavior. The exact evolution between these bounds is not checked (but may be by an external tool, see Section 2.2.4).

One goal is output evaluation, or for tracking the value of SUT internal continuous variables that affects enabled input/output actions. In most cases, it is not realistic to match the value precisely and the exact time it had that value, and thus it is important make a (tight) over-approximation in both the value and time domain.

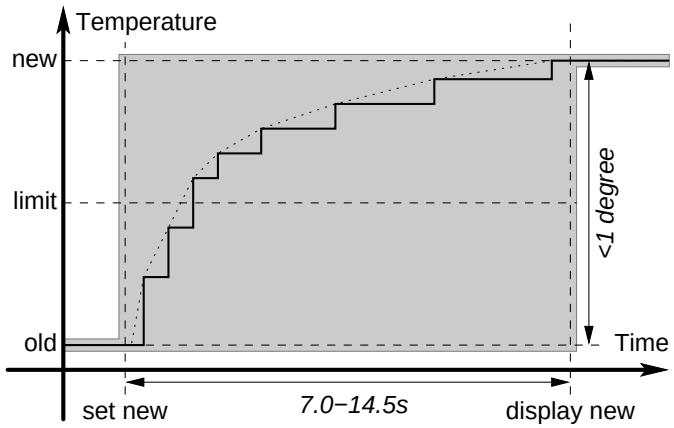

Figure 2: Tracking Temperature Interval (from [13]). Dotted lines represents the temperature, solid lines the sampled values, and the dashed perimeter the over-approximated values.

- One technique is to use the power of a non-deterministic choice to compute the allowed set of (integer) values/time-interval combinations. However, this may quickly lead to large state-sets, depending on the required accuracy.
- An alternative and often better technique is to model the signal as piecewise monotonic, use interval arithmetics and a pair of variables that respectively tracks the allowed lower and upper bound on the signal, and slides these bounds as time progresses. This is more efficient both in the possible state-set size and in the number of input/output events that the online test tool needs to process.

Figure 2 illustrates a scenario in the refrigeration controller. In the steady state the internally computed temperature is close to the injected environment temperature. When this change, and is observed by the controller, it gradually computes and outputs the calculated temperature. This is captured by the larger envelope (still tight for this application < 1 degree). The interval is collapsed when the temperature converges. In the scenario the temperature crosses a specific threshold (the limit in Figure 2that triggers other events (like engaging or disengaging the compressor or alarms).

Although these techniques work well in many cases, we do not find them generally satisfactory from neither efficiency or conceptual points of view.

2.2.3 Virtual Real-time Clock Framework

One of the challenges when testing real-time systems is that the test must executed in a timely fashion: the test input events must be supplied to the SUT at the time (or time interval) prescribed by the test case, test output events must be observed and time stamped accurately, and finally the tester and SUT must be in synchrony with respect to time.

Thus in real-time test execution in reality consists of executing two concurrent communicating real-time systems (the tester and SUT). Any disturbances in the scheduling of either system, or in their communication may lead to a false verdicts. This is often a problem when the SUT is software emulated on a host PC (typically running a non-real-time OS) and ditto for the tester. The problem is exacerbated when both are running

on the same PC. But even with dedicated test hardware and the embedded system running on target, time synchronization is problematic because both systems have their own independent private clocks that are typically not exactly synchronized.

To enable testing of timed properties in face of these settings we have developed a software solution in a form of a virtual clock framework that allows the SUT and test executor (in our case UPPAAL-TRON) to explicitly agree on time. Thus, the purpose of the virtual time framework is to provide "lab" conditions for testing software where the value of a global reference clock is controlled and detached from physical time. Such a framework enables testing of time delays specified in software in ideal conditions where the time spent on computation and communication is treated as zero. If the computation and or communication time is known and needed to be taken into account, then such delays can be replaced by "timed-wait" calls and an abstraction of control software can be tested under ideal conditions.

Time progresses for both parties when both have indicated to the framework by how much they are prepared to delay. The framework advances time by the maximum amount they can agree on. A direct positive effect of this is that test execution is (in most cases) much faster compared to physical time. The price of using the framework is that certain changes must be made to implementation to make it use the virtual clock. To make this easy for a wide range of implementations we build the framework to support the monitor programming paradigm within a subset of POSIX thread functions (Portable Operating System Interface 1003.1b-1993 real-time extension) which thereby makes it easy to adopt a large class of code with trivial function call replacement.

The Virtual clock assumes the following protocol:

- There is a single instance of a global clock in the entire test setup.
- All participating threads follow these rules:
  - Register its presence at the global clock.
  - All time-related system calls are redirected to the global clock.
  - Each thread should perform some computation and eventually call to wait for some condition to occur.
  - The thread computation time is assumed to be negligible and only the waiting times are significant.
- The global clock serves the threads in the following way:
  - If there is at least one thread computing and not waiting, then the clock value stays constant.
- If all threads are waiting for a condition to occur, then the global clock finds the smallest clock increment which would trigger a timeout for at least one thread, increments the clock and notifies appropriate threads.

Besides this original intention the virtual clock framework has served successfully for a number of applications:

- It has been very useful for demonstration purposes by running the demo on a single laptop, enabling demos to be given as part or ordinary presentations/talks.

