Testing Real-Time Embedded Software using UPPAAL-TRON

An Industrial Case Study

Kim G. Larsen Marius Mikucionis Brian Nielsen Arne Skou

Center of Embedded Software Systems, CISS
Aalborg University
Fredrik Bajersvej 7B, DK-9220 Aalborg, Denmark

Email: { kgl | marius | bnielsen | ask}@cs.aau.dk

ABSTRACT

UPPAAL-TRON is a new tool for model based online black-box conformance testing of real-time embedded systems specified as timed automata. In this paper we present our experiences in applying our tool and technique on an industrial case study. We conclude that the tool and technique is applicable to practical systems, and that it has promising error detection potential and execution performance.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Miscellaneous; D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution, monitors, testing tools*

General Terms

Algorithms, Experimentation, Languages, Theory, Verification

Keywords

Black-box testing, online testing, embedded systems, control software, real-time systems

1. INTRODUCTION

Model-based testing is a promising approach for improving the testing of embedded systems. Given an abstract formalized model (ideally developed as the design process) of the behaviour of aspects of the implementation under test (IUT), a test generation tool automatically explores the state-space of the model to generate valid and interesting test cases that can be executed against the IUT. The model specifies the required and allowed behavior of the IUT.

UPPAAL is a mature integrated tool environment for modeling, verification, simulation, and testing of real-time systems modeled as networks of timed automata [8]. UPPAAL-TRON (TRON for short) is a recent addition to the UPPAAL environment. It performs model-based black-box conformance testing of the real-time constraints of embedded systems. TRON is an *online* testing tool which means that it, at the same time, both generates and executes tests event-by-event in real-time. TRON represents a novel approach to testing real-time systems, and is based on recent advances in the analysis of timed automata. Applying TRON on small examples has shown promising error detection capability and performance.

In this paper we present our experiences in applying TRON on an industrial case study. Danfoss is a Danish company known worldwide for leadership in Refrigeration & Air Conditioning, Heating & Water and Motion Controls [2]. The IUT, EKC 201/301, is an advanced electronic thermostat regulator sold world-wide in high volume. The goal of the case study is to evaluate the feasibility of our technique on a practical example.

TRON replaces the environment of the IUT. It performs two logical functions, stimulation and monitoring. Based on the timed sequence of input and output actions performed so far, it stimulates the IUT with input that is deemed relevant by the model. At the same time it monitors the outputs and checks the conformance of these against the behavior specified in the model. Thus, TRON implements a closed-loop testing system.

To perform these functions TRON computes the set of states that the model can possibly occupy after the timed trace observed so far. Thus, central to our approach is the idea of symbolically computing the current possible set of states. For timed automata this was first proposed by Tripakis [16] in the context of failure diagnosis. Later that work has been extended by Krichen and Tripakis [9] to online testing from timed automata. The monitoring aspect of this work has been applied to NASA's Mars Rover Controller where existing traces are checked for conformance against given execution plans translated into timed automata [15]. In contrast, the work presented in this paper performs real-time online black-box testing (both real-time stimulation and conformance checking) for a real industrial embedded device consisting of hardware and software. Online testing based on timed CSP specifications has been proposed and applied in practice by Peleska [14].

Our approach, previously presented in [5, 13, 11]; an abstract appeared in [12]), uses the mature UPPAAL language and model-checking engine to perform *relativized timed input/output conformance* testing, meaning that we take environment assumptions explicitly into account.

Compared to current control engineering testing practices, our

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

emphasis is on testing discrete mode switches (possibly non-deterministic) and on real physical time constraints (deadlines) on observable input/output actions, and less on continuous state evolution characterized by differential equations. Also many engineering based approaches has no general formal correctness criteria, and correctness is assessed manually (tool assisted) by correlating a simulated-model with observed test data. In our case we have an explicit correctness relation allowing us to automatically map events and timings into model and assign verdicts to the observed behavior online. It is also important to remark that our environment models need not represent a single deterministic scenario, but represents all relevant environment behaviors/assumptions from which samples are randomly chosen during test execution. On the other hand, the strong focus and dependency on environment models common in control engineering testing appear new to formal software testing.

In Section 2 we introduce the concepts behind our testing framework. Section 3 describes the case, Section 4 our modeling experiences, and Section 5 performance results. Section 6 concludes the paper.

2. TESTING FRAMEWORK

The most important ingredients in our framework is relativized conformance, timed automata, environment modeling, and the test generation algorithm.

2.1 Relativized Conformance Testing

An embedded system interacts closely with its environment which typically consists of the controlled physical equipment (the plant) accessible via sensors and actuators, other computer based systems or digital devices accessible via communication networks using dedicated protocols, and human users. A major development task is to ensure that an embedded system works correctly in its real operating environment.

The goal of (relativized) conformance testing is to check whether the behavior of the IUT is correct according to its specification under assumptions about the behavior of the actual environment in which it is supposed to work. In general, only the correctness in this environment needs to be established, or it may be too costly or ineffective to achieve for the most general environment. Explicit environment models have many other practical applications.

Figure 1 shows the test setup. The test specification is a network of timed automata partitioned into a model of the environment of the IUT and the IUT. TRON replaces the environment of the IUT, and based on the timed sequence of input and output actions performed so far, it stimulates the IUT with input that is deemed relevant by the environment part of the model. Also in real-time it checks the conformance of the produced timed input output sequence against the IUT part of the model. We assume that the IUT is a black-box whose state is not directly observable. Only input/output actions are observable. The adapter is an IUT specific hardware/software component that connects TRON to the IUT. It is responsible for translating abstract input test events into physical stimuli and physical IUT output observations into abstract model outputs. It is important to note that we currently assume that inputs and outputs are discrete (or discretized) actions, and not continuously evolving.

