Testing Real-Time Embedded Software using UPPAAL-TRON an industrial case study

Kim G. Larsen and Marius Mikucionis and Brian Nielsen and Arne Skou

Aalborg University
Center of Embedded Software Systems, CISS
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.*

1 Introduction

Model-based testing is a promising approach for improving the testing of embedded systems. Given an abstract formalized behavioral model (ideally developed as the design process) of aspects of the implementation under test (IUT), a test generation tool automatically explores the model to generate test cases that can be executed against 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 [7]. 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 world-wide for leadership in Refrigeration & Air Conditioning, Heating & Water and Motion Controls [1]. 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.

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 [15] in the context of failure diagnosis. Later that work has been extended by Krichen and Tripakis [8] 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 [14]. 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.

Our approach, previously presented in [4, 12, 10]; an abstract appeared in [11]), uses the mature UPPAAL language and model-checking engine to perform *relativized timed input/output conformance*, meaning that we take environment assumptions explicitly into account.

Online testing based on timed CSP specifications has been proposed and applied in practice by Peleska [13].

In Section 2 we introduce the concepts behind our test-

ing framework. Section 3 describes the case, Section 4 our modeling experiences, and Section 5 performance results. Section 6 concludes the paper. For the keen reviewer the entire model is included as an appendix.

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

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 stabilized, or it may be too costly or ineffective to achieve for the most general environment. It further turns out that explicit environment models have further 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 adaptor 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.

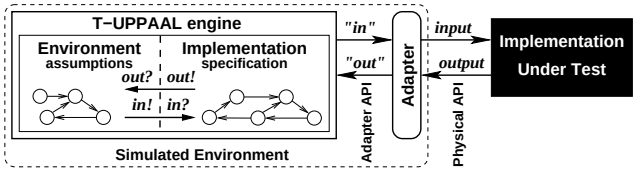


Figure 1. TRON test setup.

We extended the input/output conformance relation **ioco** [16] between a formal specification and its black-box implementation to the timed setting and relatively to a given environment.

Relativized timed input/output conformance $\mathbf{rtioco}$ [5] is defined formally in Equation 1.

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

Intuitively $\mathbf{rtioco}_e$ 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. 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 modelled as a network of timed automata. We explain timed automaton by example, and refer to to [2] for formal syntax and semantics. A timed automaton is essentially a finite state machine with input/output actions (distinguised respectivelyt by ? and !) augmented with a set of special real-valued clock variables that may be used to guard when transitions may take place. 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(c)). 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 relativised 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 on matching input/output transitions. In a *closed* timed automata network all output action transitions have a corresponding input action transition. UPPAAL supports timed automata networks with additional integer variable types, broadcast (one-to-many) synchronizations and other extensions.

2.3 Environment Modeling

For testing purposes we construct closed timed automata networks which are partitioned into two separate parts: the IUT and its environment, both required to be input enabled. The test specification also specifies which actions are observable inputs and outputs. These must be consistent with the partitioning.

Figures 2(b) to 2(d) shows three possible environment assumptions for $\mathcal{C}^r$. Figure 2(b) shows the universal (most general) and completely unconstrained environment $\mathcal{E}_0$ where room temperature may change unconstrained and may change (discretely) with any rate. However, this may not reflect the reality as temperature normally evolves slowly and continuously, e.g., it cannot change drastically from *Low* to *High* and back unless through *Med*.

Figure 2(c) shows the environment model where the temperature changes through *Med* range and with a speed bounded by d. Figure 2(d) shows an even more constrained environment $\mathcal{E}_2$ that tests the controller under the assumption that the cooling device works, e.g., temperature never increases when cooling is on. More restrictive environments reduce the effort and cost for testing, but 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 irrelevant.

For example, suppose $\mathcal{C}^r$ is implemented by a timed automaton equal to $\mathcal{C}^r$, except the transition $up \xrightarrow{Low} dn$ is missing. This error can be detected under $\mathcal{E}_0$ and $\mathcal{E}_1^{3d<r}$ but not under $\mathcal{E}_2$ by executing the timed trace d·*Med!*·d·*High!*·d·*Med!*·d·*Low!*·ε and observing output *On* shortly after $\varepsilon \leq r$ instead of *Off*. Of course, if the implementation can be fast enough to issue *On* just after the first *Med* to break our trace, then only $\mathcal{E}_0$ is capable of sufficiently fast traces *High!*·*Low!*·ε to test that transition.

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. In order to simulate the environment and monitor the implementation, TRON computes and maintains the set of symbolic states that can be reached in the (composed) specification model after the timed trace observed so far by using the UPPAAL engine to traverse internal, delay and observed action transitions. A symbolic state is a particular set of inequations on clock variables that denotes a potentially infinite set of concrete states.

Based on this 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. TRON randomly chooses between letting time pass by some (random) amount and silently observing the IUT, or offering a randomly selected relevant input.

Consider the system $(\mathcal{C}^r, \mathcal{E}_i^d)$. The initial state-set is the single symbolic state: $\{\langle \mathit{off}, L, x = 0, y = 0\rangle\}$. After a delay of d or more, $\{Med\}$ is the set of possible inputs. Suppose that TRON issues Med after δ time units. The state-set now consists of two states: $\{\langle \mathit{off}, M, x = \delta, y = 0, t = \delta\rangle, \langle up, M, x = 0, y = 0\rangle\}$. If *On* is received later the first element in the state-set will be eliminated.

Currently TRON is available to download via the Internet free of charge for evaluation, research, education and other non-commercial purposes [9]. TRON supports all UPPAAL modeling features including non-determinism, provides timed traces as test log and a verdict as the answer to `rtioco` relation.