- It has been used to perform testing of the (medium access protocol) in the chess WSN Node. The system under test is running on a PC host using emulated clocks and the virtual clock framework. If the testing were to be done at target, it would normally require a dedicated hardware test interface capable of high resolution precise timing.

- It has been used to interconnect UPPAAL-TRON with other model simulators like Phaver/SpaceEX (Linear Hybrid Automata), Simulink. In this way efficient co-simulation and refinement testing has been made possible.

Based on the many unforeseen applications and experiences from these we are now improving the framework towards a more general and more open framework that lessens the dependency on the concepts of the Posix/monitor model.

2.2.4 Co-simulation with external tools

We have developed an integration technique with other model-simulators of system aspects where UPPAAL-TRON is insufficient, and in particular of continuous signals.

The technique consists of 4 parts:

- A UPPAAL modeling pattern that enables the timed automata model to track and react to important events of the continuous models

- An augmentation of the adapter-framework to report bounds on observed variables, accompanied by an additional filter component in the engine to validate this bound.

- Optional use of the virtual clock framework.

Co-simulation with Simulink: In particular the technique has enabled integration with Simulink such that this tool can now be used for co-simulation with UPPAAL-TRON, such that (non-linear) dynamics can be simulated in Simulink whilst discrete/timed aspects are handled efficiently in UPPAAL-TRON. This has two applications. 1) Simulink can be used for the environment emulation (and monitoring of SUT outputs, if the user has encoded such a check in the Simulink model) part of conformance testing in conjunction with UPPAAL-TRON. 2) refinement testing, where a Simulink model (capturing system behavior at a low level of abstraction) is treated as the SUT, and checked against a more abstract properties specified as timed automata.

The integration is possible both in real-time and simulated-time mode.

The technique and application to Simulink is documented in Appendix A. This work has been carried out in collaboration with the EC MULTIFORM project. The Quasimodo work has focused on enabling the integration through a study of its Simulink semantics, simulator interfaces/APIs, developing and adapting the UPPAAL modeling pattern, and supporting adapter changes. The actual tool integration development work is mainly within MULTIFORM.

Co-simulation with Phaver/SpaceEX: Another alternative that we have explored is to integrate UPPAAL-TRON with a model-checker for hybrid systems based on hybrid

automata (where locations represent operating modes of the system and transitions switches between these modes, and where differential equations within locations describe evolution of continuous variables). The main idea is to combine the best of UPPAAL-TRON, namely the efficient handling of time and discrete signals in very large models, with the best of Phaver, namely the ability to describe and compute (linear) differential equations. A successful first integration with input signal generation has been done, and a more ambitious integration with the newer SpaceEX tool is being carried out that will also feature output monitoring and checking. However, these developments takes place within the EC MULTIFORM project and hence shall not be reported here, but is mentioned for completeness. The work is under submission for ICTSS'11 (International Conference on Testing of Communicating Systems).

2.3 Perspectives

We find the approach of using co-simulation with other tools for quite promising, enabling simultaneous evaluation more and other system aspects than discrete event behavior. Besides continuous behavior, also security and performance/efficiency aspects may require specialized notations and tools. Another example is the Poosl simulator of the ASML case reported in Deliverable 5.10.

3 Probabilistic Testing

A probabilistic model contains information that express with what probability the system executes a given transition; this may by an input given to the system, an output delivered by the system, or an internal computation step. Thus it may express information about distributions of both expected uses and expected responses.

In general such probabilistic information has several applications in model-based testing.

Operational Usage profile testing/statistical usage testing: Test input sequences are generated in correspondence with the distributions of the model such that they reflect the expected use of the system. This is the basis for performance evaluation and reliability estimation.

Guiding: Guiding either towards an area in the model that is of particular interest or is particularly critical, or to increase coverage of the model (utilizing information of likely outputs).

Statistical hypothesis testing: Here the goal is to estimate the probability of conformance.

An important common problem is how to draw statistically correct samples from the model (which may be fully stochastic and/or contain non-determinism).

3.1 Scheduler-based Probabilistic Testing with Profiles

This work has been recently submitted to SEFM 2011 [4].

Participants

- Hernán Baró Graf, Saarland University.
- Holger Hermanns, Saarland University.
- Jan Tretmans, Embedded Systems Institute, Radboud University Nijmegen.

3.1.1 Challenge

Model-based testing is one of the promising technologies to meet the challenges imposed on software testing. In model-based testing an implementation under test is tested for compliance with a model that describes the required behaviour of the implementation. A prominent model-based testing approach is rooted in the *ioco*-testing theory, where models are expressed as labelled transition systems, and compliance is defined with the ioco implementation relation [6]. This provides a sound foundation for labelled transition system testing, and has proved to be a practical basis for several model-based test generation tools and applications.

In the ioco approach to model-based testing, several steps in the test selection, generation and execution involve a non-deterministic selection among alternatives. In practice, these nondeterministic selections are routinely implemented by a non-biased probabilistic selection, that is, by uniform probabilistic choices among the alternatives. Uniform distribution is, for example, used in the ioco model-based testing tools TorX [1], JTorX [2], and TorXakis [5]. In this work, we investigate what happens to the theory of ioco testing, if one includes this probabilistic selection explicitly. To that end, we aim at an extension to the ioco theory by including probabilities in the test selection process. We strive for an orthogonal and expressive extension, that supports the possibility to guide the test selection using profiles derived empirically.