Depending on the construction of the adapter, TRON can be connected to the hardware (possibly via sensors and actuators) with embedded software forming hardware-in-the-loop testing, or it can be connected directly to the software forming software-in-the-loop testing.

We extended the input/output conformance relation **ioco** [17] between a formal specification and its black-box implementation to the timed setting and relatively to a given environment.

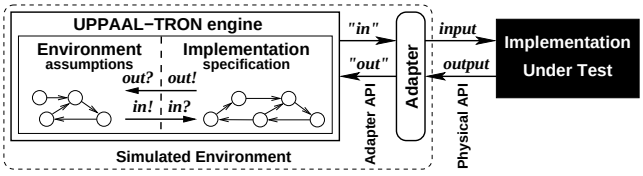


Figure 1: TRON test setup.

Intuitively $i\ \mathbf{rtioco}_e\ s$ means that after executing any timed input/output trace σ that is possible in the composition of the system specification s and environment specification e, the implementation i in environment e may only produce outputs and timed delays which are included in the specification s under environment e. Relativized timed input/output conformance **rtioco** [6] is defined formally in Equation 1.

$$i\ \mathbf{rtioco}_e\ s = \forall \sigma \in \mathsf{TTr}(s,e).\ \mathbf{out}\big((i,e)\ \mathbf{after}\ \sigma\big) \subseteq \mathbf{out}\big((s,e)\ \mathbf{after}\ \sigma\big) \quad (1)$$

Here $\mathbf{after}\ \sigma$ denotes the set of states the specification system (s,e) (resp. implementation system (i,e)) may possibly occupy after executing the timed i/o trace σ. $\mathbf{out}()$ denotes the possible outputs (including permissible delays) the system can produce from a given set of states.

The output inclusion in the relation guarantees both functional and time-wise correctness. The IUT is not allowed produce any output actions (including the special output of letting time pass and not producing outputs in time) at a time they could not be done by the specification.

2.2 Timed Automata

We assume that a formal specification can be modeled as a network of timed automata. We explain timed automaton by example, and refer to [1] for formal syntax and semantics. A timed automaton is essentially a finite state machine with input/output actions (distinguished respectively by ? and !) augmented with discrete variables and a set of special real-valued clock variables which models the time.

Clocks and discrete variables may be used in predicates on transitions (called guards) to define when the transitions may take place. A location invariant is a clock predicate on an automaton location that defines for how long the automaton is allowed to stay in that location, thus forcing the automaton to make progress within the specified time bounds. On transitions, the variables can be assigned a value, and clocks may be reset to zero.

Figure 2(a) shows an UPPAAL automaton of a simple cooling controller $\mathcal{C}^r$ where x is real-valued clock and r is an integer constant. Its goal is to control and keep the room temperature in *Med* range. The controller is required: 1) to turn *On* the cooling device within an allowed reaction time r when the room temperature reaches *High* range, and 2) to turn it *Off* within r when the temperature drops to *Low* range.

In the encircled initial location *off*, it forever awaits temperature input samples *Low*, *Med* and *High*. When $\mathcal{C}^r$ receives *High* it resets the clock x to zero and moves to location *up*, where the location invariant $x \leq r$ allows it to remain for at most r time units. Edges may also have guards which define when the transition is enabled (see e.g. in Figure 2(d)). At latest when x reaches r time units the output *on* is generated. If a *Low* is received in the mean time

it must go back *off*. Transitions are taken instantaneously and time only elapses in locations.

In location *off* the automaton reacts non-deterministically to input *Med*: $\mathcal{C}^r$ may choose either to take a loop transition and stay in location *off* or move to location *up*. When $\mathcal{C}^r$ is used as a specification a relativized input/output conforming controller implementation may choose to perform either. Thus non-determinism gives the implementation some freedom. There are two sources of non-determinism in timed automata: 1) in the timing (tolerances) of actions as allowed by location invariants and guards, and 2) in the possible state after an action.

Timed automata may be composed in parallel, communicate via shared variables and synchronize rendezvous-style on matching input/output transitions. In a *closed* timed automata network all output action transitions have a corresponding input action transition.

UPPAAL is an model checker for real-time systems, and supports timed automata networks with additional integer variable types, broadcast (one-to-many) synchronizations and other extensions. UPPAAL provides an efficient set of symbolic model-checking algorithms for performing symbolic reachability analysis of timed automata. Since clock values are real-valued, the state-space of the model is infinite, and cannot be represented and computed explicitly. A symbolic state represents a (potentially infinite) set of concrete states and is implemented as particular set of linear inequations on clock variables. Thus the evaluation of guards and computation of successor symbolic states is done symbolically.

2.3 Environment Modeling

In this section we exemplify how our conformance relation discriminates systems, and illustrate the potential power of environment assumptions and how this can help to increase the relevance of the generated tests for a given environment.

Consider the simple cooling controller of Figure 2(a) and the environment in Figure 2(c). Take $\mathcal{C}^6$ to be the specification and assume that the implementation behaves like $\mathcal{C}^8$. Clearly, $\mathcal{C}^8$ ~~ioco~~$_{\mathcal{E}_M}$ $\mathcal{C}^6$ because the timed trace $0 \cdot Med! \cdot 7 \cdot On!$ is possible in the implementation, but not in the specification. Formally, $\mathbf{out}\big(\mathcal{C}^8 \textbf{ after } 0 \cdot Med! \cdot 7\big) = \{On!\} \cup \mathbb{R}_{\geq 0} \not\subseteq \mathbf{out}\big(\mathcal{C}^6 \textbf{ after } 0 \cdot Med! \cdot 7\big) = \mathbb{R}_{\geq 0}$ (recall that C^r may remain in location *off* on input *Med* and not produce any output). The implementation can thus perform an output at a time not allowed by the specification.