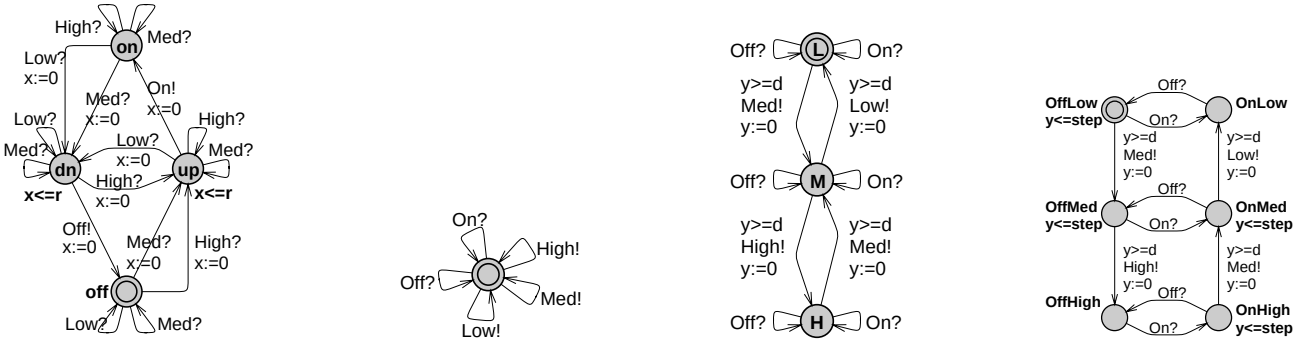
3 The Danfoss EKC-201 Refrigeration Controller

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



(a) $\mathcal{C}^r$. (b) $\mathcal{E}_0$. (c) $\mathcal{E}_1^d$. (d) $\mathcal{E}_2^{d,s}$.

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

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 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.

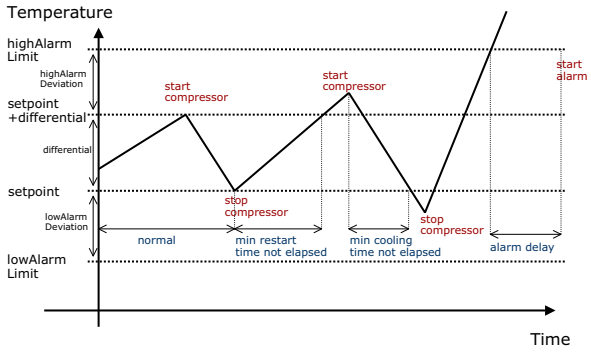


Figure 3. EKC Main Control Objective.

3.2 Test Adaptation.

A few comments are necessary about the test adaptor 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 indirectly 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 there 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 host computer connected using a TCP/IP connection properly configured to prevent unnecessary delaying of small messages. The adaptation software thus consists of a "thin" visualBasic part, and a C++ part interfacing to the TRON native adaptation API. It is important to note that this long chain 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 in 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.

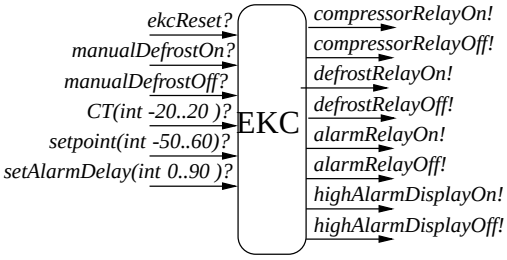


Figure 4. Model inputs and outputs.

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

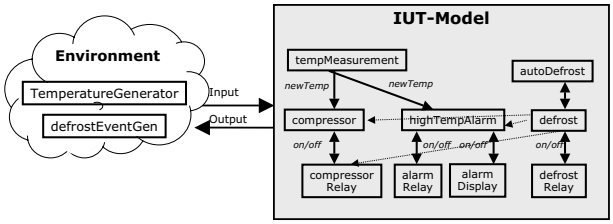


Figure 5. Main Model Components

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 is disengaged, and alarms 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 [3]. 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.,ónce 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.

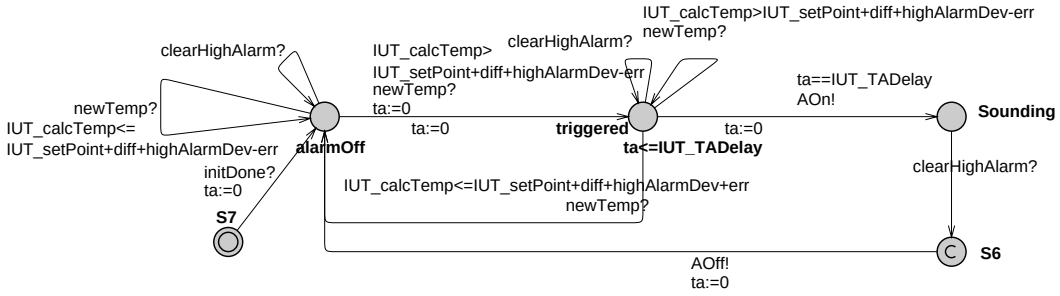


Figure 6. First High Temperature Monitor.