3.1.2 Results

After identifying the principal attack points in the test generation algorithm where nondeterministic selection occurs, we propose an expressive probabilistic test selection approach, in the form of randomised schedulers, that can pick probabilistically based on the full history observed so far. Several options and their interplay are discussed, and we in particular distinguish schedulers based on white-box and black-box histories. We then iterate the theory of ioco testing under these schedulers, and arrive at a probabilistic ioco theory.

Concretely, we link two possible ways to define schedulers. One of them directly works on the structures as they appear in the test selection process, and provides a probabilistic selection over the next actions possible after a certain trace has occurred. This is achieved by a *next action* scheduler (NA), and this is technically the directly matching interface to the testing process. To use this direct approach in practice however requires the test engineer to reason about technically involved objects, namely sets of states reached after executing some trace, where again the trace is driven by the random test selection. Because this is likely not very practical, we propose a second, indirect approach that is clean, elegant, and extends over so-called Markov chain usage model approaches (MCUM) [7, 3]. The latter prescribe the expected usage profile using Markov chains,

but do not consider ongoing interaction of input and output. We take a similar approach for the stimuli to the SUT, which we allow to be generated by a *Usage Profile* scheduler, which corresponds to a (not necessarily Markov) probabilistic usage model. The outputs of the implementation can, to some extent, be assumed to be governed by probabilities as well. This is not reflecting the usage model side, but the implementation side: it reflects assumptions over the likely behaviour in the specification, such as: *In 2 percent the account may not allow a withdrawal of money*. This opens the possibility to abstract data-dependent behaviour into probabilistic selections over the next output offered to the environment (*responses*). We therefore complement the up scheduler with a *Implementation Profile* scheduler (ip).

While this appears like a very useful and powerful approach to probabilistic testing, it raises the question how a pair of up- and ip-schedulers can be linked to the direct test selection with NA-schedulers. We show how a pair of ip- and up-schedulers induces a unique NA-scheduler. The construction of this NA-scheduler is not difficult. Still, in order to strongly argue that this scheduler is the natural object arising, we use embeddings of the resulting mathematical objects into partially observable Markov decision processes (POMDPs).

3.1.3 Perspective

The approach we introduce allows for guiding conformance test execution based on probabilistic profiles. The probabilities introduced by one or more schedulers can be rooted in different reasons and for different purposes, apart from usage and implementation profiles:

- Intended test coverage (higher/lower probability paths)
- Probabilities provided by the development leader who knows which parts may be more troubled or complex.
- Tuned probabilities based on the results of previous test executions.

We are looking into ways of providing a measure of coverage and reliability to the testing process based in the IPPO model. Also, we aim at interval estimation to validate an intended probabilistic choice directly from the specification, giving rise to a full probabilistic ioco.

References

[1] Belinfante, A., Feenstra, J., Vries, R.d., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heerink, L.: Formal Test Automation: A Simple Experiment. In *Int. Workshop on Testing of Communicating Systems 12*, 179–196, Kluwer (1999)

[2] Belinfante, A.: JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In *TACAS 2010*. LNCS 6015: 266–270, Springer (2010)

[3] Dulz, W., Zhen, F.: Matelo – Statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3. In *QSIC 2003*, 336-342, IEEE CS Press (2003)

[4] Baró Graf, H., Hermanns, H., Tretmans, J.: Scheduler-based Testing with Profiles. Submitted for publication.

[5] Mostowski, W., Poll, E., Schmaltz, J., Tretmans, J., Wichers Schreur, R.: Model-Based Testing of Electronic Passports. In *FMICS 2009*. LNCS 5825: 207–209, Springer (2009)

[6] Tretmans, J.: Model based testing with labelled transition systems. In *Formal Methods and Testing*. LNCS 4949:1–38, Springer (2008)

[7] Whittaker, J.A., Poore, J.H.: Markov analysis of software specifications. *ACM Trans. Softw. Eng. Methodol.* 2:93–106 (1993)

[8] Åström, K.J.: Optimal control of Markov processes with incomplete state information. Journal of Mathematical Analysis and Applications 10:174–205 (1965)

[9] Franck Cassez and Kim Guldstrand Larsen. The impressive power of stopwatches. In Catuscia Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2000.

[10] Goran Frehse, Kim G. Larsen, Marius Mikuionis, and Brian Nielsen. Testing and monitoring of hybrid systems. In *Proceedings of the* 1^{st} *International Conference on Runtime Verification*. Springer-Verlag, 2010, submitted.

[11] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In *Formal Approaches to Software Testing*, pages 79–94. 2005.

[12] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306, New York, NY, USA, 2005. ACM.

[13] Marius Mikucionis. *Online Testing of Real-Time Systems*. PhD thesis.

A Uppaal TRON to Matlab/Simulink Interface and Co-Simulation

Author: Marius Mikuionis, Kim G. Larsen, and Brian Nielsen, AAU

(The technical report of this appendix has also been sumbitted as part of EC Multiform Deliverable 3.2.2)

The paper shows how to connect simulation and testing tools in order to enhance environment emulation capabilities in online testing. UPPAAL TRON [11] is a successful online testing tool for real-time systems using timed automata model-checking engine. Matlab/Simulink is a popular tool for simulating non-linear dynamical systems. We propose to augment the timed automata models with dynamical behavior co-simulation by Simulink for environment emulation purposes. The resulting framework can also be used to test conformance of Simulink models against timed automata specification.