Next, suppose $\mathcal{C}^r$ is implemented by a timed automaton $\mathcal{C}'^r$ equal to $\mathcal{C}^r$, except the transition $up \xrightarrow{Low} dn$ is missing, and replaced by a self loop in location up. They are distinguishable by the timed trace $0 \cdot$*Med?*$\cdot 0 \cdot$*High?*$\cdot 0 \cdot$ *Low?*$\cdot 0 \cdot$ *On!* in the implementation that is not in the specification (switches the compressor *Off* instead).

Figures 2(b) to 2(e) show four possible environment assumptions for $\mathcal{C}^r$. Figure 2(c) shows the universal and completely unconstrained environment $\mathcal{E}_M$ where room temperature may change unconstrained and may change (discretely) with any rate. This is the most discriminating environment that can generate any input output sequence and thus (in principle) detect all errors.

This may not be realistic in the given physical environment, and there may be less need to test the controller in such an environment, as temperature normally evolves slowly and continuously, e.g., it cannot change drastically from *Low* to *High* and back unless through *Med*. Similarly, most embedded and real-time systems also interact with physical environments and other digital systems that— depending on circumstances—can be assumed to be correct and correctly communicate using well defined interfaces and protocols. The other extreme in Figure 2(c) is the least discriminating environment; it merely passively consumes output actions.

Figure 2(d) shows the environment model $\mathcal{E}_1^d$ where the temperature changes through *Med* range and with a speed bounded by d. Figure 2(e) shows an even more constrained environment $\mathcal{E}_2$ that assumes that the cooling device works, e.g., temperature never increases when cooling is on. Notice that $\mathcal{E}_2$ and $\mathcal{E}_1$ have less discriminating power and thus may not reveal faults found under more discriminating environments. However, if the erroneous behavior is impossible in the actual operating environment the error may be irrelevant. Consider again the implementation $\mathcal{C}'^r$ from above. This error can be detected under $\mathcal{E}_0$ and $\mathcal{E}_1^{3d<r}$ via the timed trace that respects the environments $d \cdot$*Med?*$\cdot d \cdot$*High?*$\cdot d \cdot$*Med?*$\cdot d \cdot$*Low?*$\cdot \varepsilon \cdot$ *On!*, $\varepsilon \leq r$. The specification would produce *Off*. The error cannot be detected under $\mathcal{E}_1$ if it too slow $3d > r$, and never under $\mathcal{E}_2$ for no value of d.

In the extreme the environment behavior can be so restricted that it only reflects a single test scenario that should be tested. In our view, the environment assumptions should be specified explicitly and separately.

2.4 Online Testing Algorithm.

Here we outline the algorithm behind TRON informally. The precise formal definitions and algorithms behind TRON have been documented in [12, 11, 6, 7] and we refer to these for further details.

The environment model functions as a (state-dependent) input-stimuli (load) generator. The IUT-model functions as as a test oracle, and is used to evaluate the correctness of the observed timed input output sequence. In order to simulate the environment and monitor the implementation, Algorithm 1 maintains the current reachable symbolic state set $\mathcal{Z} \subseteq S \times E$ that the test specification can possibly occupy after the timed trace observed so far.

Based on this symbolic state-set, TRON checks whether the observed output actions and timed delays are permitted in the specification. In addition TRON computes the set of possible inputs that may be offered to the implementation.

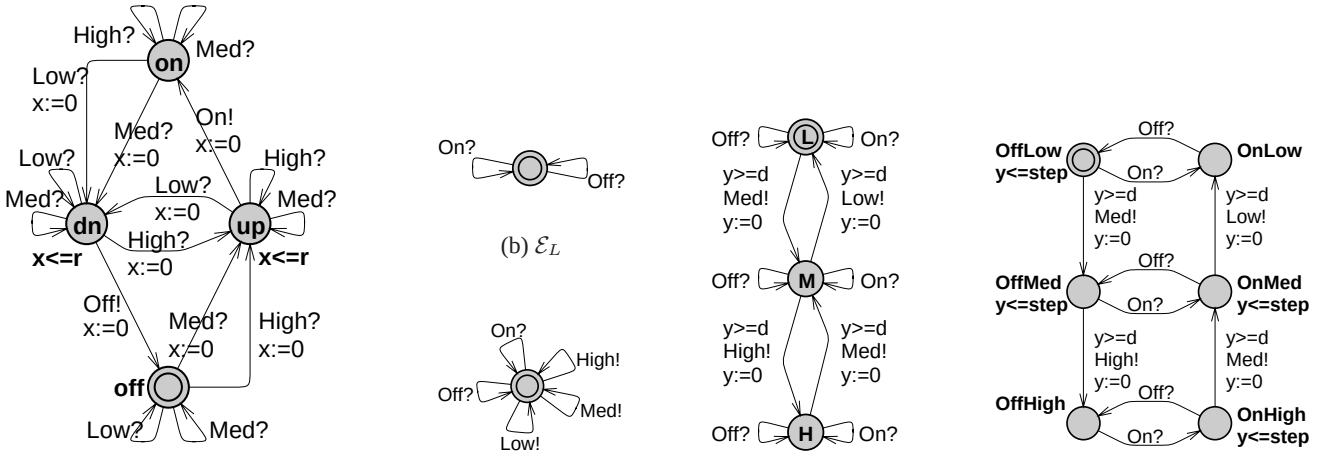
ALG. 1. *Test generation and execution:* $\mathcal{Z} := \{(s_0, e_0)\}$.