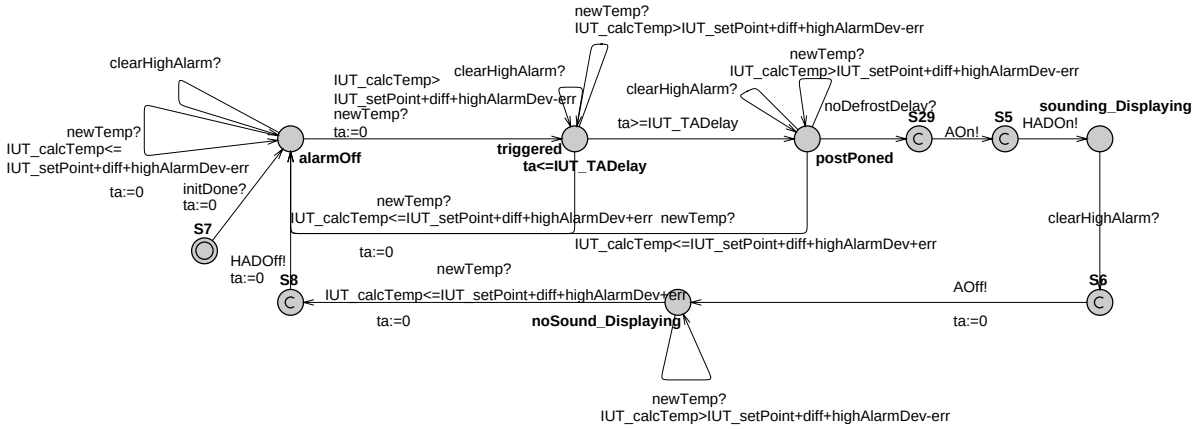


Figure 7. Second High Temperature Monitor

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

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.

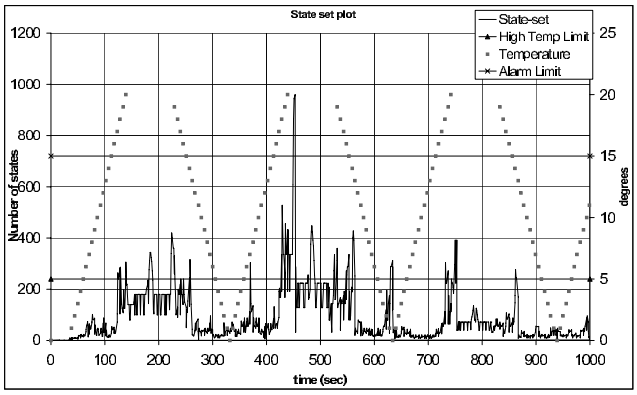


Figure 8. Evolution of State-set.

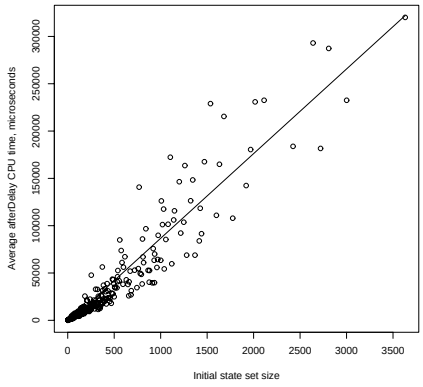


Figure 9. Cost of State-set Update: Delay action

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 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 currently extending our tool with coverage measurements, coverage based guiding, and features for error diagnosis. These features include importing the trace collected during a test run into UPPAAL and from here running it against the IUT model. It can also be replayed against the actual IUT(within the limits of its non-determinism).

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

References

[1] Danfoss internet website.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.

A Model WalkThrough

This section offers detailed technical model documentation and is organized by a by-component walk-through of the model. Each subsection describes one component (or closely related components). First the channels, variables and constants used in the component is defined, and then the behavior is explained. One model time unit corresponds to 0.1 seconds of real-time. The model is available at `http://www.cs.aau.dk/~bnielsen/compressor.xml`.

A.1 Calibrated Temperature Communication

CT!, CT?: Communicates a change in callibration temperature $\pm 20\ ^\circ C$ between ENV model and IUT model.

reportTemp?, reportDone!: Used to synchronize the communication of environment temperature with other environment components, specifically *ENV_TemperatureSinus*.

receivedTemp!: Used to indicate to the temperature calculation component *IUT_TemperatureMeasurementErr* that a new temperature value has been received.

fixedTemp: The actual physical temperature sensor of the EKC is hardwired via a resistor to a fixed value of $16.6\ ^\circ C$.

CT_env: The room temperature stored by the environment model (or more precisely, the callibration offset from the fixed input temperature sensor).

CT_iut: The room temperature sensed by the EKC.

IUT_calcTemp: The weighted averaged room temperature. Each new sample is weighted 20% compared to the previous calculated value.

t: A local clock used to constrain the allowable slack in the input of a new temperature.

The environment model emulates changes in room temperature. The actual room temperature is stored in the environment part of the model under the name *CT_env*, "calibrated Temperature, environment". Recall that the room temperature can only be controlled indirectly via the temperature calibration feature of the EKC. The value of this variable is communicated to the iut part of the model using value passing into the variable *CT_iut* along with *CT*. The value passing is realised by the two components *ENV_TemperatureReporter* and *IUT_TemperatureReceiver* shown in Figure 10

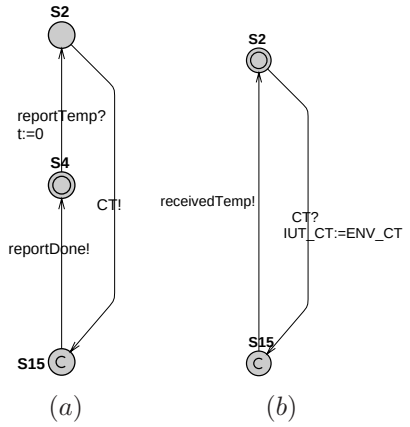


Figure 10. Communicating Calibrated Temperature between ENV and IUT. (a) *ENV_TemperatureReporter*, **(b)** *IUT_TemperatureReceiver*

A.2 Temperature Calculation

receivedTemp?: Used to indicate to the temperature calculation component that a new temperature sample has been received.

newTemp!: Broadcast channel used to indicate to other IUT components that a new temperature estimate has been calculated, and that they should re-evaluate their state.

fixedTemp: The actual physical temperature sensor of the EKC is hardwired via a resistor to a fixed value of $16.6\ ^\circ C$.

CT_iut: The room temperature sensed by the EKC. I.e., the temperature is *fixedTemp+CT_IUT* in the IUT model, and *fixedTemp+CT_ENV* in the environment model.