A.1 Motivation

Simple embedded systems consist of a controller and its plant under control. The most of requirements for controller software can be described by timed automata models, however the plant often consists of physical processes which are best modeled by differential equations or similar dynamical models.

Fig. 3 shows a setup for testing Danfoss cooling controller [12], where the controller is connected to UPPAAL TRON. The test specification is a timed automata model consisting of processes modeling assumptions about environment and requirements for the controller. In this particular case the environment consists of a room temperature, display and relay actuators. The requirements consist of temperature calculation converting temperature sensor signal into meaningful temperature values, low and high temperature alarms and compressor depending on temperature values, and finally fan and defrost cycle. The specification also defines the input/output interface between the environment and the controller: the temperature sensor values are inputs, calculated temperature value and relay activations are outputs.

The goal of this paper is to show how continuous input trajectories can be generated by an external simulation tool like Simulink during the online test. In order to achieve the goal the following problems need to be solved:

1. Exchange UPPAAL integer variable values with real-valued signals from Simulink in a sound way.
2. Synchronize discrete timed automata events with continuous signals.
3. Time synchronization between the tools.

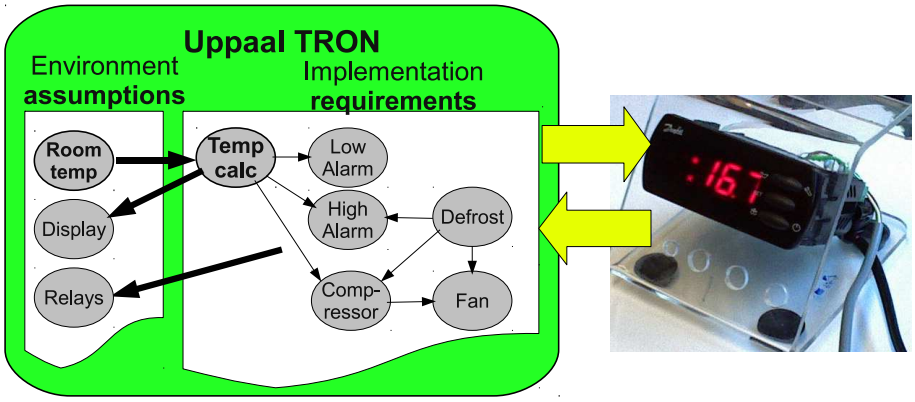

Figure 3: Online test setup for Danfoss EKC cooling controller.

A.2 Method

The first problem of exchanging variable values is solved by a modeling pattern and UPPAAL TRON extension. The continuity of signals is then solved by Simulink integration components (S-Function library). Finally the time synchronization is performed either by using TRON's Virtual Time framework or host's clock.

Fig. 4 shows the test setup where timed automata environment emulation is complemented by Simulink trajectories. The connection of tools is achieved through UPPAAL TRON test adapter and Simulink S-Function interfaces.

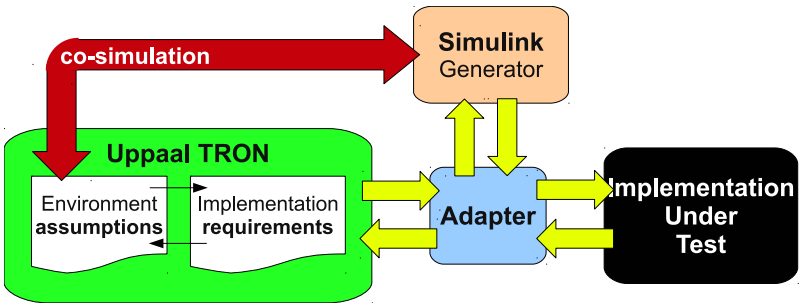

Figure 4: Complementing environment emulation with Simulink trajectories.

The soundness of timed automata co-simulation with Simulink trajectories is ensured through abstraction: the environment model has is an abstract representation of the generated trajectory.

A.2.1 Modeling Pattern

In online testing, the role of the tester is to emulate the environment model and check that implementation conforms to the model of requirements. UPPAAL TRON assumes that the system model is closed, i.e. the system consists of only an environment and an implementation and that there is no third party perturbing the test. Hence whatever input trajectory is being generated, it has to be generated in the environment model too.

We propose a systematic way of creating sound UPPAAL timed automata abstractions of continuous trajectories by encoding real-valued signals in two integer bounds.

Fig. 5 shows an abstract temperature calculation, where the concrete temperature rises from an old to a new value and its abstract value is represented by an expanded interval which is collapsed when the temperature converges. The corresponding temperature ab-

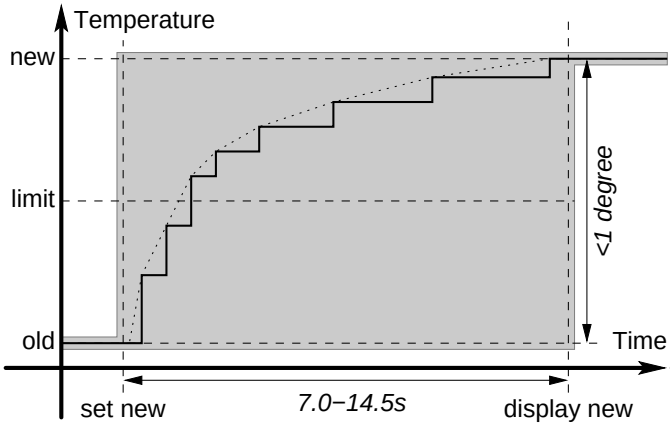

Figure 5: Abstract temperature calculation.