while $\mathcal{Z} \neq \emptyset \wedge \sharp iterations \leq T$ ***do*** *choose randomly:*
- offer input action*:*
  - ***if*** $\mathsf{EnvOutput}(\mathcal{Z}) \neq \emptyset$
    - *randomly choose* $i \in \mathsf{EnvOutput}(\mathcal{Z})$
    - *send* i *to IUT,* $\mathcal{Z} := \mathcal{Z}$ **after** i
- delay and wait for an output*:*
  - *randomly choose* $d \in \mathsf{Delays}(\mathcal{Z})$
  - *sleep* d *or wake up on output* o *at* $d' \leq d$
  - ***if*** o *occurs* ***then***
    - $\mathcal{Z} := \mathcal{Z}$ **after** d'
    - ***if*** $o \notin \mathsf{ImpOutput}(\mathcal{Z})$ ***then return*** fail
    - ***else*** $\mathcal{Z} := \mathcal{Z}$ **after** o
  - ***else*** $\mathcal{Z} := \mathcal{Z}$ **after** d
- reset and restart*:* $\mathcal{Z} := \{(s_0, e_0)\}$, ***reset*** *IUT*

if $\mathcal{Z} = \emptyset$ ***then return*** fail ***else return*** pass

TRON randomly chooses between one of three basic actions: either send a randomly selected relevant input to the IUT, letting time pass by some (random) amount and silently observe the IUT for outputs, or reset the IUT and restart. The set of input actions that are possible in the current state-set $\mathcal{Z}$ (enabled environment output) is denoted by $\mathsf{EnvOutput}(\mathcal{Z})$. Similarly, $\mathsf{ImpOutput}(\mathcal{Z})$ denotes the allowed set of implementation outputs, and $\mathsf{Delays}(\mathcal{Z})$ the possible delays before the tester must give an input to the IUT (as constrained by invariants the environment model). In the practical implementation the probability of restarting is chosen comparatively very low.

If the tester observes an output or a time delay it checks whether



(a) $\mathcal{C}^r$: simple cooling controller.

(b) $\mathcal{E}_L$

(c) $\mathcal{E}_M$.

(d) $\mathcal{E}_1^d$.

(e) $\mathcal{E}_2^{d,s}$.

Figure 2: Timed automata of simple controller and various environments.

this is legal according to the state set. The state set is updated whenever an input is offered, an output or a delay is observed. Consider the system $(\mathcal{C}^r, \mathcal{E}_1^d)$. The initial state-set is the single symbolic state: $\{\langle off, L, x = 0 \wedge y = 0\rangle\}$. After a delay of d or more, $\{Med\}$ is the set of possible inputs. Suppose that TRON issues Med after $\delta \geq d$ time units. The state-set now consists of two states: $\{\langle off, M, x = \delta \wedge y = 0\rangle, \langle up, M, x = 0 \wedge y = 0\rangle\}$. If *On* is received later at time $\delta' \leq r$ the first element in the state-set will be eliminated resulting in $\{\langle on, M, x = 0 \wedge y = \delta'\rangle\}$. Illegal occurrence or absence of an output is detected if the state set becomes empty which is the result if the observed trace is not in the specification.

TRON uses using the UPPAAL engine to traverse internal, delay and observed action transitions, to evaluate clock and variable guards, and to perform variable assignments. We use the efficient reachability algorithm implementation [3] to implement the operator **after** . It operates on bounded symbolic states, checks for symbolic-state inclusions and thus always terminates even if the model contains loops of internal actions. Further information about the implementation of the required symbolic operations can be found in [6].

Currently TRON is available to download via the Internet free of charge for evaluation, research, education and other non-commercial purposes [10]. TRON supports all UPPAAL modeling features including non-determinism, provides timed traces as test log and a verdict as the answer to **rtioco** relation, and features for model-coverage measurements.

3. THE DANFOSS EKC-201 REFRIGERATION CONTROLLER

We applied UPPAAL-TRON on a first industrial case study provided by Danfoss Refrigeration Controls Division. The EKC controls and monitors the temperature of industrial cooling plants such as cooling and freezer rooms and large supermarket refrigerators.

3.1 Control Objective

The main control objective is to keep the refrigerator room air temperature at a user defined set-point by switching a compressor on and off. It monitors the actual room temperature, and sounds an alarm if the temperature is too high (or too low) for too long a period. In addition it offers a myriad of features (e.g. defrosting and safety modes in case of sensor errors) and approximately 40 configurable parameters.

The EKC obtains input from a room air temperature sensor, a defrost temperature sensor, and a two-button keypad that controls approximately 40 user configurable parameters. It delivers output via a compressor relay, a defrost relay, an alarm relay, a fan relay, and a LED display unit showing the currently calculated room air temperature as well as indicators for alarm, error and operating mode.

Figure 3 shows a simplified view of control objective, namely to keep the temperature within *setPoint* and *setPoint+differential* degrees. The regulation is to be based on an weighted averaged room temperature T_n calculated by the EKC by periodically sampling (around 1.2 sec.) the air temperature sensor such that a new sample T is weighted by 20% and the old average T_{n-1} by 80%:

$$T_n = \frac{T_{n-1} * 4 + T}{5} \tag{2}$$

A certain minimum duration must pass between restarts of the compressor, and similarly the compressor must remain on for a minimum duration. An alarm must sound if the temperature increases (decreases) above (below) *highAlarmLimit* (*lowAlarmLimit*) for *alarmDelay* time units. All time constants in the EKC specification are in the order of seconds to minutes, and a few even in hours.

3.2 Test Adaptation.

A few comments are necessary about the test adapter for the EKC since it determines what and how precise the IUT can be controlled and observed.