IUT_calcTemp: The weighted averaged room temperature. Each new sample is weighted 20% compared to the previous calculated value.

t: A local clock controlling the periodic sampling of the *CT_iut* variable and following calculation of the weighted average.

samplingPeriod: Constant sampling period (1.2 seconds).

The calculation of the weighted averaged room temperature is done by the *IUT_MeasurementErr* component in Figure 11. It periodically samples the received (calibrated) temperature, and based on this value computes the weigted average, and broadcasts the change via the *newTemp* action.

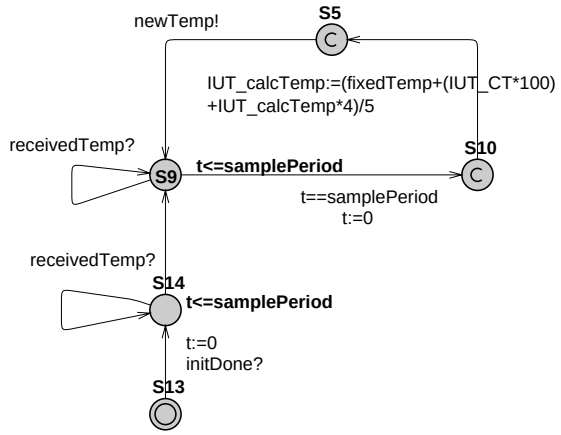


Figure 11. *IUT_MeasurementErr*: **Sampling Temperature Sensor.**

A.3 IUT_Compressor

newTemp?: The compressor state is re-evaluated whenever the calculated temperature has changed. This is indicated via reception of this event.

COn!, COff!: Actions used to switch the compressor relay on or off.

IUT_calcTemp: The weighted averaged room temperature. Each new sample is weighted 20% compared to the previous calculated value.

IUT_setPoint, diff: The temperature should be regulated to be within $IUT\_setPoint\ °C$ and $IUT\_setPoint + diff\ °C$.

on: global boolean variable (shared with component *IUT_defrost*) to store the compressor state, ie. tracks whether cooling is currently on or off.

defrosting: boolean global variable (shared with *IUT_defrost*) to store the defrosting state.

err: Error tolerance on the threshold values of the calculated temperature

minRestartTime: A minimum amount of time must elapse before the compressor may be switched back on (default value is 0 seconds).

minCoolingTime: A minimum amount of time must elapse before the compressor may be switched back off (default value is 0 seconds).

Xcompr: Local clock used to restrict the speed which the compressor engages/disengages.

This component models the compressor functionality of the EKC. It is triggered by the *newTemp* action periodically generated by the *IUT_TemperatureMeasurement* component. Generally cooling must be on when the calculated temperature exceeds the *setPoint* plus *differential*, and off when below the *setPoint*. Because the exact precision of the EKC is unknown, the models allows some tolerance $\pm err$ on the threshold values of the calculated temperature. In consequence, when the calculated temperature is "around" a threshold value, the exact cooling state of the EKC is unknown. This is modeled through a non-deterministic choice between switching on (off) the compressor or leaving it off (on), see Figure 12.

- If the EKC is defrosting it may not switch on the compressor.
- The compressor is switched on by issuing the *COn* to the *Compressor Relay* if it is not defrosting, not already on, the calculated temperature exceeds the *setPoint* plus *differential* minus *error* tolerance, and *minRestartTime* has elapsed.
- The compressor is switched off by issuing the *COff* to the *Compressor Relay* if it is not defrosting, not already off, the calculated temperature is below the *setPoint* plus plus *error* tolerance, and *minCoolingTime* has elapsed.
- Otherwise the state is unchanged.

A.4 IUT_Relay

RON?, ROFF: triggers the relay to go on, or off respectively.

realOn!, realOff!: controls the physical outputs of the relay.

t: Local clock used to limit the switching speed of the relay.

switchDelay: models the uncertainty in the exact switching time of the relay, or time delay through the adaptation software.

This template in Figure 13 models a generic on/off switch (relay) with some time tolerance (*switchDelay*) on the switching time. It is parameterized through four channels: inputs *RON* and *ROFF* are used to trigger the relay to go on, or off respectively. The outputs *realOn* and *realOff* are the physical output actions. Initially the relay may be on or off. The (*switchDelay*) is used to model uncertainty in the exact swithching time of the relay, or time delay through the adaptation software. Note that the relay is input enabled

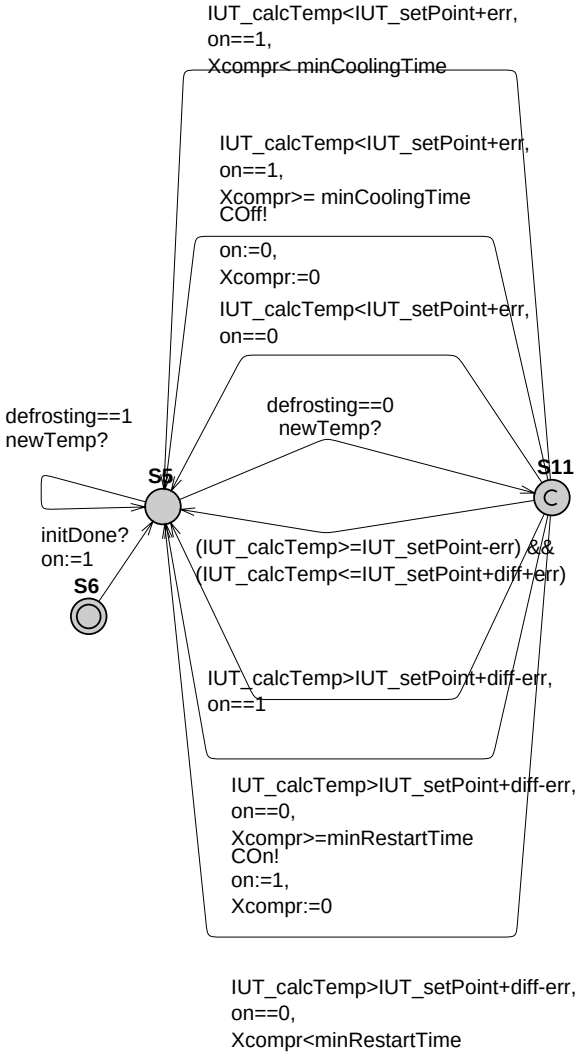


Figure 12. Compressor Control Model (*IUT_Compressor*.

such that if instructed to go back off while in location *OnTriggered* (but has not yet fired the physical *realOn!*) it will go off without changing physical output state.