straction is modeled in the UPPAAL timed automaton in Fig. 6 by `IUTCalcTemp` variable which is a structure consisting of `L` and `H` integer members denoting the lower and higher bound respectively. Initially, the automaton waits for new values from temperature sensors in `IUTTemp` variable signaled by observable `temp_r` channel. In location `Widened` the calculated temperature interval is widened denoting larger possibilities for calculated temperature value. Then the rest of the model the model is notified by `tempChange` at any and all times between 0 and 145 time units (the transition is not observable). Later in location `Narrowed` the interval is collapsed to a more definite temperature and the rest of the model is notified by `tempChange`. The automaton is made input-enabled by additional edges with `temp_r` synchronizations. Fig. 7 shows a low temperature alarm requirement which is sensitive to calculated temperate range in `IUTCalcTemp`, e.g. in location `Off` automaton has two options: to stay in location `Off` when higher bound is greater or equal than `LowTempLimit` by executing a loop or move to location `Triggered` when the lower bound is less than `LowTempLimit`.

Similarly, Fig. 8 shows the environment model generating ranges of possible temperature values in `ENVTemp`: in location `Dec` the range values are decreased and in location `Inc` the range values are increased and passed to adapter as inputs. The change of temperature is changed by smoothly widening the interval: only one bound is changed while the other uses the displayed temperature bound. The generated pair of low and high bounds can then be interpreted by the test adapter as suggestion to generation any value from that range. At this point the test adapter may consult Simulink to generate a concrete value or trajectory from the range. The generated concrete value may be approximated by a closest integer pair and fed back to tester, where UPPAAL TRON would assign more concrete values to the temperate bounds resulting in more concrete emulation.

A.2.2 Test Adapter API

This section describes the variable value exchange in adapter protocol in more details. Consider a situation shown in Fig. 9, where the lower and higher bounds are encoded

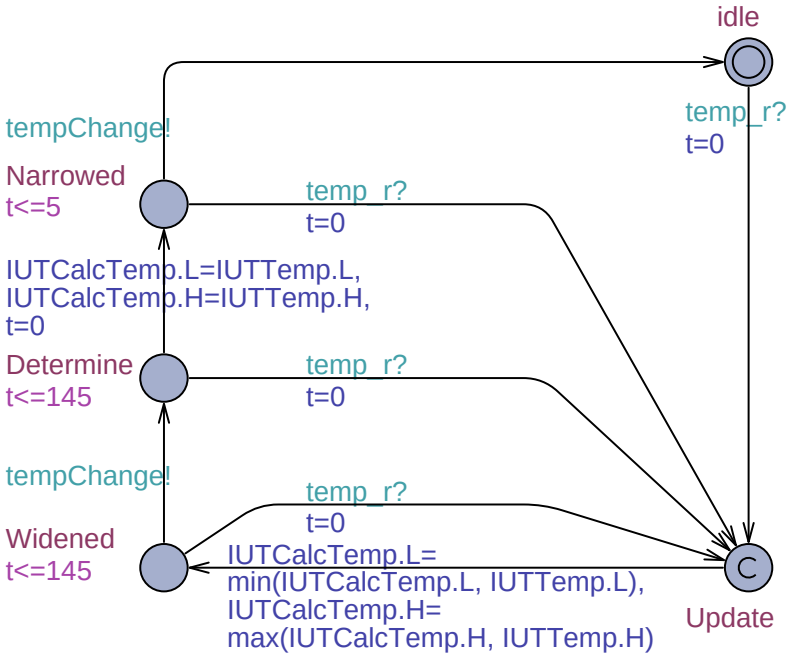

Figure 6: Abstract model for temperature calculation.

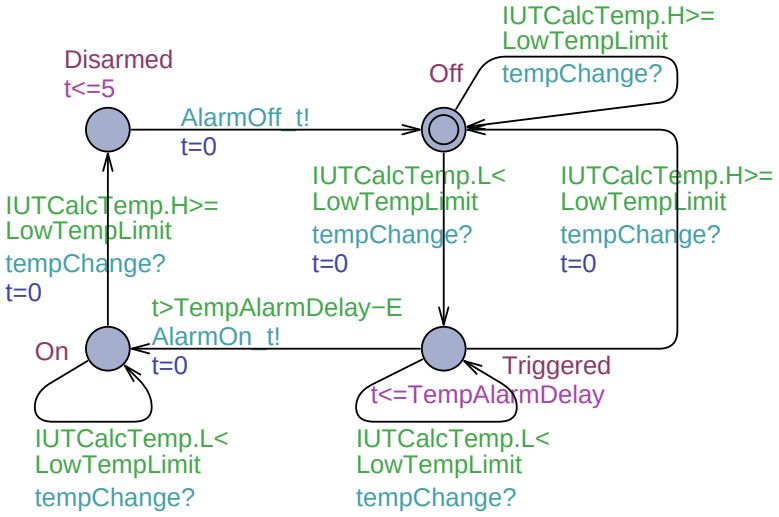

Figure 7: Low temperature alarm depending on temperature bounds.