Internally, the EKC is organized such that nearly every input, output and important system parameter is stored in a so-called parameter database in the EKC that contains the value, type and permitted range of each variable. The parameter database can be in-

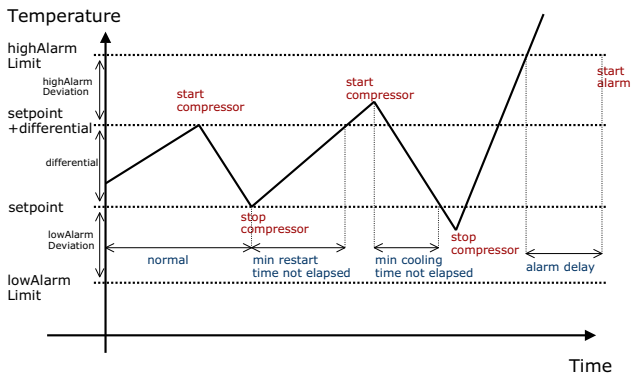


Figure 3: EKC Main Control Objective.

directly accessed from a visual Basic API on a MS Windows XP PC host via monitoring software provided by Danfoss. The EKC is connected to a MS Windows XP PC host, first via a LON network from the EKC to a EKC-gateway, and from the gateway to the PC via a RS-232 serial connection. The required hardware and software were provided by Danfoss. As recommended by Danfoss we implemented the adaptation software by accessing the parameter database using the provided interface. However, UPPAAL-TRON only exists in UNIX versions, and thus it required a second UNIX-host computer connected to the MS windows PC using a TCP/IP connection properly configured to prevent unnecessary delaying of small messages. The adaptation software thus consists of a "thin" visualBasic part running on the MS windows host, and a C++ part interfacing to the TRON native adaptation API running on a UNIX host. It is important to note that this long chain (three network hops) adds both latency and uncertainty to the timing of events.

More seriously it turned out that the parameters representing sensor inputs are read-only, meaning that the test host cannot change these to emulate changes in sensor-inputs. Therefore some functionality (temperature based defrosting, sensor error handling, and door open control) related to these is not modeled and tested. The main sensor, the room temperature, is hardwired to a fixed setting via a resistor, but the sensed room temperature can be changed indirectly via a writable calibration parameter with the range $\pm 20\,^{\circ}C$.

It quickly became evident to us that the monitoring software was meant for "coarse grained" event logging and supervision by an operator, not as a (real-time) test interface. An important general lesson learned is that an IUT should provide an test interface with suitable means for control and observation. We are collaborating with Danfoss to provide a better test interface for future versions of the product.

3.3 Model Structure

We modeled a central subset of the functionality of the EKC as a network of UPPAAL Timed Automata, namely basic temperature regulation, alarm monitoring, and defrost modes with manual and automatic controlled (fixed) periodical defrost (de)activation. The allowed timing tolerances and timing uncertainties introduced by the adaptation software is modeled explicitly by allowing output events to be produced within a certain error envelope. For example, a tolerance of 2 seconds is permitted on the compressor-relay. In general, it may be necessary to model the adaptation layer as part of the model for the system under test. The abstract input/output actions are depicted in Figure 4.

From the beginning it was decided to challenge our tool. Therefore we decided that the model should be responsible of tracking

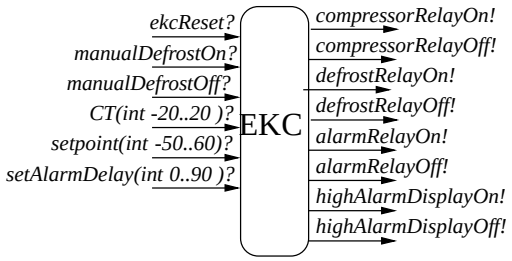


Figure 4: Model Inputs and Outputs.

the temperature as calculated by the EKC and base control actions on this value. To make this work, the computation part of the model and also its real-time execution must be quite precise. This part of the model thus approximates the continuous evolution of a parameter, and almost approaches a model of a hybrid system, which is on the limit of the capability of timed automata. An alternative would be to monitor the precision of the calculated temperature in the adaptation software and let that generate events (e.g., *alarmLimitReached!*) to the model as threshold values are crossed. This would yield a simple and more abstract "pure" event driven model.

The model consists of 18 concurrent components (timed automata), 14 clock variables, and 14 discrete integer variables, and is thus quite large. The main components and their dependencies are depicted in Figure 5 and explained below.

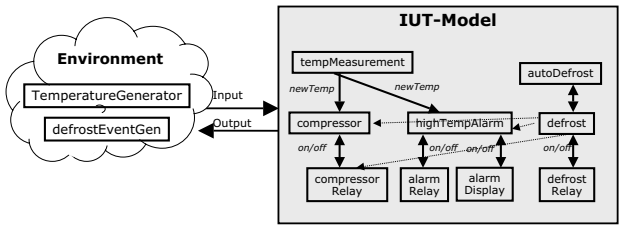


Figure 5: Main Model Components

The **Temperature Measurement** component periodically samples the temperature sensor and calculates a new estimated room air temperature. The **Compressor** component controls the compressor relay based on the estimated room temperature, alarm and defrost status. The **High Temperature Alarm** component monitors the alarm state of the EKC, and triggers the alarm relay if the temperature is too high for too long. The **Defrost** component controls the events that must take place during a defrost cycle. When defrosting the compressor must be disengaged, and alarms must be suppressed until *delayAfterDefrost* time units after completion. Defrosting may be started manually by the user, and is engaged automatically with a certain period. It stops when the defrosting time has elapsed, or when stopped manually by the user. The **Auto Defrost** component implements automatic periodic time based defrosting. It automatically engages the defrost mode periodically. The **Relay** component models a digital physical output (compressor relay, defrost relay, alarm relay, alarm display) that when given a command switches on (respectively off) within a certain time bound. The **Temperature Generator** is a part of the environment that simulates the variation in room temperature, currently alternatingly increases the temperature linearly between minimum and maximum temperature, and the reverse. Finally, the **Defrost Event Generator** environment component randomly issues user initiated defrost start and stop commands.