There exists four instances of the relay-template: *IUT_compressorRelay*, *IUT_alarmRelay*, *IUT_defrostRelay*, and *IUT_highAlarmDisplay*. The *IUT_highAlarmDisplay* does not correspond to a physical relay of the EKC but controls an alarm-on/alarm-off indication in its display.

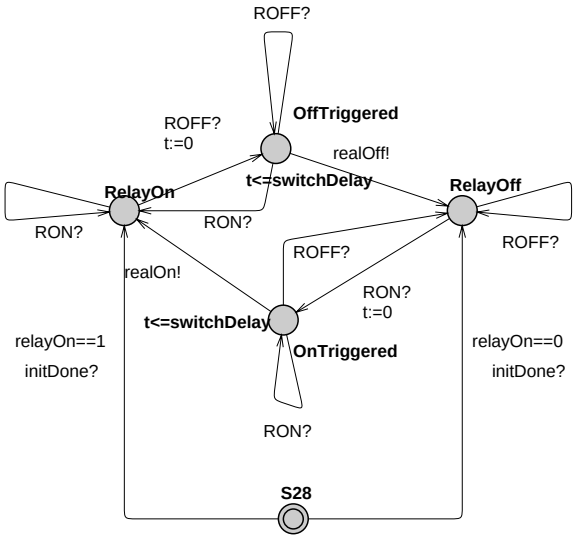


Figure 13. Relay Template.

A.5 High Temperature Alarm Monitoring

newTemp?: Input channel used to trigger temperature dependent transitions.

clearHighAlarm: User input to acknowledge high alarm status.

AOn!, *AOff!*: Used to switch alarm relay on and off.

HAOn!, *HAOff!*: Used to switch display alarm indication on and off.

noDefrostDelay?: Input channel used postpone the high temperature alarm, if raised during or shortly after a defrost cycle.

ta: Clock variable used locally to control time constraints in the high alarm monitor.

IUT_calcTemp: Global variable containing the average temperature calculated by the EKC.

IUT_setPoint+diff+highAlarmDev: High alarm threshold constant ($\pm err$) for high alarm status.

IUT_TADelay: Alarm must sound when the temperature exceeds the high alarm threshold value for more than *IUT_DADelay* time units.

The alarm status may be in 5 different locical states each modeled by a location in the high alarm monitor component in Figure 14:

alarmOff: The temperature is below the high alarm threshold.

triggered: The temperature is detected to be above the high alarm threshold, but the alarm should not be raised until *IUT_DADelay* time units have elapsed (as controlled by the location invariant). If the temperature drops to below the threshold in the mean time, the alarm monitor cancels the alarm and moves back to location *alarmOff*.

postponed: The temperature has now neen too high for too long, and the alarm should sound unless it has occured after a defrost cycle. The component will only stay in the *postponed* if this is the case; otherwise the *IUT_Defrost* controller will be able to synchronize urgently on the *noDefrostDelay?* channel, after which the alarm sound and display indication are switched on.

sounding_Displaying: In this state the alarm is both sounding and being displayed on the EKC. It remains in this state until the alarm is cleared manually by the operator. When cleared, the alarm sound is switched off, but it remain indicated on the display.

noSound_Displaying: When cleared the alarm remains indicated in the display until a temperature reading indicates that it has droped below the alarm threshold.

A.6 Defrost Control

In the time controlled defrost mode[1] the EKC is defrosting the cooler (evaporator) by activating the defrost relay for a fixed amount of time. A defrost cycle can be started manually by the user through key presses on the EKC, or automatically periodically. During a defrost cycle the compressor must remain off, no temperature alarm may be given, and high temperature alarms must be postponed some amount of time after completion of the defrost cycle.

A.6.1 IUT_AutoDefrost

The component *IUT_AutoDefrost* periodically starts a defrost cycle by issuing a *autoDefrostOn* action, see Figure 15

[1]Recall that in the current model temperature and real-time clock based defrosting is not possible to test using the available equipment.

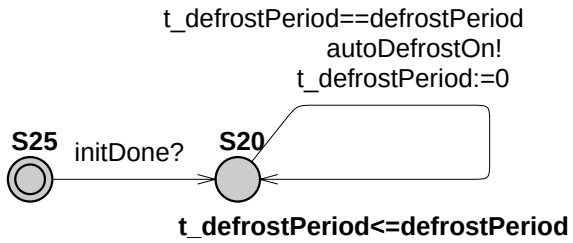


Figure 15. Auto-Defrost Control (*IUT_AutoDefrost*).

A.7 Defrost Control

manualDefrostOff?, *manualDefrostOn?*: input channels used to stop/start a manual defrost cycle.

autoDefrostOn: input used to start an automatic periodic defrost cycle.

DOff!, *Don!*: outputs used to switch the defrost relay off and on.

COff!: output action used to switch compressor off.

noDefrostDelay! Urgent output channel used to allow high temperature alarms.

on: global variable (shared with component *IUT_compressor*) to store the compressor state.

defrosting: boolean global variable (shared with *IUT_compressor*) to store the defrosting state.

t_defrostDuration: clock used to control the duration of a defrost cycle, and the duration of high alarm delays after a defrost.

defrostTime: A constant indicating the duration of a defrost cycle.

afterDefrostDelay: A constant indicating how long high temperature alarms must be postponed after a defrost cycle.