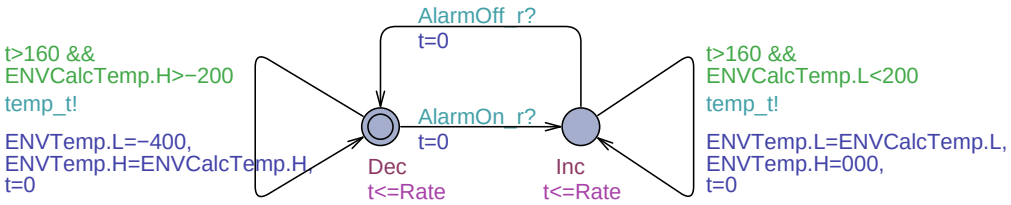

Figure 8: Abstract temperature generation.

by variables `L` and `H` respectively. Suppose, initially the bounds are 5 and 10 which are sent as input values to a test adapter. The adapter interprets the values as interval

$[5; 10]$, generates a concrete input with value 7.5 and returns approximating interval $[7; 8]$ as variable values 7 and 8. TRON then checks that the delivered interval is within the model bounds and updates the bounds accordingly leading to a more concrete emulation of environment than it is specified in the model.

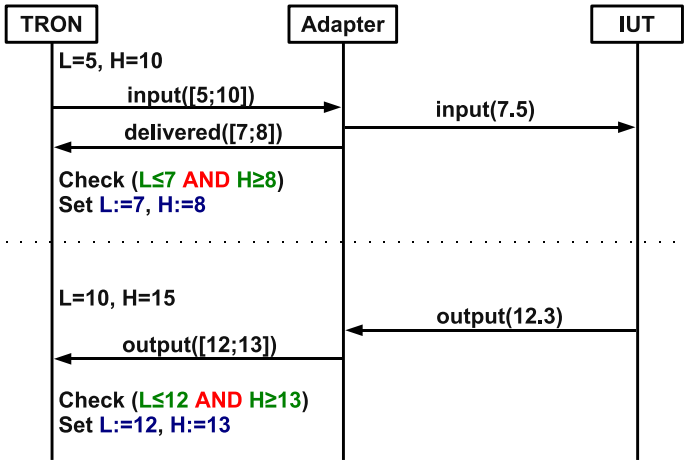

Figure 9: Message sequence chart of input and output signal translation.

The same way the outputs from IUT can be checked: suppose, model dictates that the bound variables contain 10 and 15, the adapter receives value 12.3, approximates the value with interval $[12; 13]$ and passes to tester, then TRON checks that the output range is within modeled bounds and updates the bound variables.

Note that the variable update should always be within modeled bounds, thus the tested behavior is always a subset of the abstract modeled behavior which is consistent with rtioco conformance relation.

The test adapter API is extended with a concept of bounds along the old concrete variable values. Listing 1 shows the extended reporter interface used to configure input and output channels together with variables. The lines 14-17 are used to configure bounds to input and output actions on specified channels.

The bounds are then checked and updated during the test execution by the algorithm in Listing 2. The algorithm computes a conjunction of the abstract model region and more concrete region built from adapter values.

Note that it is possible that the region from adapter is not completely included in the model's region, meaning that the signal may potentially be out of modeled range due to limited precision in measurement or over-approximation. In such cases the testing continues on the basis that there is no conclusive evidence that the IUT behavior is outside the specification, but such deviation is a good candidate for diagnostic check later if the test eventually fails.

A.2.3 Simulink Integration

Fig. 10 shows the architecture of TRON integration into Simulink:

- The upper layer shows that TRON appears as a S-Function component which can be connected to other Simulink components.

Listing 1: Extended UPPAAL TRON test adapter API.

```
struct Reporter
{
    void (*report_now)(Reporter*,
                       int32_t chan, uint16_t n,
                       const int32_t data[]);
    int32_t (*getInputEncoding)(Reporter*,
                                const char* inputChanName);
    int32_t (*getOutputEncoding)(Reporter*,
                                const char* outputChanName);
    int32_t (*addVarToInput)(Reporter*, int32_t chan,
                       const char* variable);
    int32_t (*addVarToOutput)(Reporter*, int32_t chan,
                       const char* variable);
    int32_t (*addBoundsToInput)(Reporter*, int32_t chan,
                       const char* low, const char* upp);
    int32_t (*addBoundsToOutput)(Reporter*, int32_t chan,
                       const char* low, const char* upp);
    int32_t (*setTimeUnit)(Reporter*,
                       const int64_t& microsecs_per_unit);
    int32_t (*setTimeout)(Reporter*,
                       int32_t timeout_in_units);
    const char* (*getErrorMessage)(Reporter*,
                       int32_t error_code);
};
```

- The layer below stands for Matlab/Simulink machinery simulating the whole model.
- The execution of `TronSFun` component requires loading the TRON adapter library `TronSFun.mexglx` which translates Simulink input/output to TRON events and back.
- Subsequently TRON is loaded as a dynamically linked library `libtron.so` which reads the model, interacts with the test adapter and produces test logs.

Discrete inputs from TRON are converted to continuous input to Simulink in the following way:

1. TRON offers input action with values and bounds attached.
2. The test adapter updates and stores a local copy of the values and the bounds.
3. The test adapter provides the copied values whenever is requested by Simulink.
4. Simulink uses the values to feed to further components and generate trajectories.