4. COMPONENT MODELING AND REVERSE ENGINEERING

The modeling effort was carried out by computer scientists without knowledge of that problem domain based on the EKC documentation provided by Danfoss. It only consisted of the internal requirements specification and the users manual, both in informal prose. In addition we had access to questioning the Danfoss Engineers via email and two meetings, but no design documents or source code were available. In addition we were given documentation about the EKC PC-monitoring software and associated API allowing us to write the adaptation software.

In general the documentation was insufficient to build the model. In part this was due to a lack of a detailed understanding of the implicit engineering knowledge of the problem domain and how previous generations of controllers worked. But more importantly much functional behavior and especially timing constraints were not explicitly defined. In general the requirements specification did not state any timing tolerances, e.g, the allowed latency on compressor start and stop when the calculated temperature crosses the lower or higher thresholds.

Therefore the modeling involved a lot of experimentation to deduce the right model and time constraints, which to some extent best can be characterized as reverse engineering or model-learning [4]. Typically the work proceeded by formulating a hypothesis of the behavior and timing tolerances as a model (of the selected aspect/sub functionality), and then executing TRON to check whether or not the EKC conformed to the model. If TRON gave a fail-verdict the model was revised (either functionally, or by loosening time tolerances). If it passed the timing tolerances were tightened until it failed. The process was then iterated a few times, and the Danfoss engineers were consulted to check whether the behavior of the determined model was acceptable.

In the following we give a few examples of this procedure.

4.1 Room Temperature Tracking.

The EKC estimates the room temperature from Equation 2 based on periodically samples of the room temperature sensor, and bases most control actions like switching the compressor on or off on this value. However, the requirements only requires a certain precision on the sampling accuracy of the temperature sensors ($\pm 0.5\,^{\circ}C$) and a sensor sampling period of at most 2 seconds, and nothing about how frequently the temperature should be reevaluated. This led to a series of tests where the temperature change rate, the sampling period, and temperature tolerance were changed to determine the best matching configuration. The model now uses a period of 1.2 seconds, and allows $\pm$ 2 seconds tolerance on compressor start/stop.

4.2 Alarm Monitoring

Executing TRON using our first version of the high temperature alarm monitor caused TRON to give a fail-verdict: The EKC did not raise alarms as expected. The model shown in Figure 6 assumed that the user's clearing of the alarm would reset the alarm state of the EKC completely. The consequence of this is that the EKC should raise a new alarm within *alarmDelay* if the temperature remained above the critical limit. However, it did not, and closer inspection showed that the EKC was still indicating high temperature alarm in its display, even though the alarm was cleared by the user. The explanation given by Danfoss was that clearing the alarm only clears the alarm relay (stopping the alarm noise), not the alarm state which remains in effect until the temperature drops below the critical limit. The model was then refined, and includes the *noSound_Displaying* location in Figure 7.

4.3 Defrosting and Alarm Handling.

A similar discrepancy between expected and actual behavior detected by TRON was in the way that the alarm and defrost functions interacts. After a defrost the room temperature naturally risks being higher than the alarm limit, because cooling has been switched off during the defrost activity for an extended period of time. Therefore a high temperature alarm should be suppressed in this situation which can be done by configuring the EKC parameter *alarmDelayAfterDefrost*. However, reading different sections of the documentation gives several possible interpretations:

1. When defrosting stops and the temperature is high, alarms must be postponed for *alarmDelayAfterDefrost* in addition to the original *alarmDelay*, i.e., never alarms during a defrost.
2. Same as above (1) except it is measured from the time where the high alarm temperature is detected, even during a defrost.
3. When defrosting stops and the temperature is high, alarms must be suppressed for *alarmDelayAfterDefrost*, i.e., *alarmDelayAfterDefrost* replaces the original *alarmDelay* after a defrost until the the temperature becomes below critical, after which the normal *alarmDelay* is used again.

The engineering department could not give an immediate answer to this (without reluctantly consulting old source code), but based on their experiences and requirements for other products they believed that 3 is the correct interpretation. Note that we are not suggesting that the product was implemented without a clear understanding of the intended behavior, only that it was not clear from its documentation.

4.4 Defrost Time Tolerance.

Another discrepancy TRON found was that defrosting started earlier than expected or was disengaged later. It turned out that the internal timer in the EKC responsible for controlling the defrost period has a very low precision (probably because defrosting is rare (e.g., once a day) and has along duration (lasts several hours)). The default tolerance used in the model on the relays thus had to be further relaxed.

5. QUANTITATIVE EVALUATION

During a test-run, the testing algorithm computes, on a per timed event basis, the set of symbolic states in the model that can be reached after the timed event trace observed so far, and generates stimuli and checks the validity of IUT-outputs based on this state-set.

Since we use a non-deterministic model to capture the timing and threshold tolerances of the IUT and since internal events in a concurrent model may be executed in (possibly combinatorially many) different orders, this set will usually contain numerous possible states. The state-set reflects the allowed states and behavior of the IUT, and intuitively, the larger the state-set, the more uncertain the tester is about the state of the implementation.

Since we generate and execute tests in real-time the state-set must also be updated in real-time. Obviously, the model and the state-set size affects how much computation time this takes, and one might question wheter doing this is feasible in practice. In the following we investigate whether real-time online testing is realistic for practical cases, like the Danfoss EKC.