The state of the EKC during defrosting is controlled by the *IUT_Defrost* component in Figure 16. Initially, defrosting is off (location *OFF*). When engaged, either manually or automatically, the component switches off the compressor relay and switches on the defrost relay, and enters state defrosting (compressor state variable *on* is 0, boolean variable *defrosting* is 1, and the clock variable *t_defrostDuration* controlling the defrost duration is reset. The controller remains in defrost mode for *defrostTime* time plus minus some tolerance, or until disengaged manually through the *manualDefrostOff*. Then the defrost relay is swithced off,

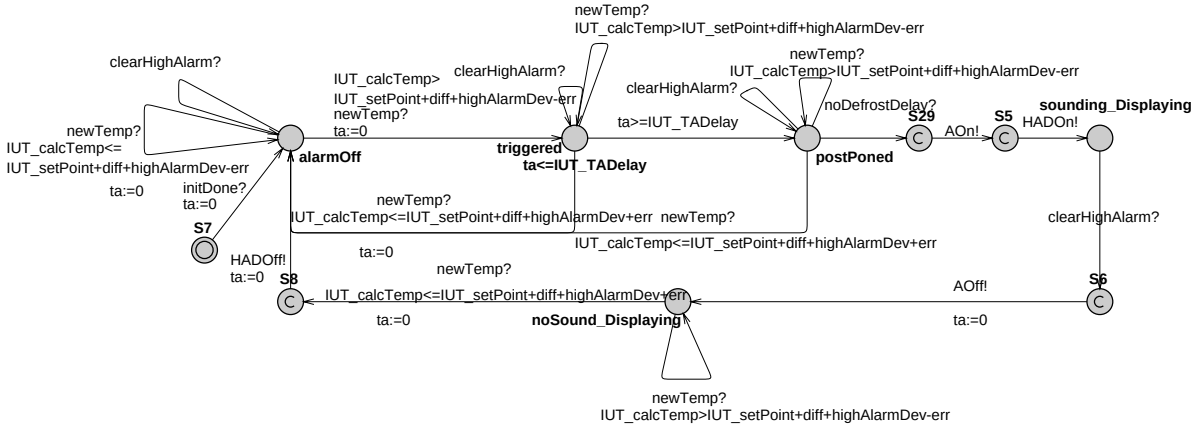


Figure 14. High Temperature Monitor (*IUT_HighTemperatureAlarm*).

and the controller enters location *afterDefrostDelay* that it occupies for approximately *defrostAlarmDelay* time units. The purpose of this location is to prevent high temperature alarms from being raised for at least *defrostAlarmDelay* time units after the system has defrosted. In contrast the *OFF* allows high temperature alarms by allowing synchronization on the urgent channel *noDefrostDelay*.

A.8 ENV_TempGen

This environment component controls execution of the main scenarie being tested. It consists of variying the temperature linearly ± 20 degrees. When the temperature has reached a maximum an user controlled *alarmReset* signal is sent, and the cycle continues, see Figure 17.

A.9 ENV_TemperatureSinus

This template belongs to the environment model, and it simulates the room temperature through sequences of linearly increasing or decreasing temperature settings, see Figure 18. It increases (or decreases, depending on the direction contained in variable *dir*) the temperature in the range from *minT* degrees to *maxT* degrees by *step* degrees with a step time of at least *delay* time units. It alternates between generating the temperature samples in increasing and decreasing order. When the temperature has changed in issues a *reportTemp* action and awaits acknowledgement (*reportDone*).

The initiation and direction may be controlled by other environment components through the *direction* variable and *continue* action. Likewise, the temperature change may be temporarlily suspended when a certain temperature is reached (variable *highStop* and *lowStop* respectively.

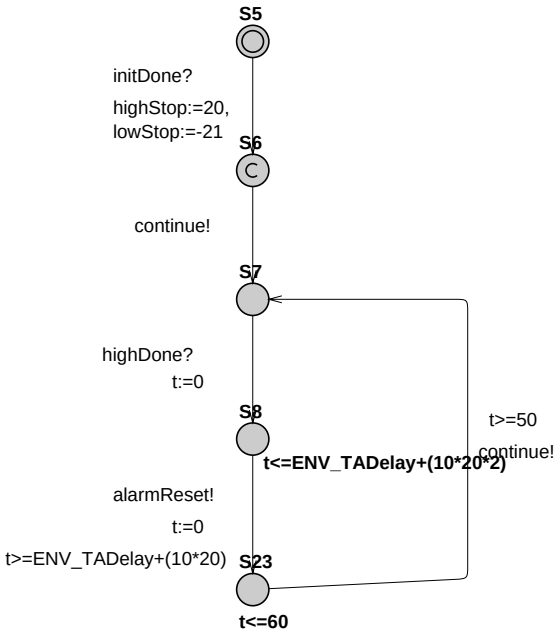


Figure 17. Temperature Generator (*ENV_TempGen*).