Continuous outputs from Simulink are converted into discrete UPPAAL events by the following way:

1. The test adapter monitors the output signal from Simulink.
2. The test adapter reports and output event (potentially with values and ranges) whenever output signal reaches or crosses some discrete bound.

Listing 2: Extended data check for variable bounds.

```
bool satisfies(SymbolicState state,
               VarBound lower, VarBound upper,
               int32_t valueLower, int32_t valueUpper)
{
  int32_t modelLower = max(lower.getValue(state), valueLower);
  int32_t modelUpper = min(upper.getValue(state), valueUpper);
  if (modelLower <= modelUpper) { // non-empty region
    lower.setValue(state, modelLower);
    upper.setValue(state, modelUpper);
    return true;
  } else return false;
}
```

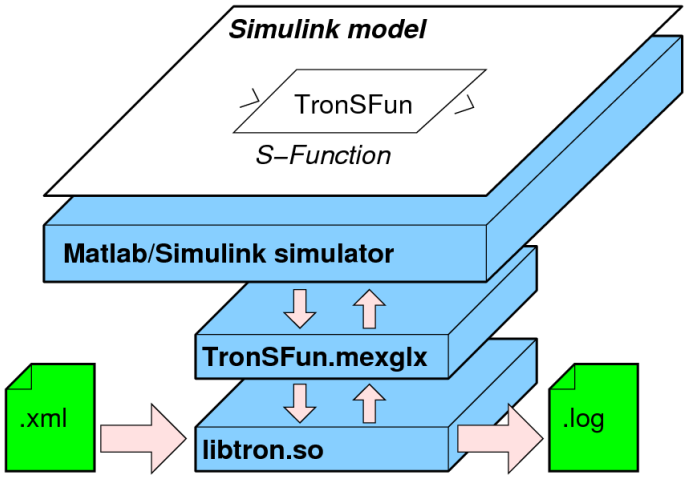

Figure 10: Abstraction of temperature calculation.

3. TRON checks whether the observed output (and values with ranges) conforms to the model behavior.

A.2.4 Timing

We need to synchronize the time between the tools at runtime since both tools participate in online test execution at the same time and any delay may affect the outcome. The time synchronization is achieved through Virtual Time framework implemented in UPPAAL TRON.

Briefly, the Virtual Time framework consists of global clock (GC) object, where all time related and thread creation systems calls are routed to, the threads perform computations and eventually ask the system to sleep or wait for some event, GC is advanced when all threads agree to wait and GC wakes or notifies the earliest thread.

In a fixed time-step simulation, Simulink is represented by a single thread loop which computes and routes the input/output signals between its components and then requests to sleep for one time-step. Similarly TRON uses one thread to do its computations and request for varying time delays which usually are at least an order of magnitude longer than one time-step. The Virtual Time framework ensures that global clock is advanced

efficiently, i.e. no thread is woken until it needs to and the clock is advanced at the shortest requested pace at once. Consequently TRON's performance is not affected by tiny steps of Simulink simulation and Simulink seamlessly read variable snapshots.

The Virtual Time framework requires minimal adjustments and the global clock can be switched to a host's clock via command line without a need for recompilation.

A.3 Example

Fig. 11 shows an overview of Simulink model where `Tester` is connected in the loop together with generator component (`tempGen`) and IUT (`tempMonitor` and `lowTempAlarm`).

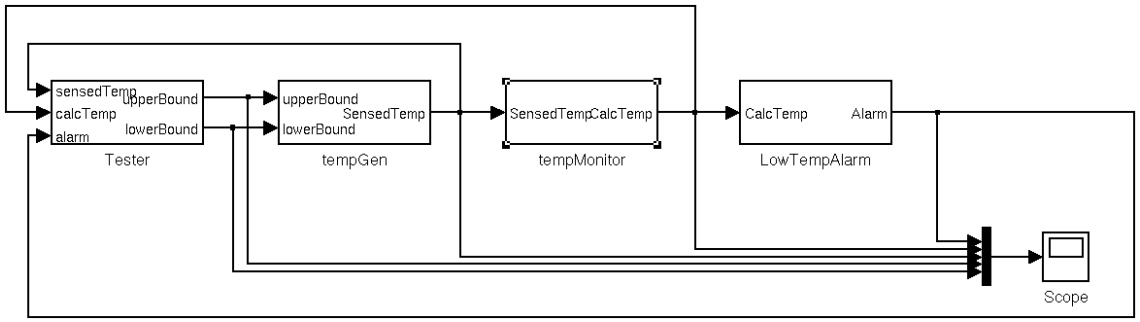

Figure 11: Simulink model of overall system.

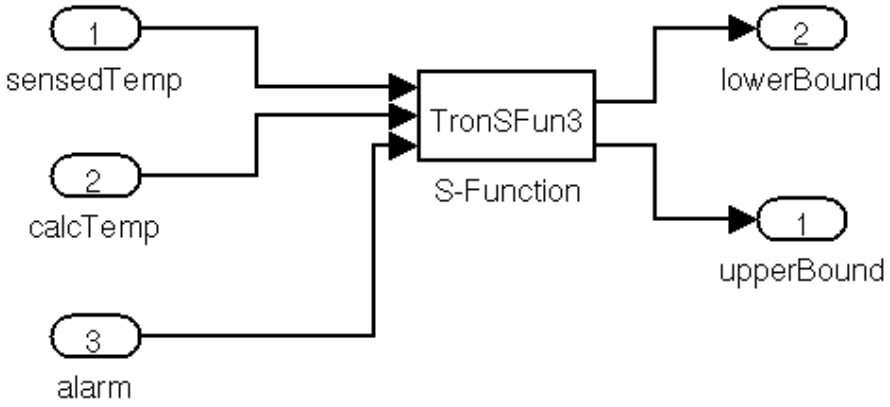

Figure 12: The details of tester component showing TronSFun.