Figure 8 plots the evolution of the state-set size (number of *symbolic states*) for a sample test run. Also plotted in the graph is the input temperature, temperature threshold value for high temperature (compressor must switch on) and high temperature alarm (the

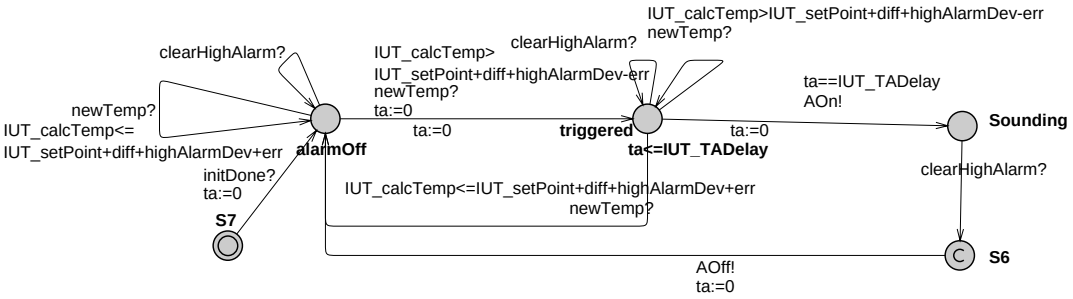


Figure 6: First High Temperature Monitor.

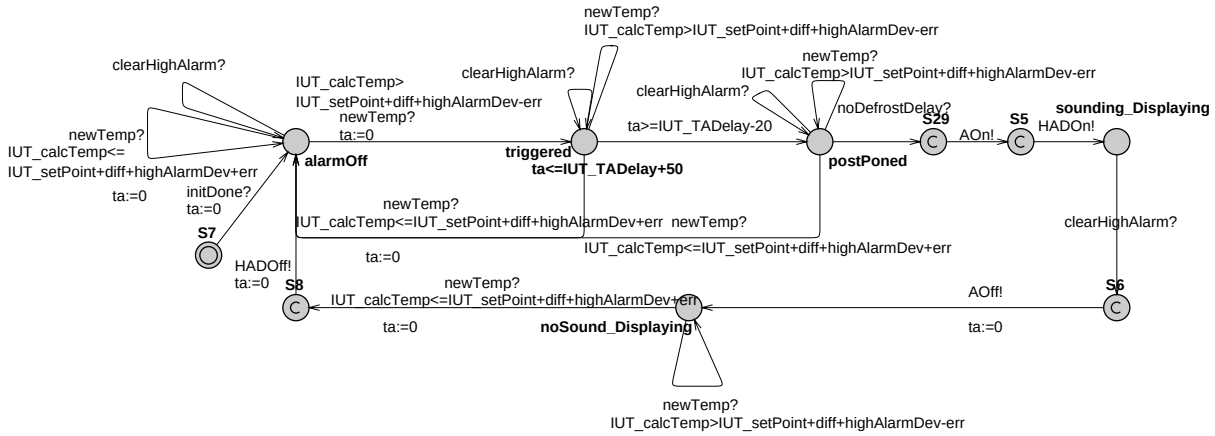


Figure 7: Second High Temperature Monitor

alarm must sound if it remains high for more than *alarmDelay* (120 sec) time units.

It is interesting to observe how the state-set size depends on the model behavior. For instance, the first larger increase in state-set size occurs after 55 seconds. At this time the temperature crosses the limit where the compressor should switch on. But due to the timing tolerances, the model does not "know" if the compressor-relay is in on-state or off-state, resulting in a larger state-set. The state-set size then decreases again, only to increase again at 93 seconds at which a manual defrost period is started. The next major jump occurs at 120 seconds and correlates nicely with the time where the temperature crosses high-alarm limit and the alarm monitor component should switch into *triggered* state. Similarly, 260 second into the run, the temperature drops below the threshold, and there is no uncertainty in the alarm state. The fluctuations inside this period is caused by a manually started and stopped defrost session. In fact 5 defrost cycles are started and stopped by the tester in this test run. The largest state-set size (960 states) occurs at 450 seconds and correlates to the time-out of a defrost cycle. There is a large tolerance on the timer controlling defrosting, and hence the model can exhibit many behaviors in this duration.

The state-set contains most often less than a few hundred states. Exploring these is unproblematic for a modern model-checking engine employed by TRON. Figure 9 plots the the cpu-time used to update the state-set for delay-actions (typically the most expensive operation) for 5 test-runs of our model on a Dual Pentium Xeon 2.8 GHz CPU (one used). It can be seen that the far majority of state set sizes are reasonably small. Updating medium sized state-sets with around 100 states requires only a few milli-seconds (ms) of cpu-time. The largest encountered state-sets (around 3000 states) are very infrequent, and requires around 300 ms.

Real-time online testing thus appear feasible for a large range of embedded systems, but also that very non-deterministic model such as the EKC-model may limit the granularity of time constraints that can be checked in real-time.

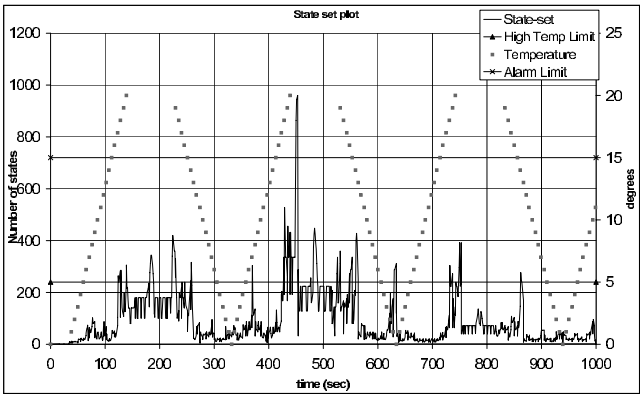


Figure 8: Evolution of State-set.

6. CONCLUSIONS AND FUTURE WORK

Our modeling effort shows that it is possible to accurately model the behavior of EKC like devices as Timed Automata and use the resulting model as a test specification for online testing.