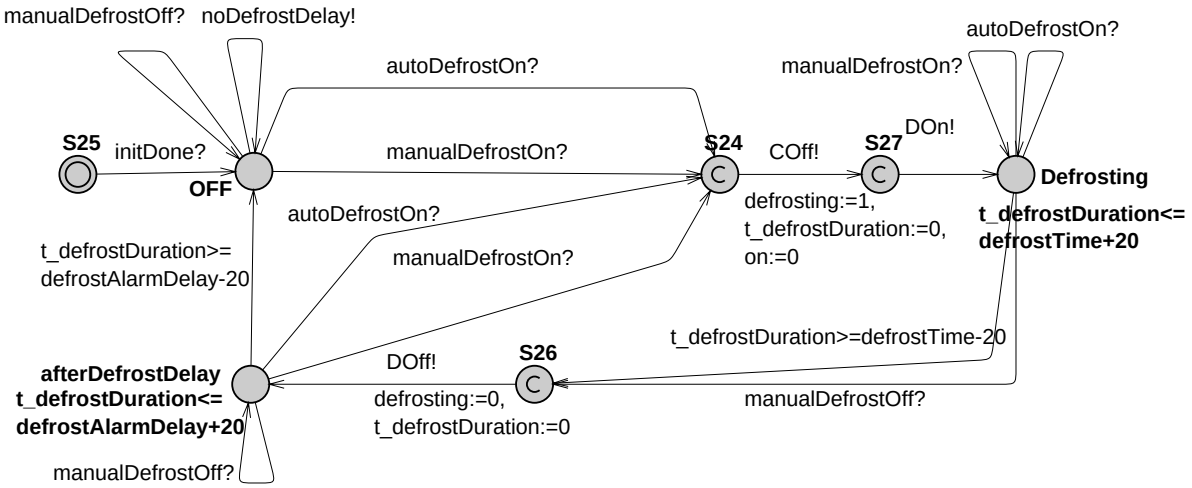


Figure 16. Defrost Control (*IUT_Defrost*).

A.10 ENV_DefrostGen

The defrost generator in Figure 19 starts and stops manual defrost with some minimum time seperation between these relatively rare events.

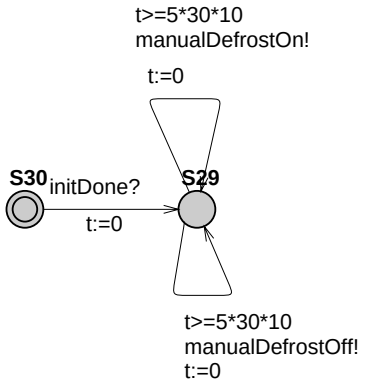


Figure 19. Manual Defrost Event Generator (*ENV_DefrostGen*).

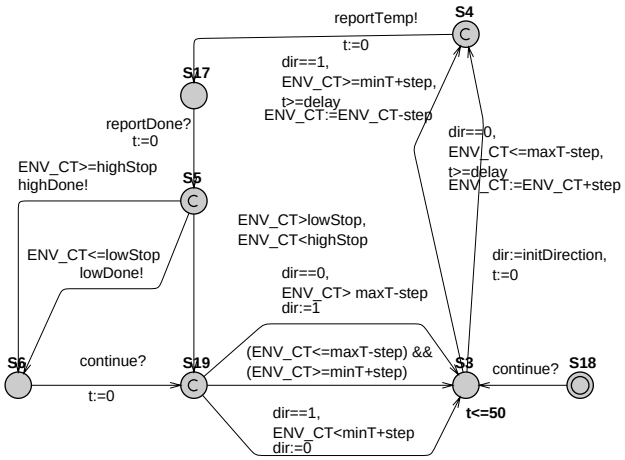


Figure 18. Room Temperature Simulator (*ENV_TemperatureSinus*).

A.11 Initialization

Before a test run the EKC must be brought to a known initial state and it must be configured with the parameters used during testing. Those that differ from the default settings must be changed explicitly. Figure 20 shows the current initialization sequence. First the EKC is reset. The EKC then generates an *alarmRelayOff* event. To speedup testing a number time parameters are changed:

- the parameter *alarmDelay* is changed to 1 minute.

- the default defrost period is 1 hour, the minimum allowed value.
- the duration of a defrost cycle is changed to 2 minutes.
- the alarm delay after defrost is changed to 2 mins.

The defrost mode is configured to be periodic (*setEKCPars*). The setPoint set to 20 degrees. To allow the EKC to settle and stabilize (eg. the weighted average temperature calculation) some time elapses before testing starts. Finally, the testing begins by broadcasting the *initDone* event to all other components.

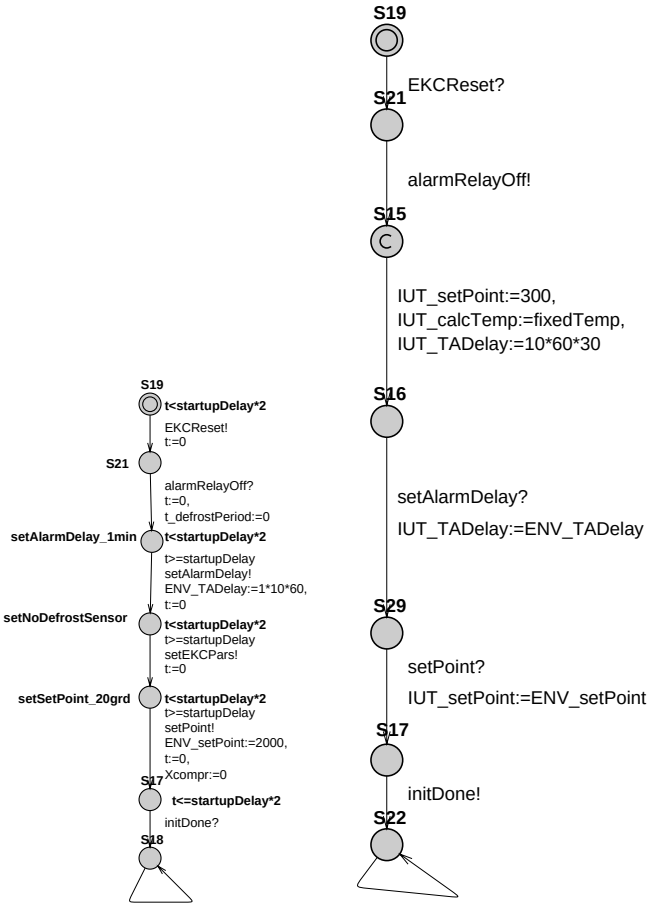


Figure 20. Initialization Sequence. (*Env_INIT* (a), and *IUT_INIT* (b)).