A.4 Results and Discussion

The paper provides modeling abstraction paradigm for testing using UPPAAL timed automata in conjunction with Simulink models. The two models are synchronized with data variables and time.

The connection is implemented as test adapter, thus Simulink may equally act as an implementation under test.

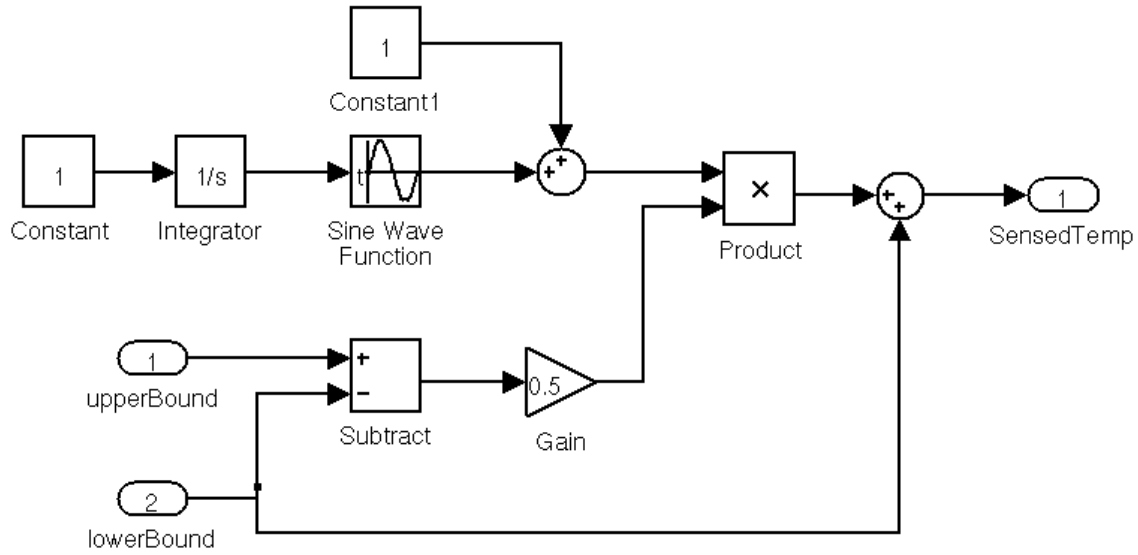

Figure 13: The details of temperature generator which generates a sine curve between lower and upper bounds.

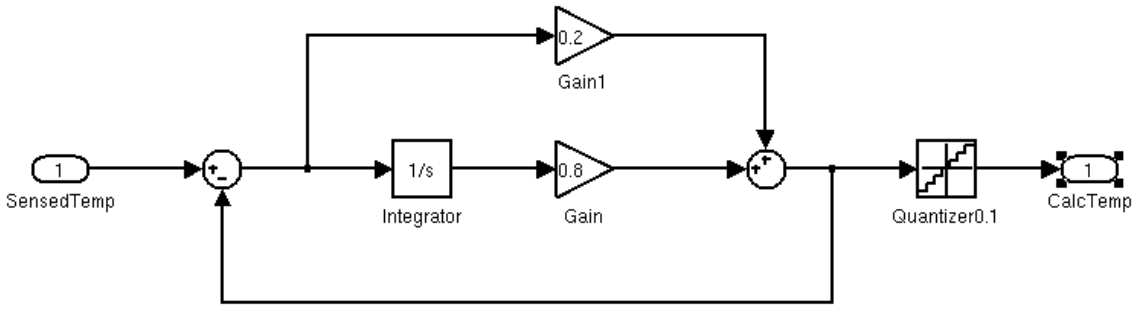

Figure 14: The details of temperature monitor which calculates the displayed temperature.

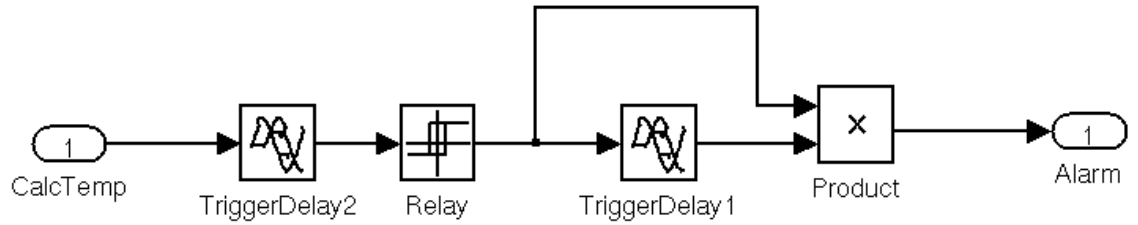

Figure 15: The details of temperature monitor which calculates the displayed temperature.

In the future it would be interesting to connect UPPAAL TRON, Simulink, `PHAVer` and implementation under test to achieve a complete testing framework of hybrid systems.