It is possible to model only selected desired aspects of the system behavior, i.e. a complete and detailed behavioral description is not required for system testing. Thus, model based testing is feasible even if a clear and complete formal model is not available from the

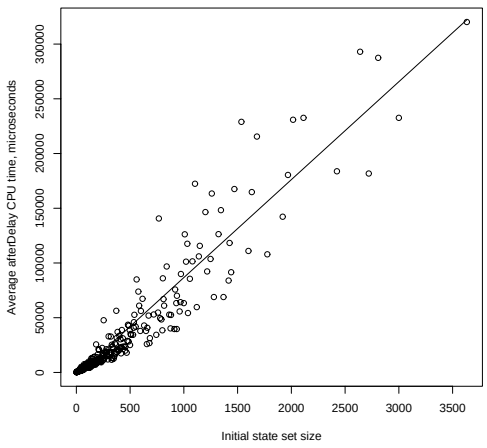


Figure 9: Cost of State-set Update: Delay action

start, although it will clearly benefit from more explicit modeling during requirements analysis and system design.

In the relative short testing time, we found many discrepancies between our model and the implementation. Although many of these were caused by a wrong model due to incomplete requirements or mis-interpretations of the documentation, and not actual implementation errors, our work indicates that online testing seems an effective technique to find discrepancies between the expected model behavior and actual behavior of the implementation under test. Thus there are also reasons to believe that it is effective in detecting actual implementation errors.

It should be mentioned that the EKC is a mature product that has been produced and sold for a number of years. Future work includes testing a less mature version of a EKC like controller.

Performance-wise we conclude that real-time online testing appear feasible for a large range of embedded systems. To target even faster real-time systems with even time constraints in the (sub) milli-second range we plan to separate our tool into two parts, an environment emulation part, and a IUT monitoring part. Monitoring need not be performed in real-time, and may in the extreme be done offline. The model that will need to be interpreted in real-time is thus much smaller and can be done much faster.

We are extending our tool with coverage measurements, coverage based guiding, and features for error diagnosis. By importing a trace collected during a test run into UPPAAL it can be run against the IUT model. It can also be replayed against the actual IUT (within the limits of non-determinism). Future work also includes extensions for testing hybrid systems, i.e., systems with general continous state evolution besides progress of time, e.g. by using hybrid automata, but analyzing such systems exactly and formally is much more difficult and costly.

Acknowledgments

We would like to thank Danfoss for providing the case-study and especially Finn Andersen, Peter Eriksen, and Søren Winkler Rasmussen from Danfoss for engagement and constructive information and help during the project.

7. REFERENCES

[1] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Comput. Sci.*, 126(2):183–235, Apr. 1994.

[2] D. A/S. Danfoss internet website, http://www.danfoss.dk.

[3] G. Behrmann, J. Bengtsson, A. David, K. Larsen, P. Pettersson, and W. Yi. Uppaal implementation secrets. In *Formal Techniques in Real-Time and Fault-Tolerant Systems: 7th International Symposium, FTRTFT 2002*, pages 3–22, September 2002.

[4] T. Berg, B. Jonsson, M. Leucker, and M. S. August. Insights to Angluin's Learning. In *International Workshop on Software Verification and Validation (SVV 2003)*, 2003.

[5] E. Brinksma, K. Larsen, B. Nielsen, and J. Tretmans. Systematic Testing of Realtime Embedded Software Systems (STRESS), March 2002. Research proposal submitted and accepted by the Dutch Research Council.

[6] K. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using UPPAAL. In *Formal Approaches to Testing of Software*, Linz, Austria, September 21 2004. Lecture Notes in Computer Science.

[7] K. Larsen, M. Mikucionis, and B. Nielsen. Online Testing of Real-time Systems using Uppaal: Status and Future Work. In E. Brinksma, W. Grieskamp, J. Tretmans, and E. Weyuker, editors, *Dagstuhl Seminar Proceedings volume 04371: Perspectives of Model-Based Tes ting*, Schloss Dagstuhl, D-66687 Wadern, Germany., September 2004. IBFI gem. GmbH, Schloss Dagstuhl.

[8] K. Larsen, P. Pettersson, and W. Yi. UppAal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.

[9] M. Krichen and S. Tripakis. Black-box Conformance Testing for Real-Time Systems. In *Model Checking Software: 11th International SPIN Workshop*, volume LNCS 2989. Springer, April 2004.

[10] M. Mikucionis. Uppaal tron internet page, http://www.cs.aau.dk/~marius/tron.

[11] M. Mikucionis, K. Larsen, and B. Nielsen. Online on-the-fly testing of real-time systems. Technical Report RS-03-49, Basic Research In Computer Science (BRICS), Dec. 2003.

[12] M. Mikucionis, B. Nielsen, and K. Larsen. Real-time system testing on-the-fly. In *the 15th Nordic Workshop on Programming Theory*, number 34 in B, pages 36–38, Turku, Finland, October 29–31 2003. Åbo Akademi, Department of Computer Science, Finland. Abstracts.

[13] M. Mikucionis and E. Sasnauskaite. On-the-fly testing using UPPAAL. Master's thesis, Department of Computer Science, Aalborg University, Denmark, June 2003.

[14] J. Peleska. Formal Methods for Test Automation - Hard Real-Time Testing of Controllers f or the Airbus Aircraft Families. In *Integrated Design and Process Technology (IDPT-2002)*, 2002.

[15] M. K. S. Bensalem, M. Bozga and S. Tripakis. Testing conformance of real-time applications with automatic generation of ob servers. In *Runtime Verification 2004*, 2004.

[16] S. Tripakis. Fault Diagnosis for Timed Automata. In *Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02)*, volume LNCS 2469. Springer, 2002.

[17] J. Tretmans. Testing concurrent systems: A formal approach. In J. Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer-Verlag, 1999.