A.12 IUT_Action

The component in Figure 22 is a "closure" component that handles the actions that are so simple to handle that they to require a dedicated component. After initialization these include *alarmReset*, changes in *setPoint* and *alarmDelay*;

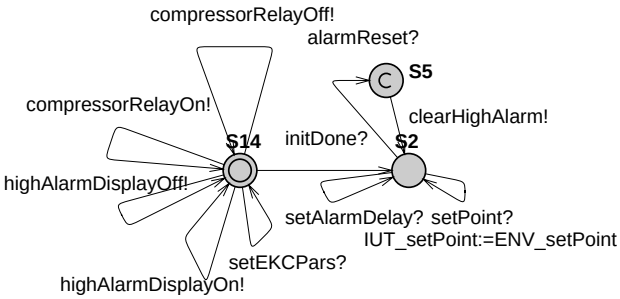


Figure 21. *IUT_Action*: Handling of simple actions.

A.13 ENV_Action

The component in Figure 22 is a "closure" component that handles the actions that are so simple to handle that they to require a dedicated component. All actions handled by this component does not change the state of the environment model.

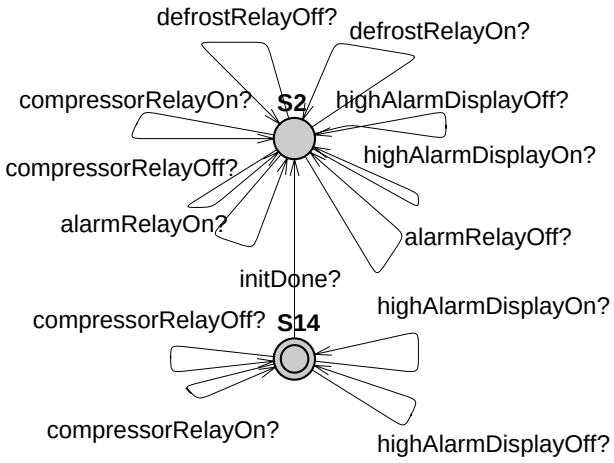


Figure 22. *ENV_Action*: Handling of simple actions.

A.14 Model Declarations

```
//OBSERVABLE CHANNELS
chan CT; // 1, temperature callibration send as input:
int[-20,20] ENV_CT;
chan setEKCPars; // 2, send the following EKC parameters as input:
int EKCPar13 := 1; //change default defrost period to 1hr
int EKCPar14 := 2; //change default defrost duration to 2 mins.
int EKCPar17 := 0; //we have no defrost temp sensor
int EKCPar18 := 2; //change default defrost alarm delay to 2 mins.
chan setPoint; // 3, send the target temp. value as input:
int [-6000,5000] ENV_setPoint; //the target temperature for regulation
chan setAlarmDelay; // 4, send the temp.alarm delay as input:
int ENV_TADelay:=1*10*60;
chan manualDefrostOn,manualDefrostOff,alarmReset; // 5, 6, 7
chan compressorRelayOn, compressorRelayOff; // 8, 9
chan defrostRelayOn, defrostRelayOff; // 10, 11
chan alarmRelayOn, alarmRelayOff; // 12, 13
chan highAlarmDisplayOn, highAlarmDisplayOff; // 14, 15
chan lowAlarmDisplayOn, lowAlarmDisplayOff; // 16, 17
chan EKCReset; // 18

//INTERNAL CHANNELS IUT MODEL
chan reportTemp,reportDone,receivedTemp;
broadcast chan initDone;
broadcast chan newTemp;
chan COn, COff; //internal compressor on/off
chan AOn, AOff; //internal alarmRelay on/off
chan DOn, DOff; //Internal defrostRelay on/off
chan HADOn, HADOff; //internal highAlarmDisplay On/Off
chan clearHighAlarm; //
chan autoDefrostOn; //internal channel to activate defrost periodically
urgent chan noDefrostDelay; //internal channel to prevent alarms going on untill defrost delay after defrost
//INTERNAL CHANNELS ENV MODEL
chan lowDone, highDone, continue;

//FIXED CONSTANTS
const compressorSwitchDelay 20;
const alarmSwitchDelay 30;
const defrostSwitchDelay 40;
const defrostPeriod 10*60*60*1; //1 hrs
const defrostTime 10*2*60; //2 mins
const defrostAlarmDelay 10*2*60; //2 mins
const firstAutoDefrostDelay 100; //10 secs.
const samplePeriod 12; //the sampling frequency on calibration temperature
const delay 50; //env temperature change speed (5 sec)
const fixedTemp 1670; //The fixed setting of of the air temperature sensor
const startupDelay 150; //Allow this amount of time to let EKC stabilize after reset
const err 50; //Error tolerance on calculated temp 1/5 degree
const diff 200; // the differential
const highAlarmDev 1000; //10 degr.
const minRestartTime 0;const minCoolingTime 0;

// IUT MODEL VARIABLES
int [0,90*60*10] IUT_TADelay; // 0-90 min
int[-20,20] IUT_CT; //IUT calibrated temperature
int[-6000,5000] IUT_calcTemp;
int [-6000,5000] IUT_setPoint; //the target temperature for regulation
clock t_defrostDuration; //tracks the duration of a defrost
clock t_defrostPeriod; // tracks the period between autostarts of defrost
clock Xcompr; // ensures min cooling time and min time to restart
int [0,1] defrosting; //defrosting or not
int [0,1] on; //compressor relay on
//ENVIRONMENT MODEL VARIABLES
int [-22,22] highStop,lowStop;
```

Figure 23. Model Global Declarations.