A Practical Application of UPPAAL and DTRON for Runtime Verification

Truscan, Dragos; Ahmad, Tanwir; Siavashi, Faezeh; Tuuttila, Pekka

Published in:
IEEE/ACM 2nd International Workshop on Software Engineering Research and Industrial Practice

DOI:
10.1109/SERIP.2015.15

Published: 01/01/2015

A Practical Application of UPPAAL and DTRON for Runtime Verification

Dragos Truscan*, Tanwir Ahmad*, Faezeh Siavashi*, Pekka Tuuttila†
* Åbo Akademi University, Turku, Finland
† Nokia Networks, Oulu, Finland

***Abstract*—We present our experience in applying runtime verification to a real-time system using UPPAAL timed automata and a set of related tools. We discuss the benefits and limitations, and propose a concrete solution to address the latter. Using the resulting solution we are able to run quick validation cycles as well as more thorough ones depending on the scope of validation. Finally, we show that our solution was able to detect faults which were not detected by more traditional testing techniques.**

I. Introduction

In many situations, real-time systems are deployed in unpredictable environments, where the behavior of the systems is nondeterministic and proving its correctness depends on certain assumptions of the environment which are only available at runtime [1]. In addition, some faults only manifest when the system is deployed in production environments or when the system runs for longer periods of time [2].

Runtime verification (RV) is the process of verifying that certain properties of the system hold in certain states reached during the execution of the specification. It combines both *formal specifications* and *testing* to validate a system. Properties of a given system can be identified by a formal language and automatically translated to a *monitor* [3] which checks their correctness against the implementation. As the verification result, the monitor assigns a verdict that shows whether the implementation satisfies the properties or not. Thus, the runtime verification task deals with the observed executions of the implementation.

Verification techniques such as online testing executes finite test traces within a limited time and may not detect these kinds of anomalies. Furthermore, in some cases the actual real-time system is not accessible or it is expensive to be run every time. Thus, one solution is to monitor previous executions of the systems using the execution logs as replacement of the actual *implementation under test* (IUT). Monitoring is a continuous real-time process of detecting anomalies in the behavior of the systems [4]. An *adapter* is used as an interface between the model-level events and the system-level inputs/outputs.

In this paper, we present a runtime verification approach which uses UPPAAL timed automata (UPTA) [5] to specify the behavior of the IUT and of its environment. The properties of the model are checked via a set of verification rules in UPPAAL. The Distributed TRON (DTRON) [6] online testing tool is used as a monitor to compare the observed behavior of the IUT with the expected input/output traces created from the model. In our case, the behavior of the system is captured from its execution logs due to the fact that we want to validate it in its real environment and, as it will be discussed later, to be able to speed up the validation process using offline logs. In addition, we discuss different implementation challenges and solutions, and how the proposed solution is used for verifying the correct execution of the IUT in its real environment.

The rest of the paper proceeds as follows: Section II briefly discusses the work related to runtime verification and monitoring techniques. Section III briefly introduces UPPAAL timed automata, and the UPPAAL and DTRON tools. In Section IV, an overview of our monitoring approach and tool chain is given. Section V shows how our approach has been applied on a concrete system, while Section VI evaluates the approach and discusses different encountered challenges. Section VII concludes the paper.

II. Related work

There is a large body of work on runtime verification, monitoring and testing for real-time systems. A comprehensive study can be found in [7]. In the following, we only consider those works which have similar approaches to ours.

Chupilko et al. focus on using a method for runtime verification of reactive systems [3]. They generate a monitor with timed automata to verify the correctness of the implementation. The implementation is developed in *hardware description languages* (HDLs), the monitor is implemented in a C++ library and co-executes the specifications as inputs and verifies the expected outputs. The difference between this approach and ours is that their monitor is a program, whereas we use a model to specify and monitor the IUT.

Zhao and Rammig presented a model-based runtime verification method that explores the system model before the current state of system execution is known and so that the property violations are detected before they occur [8]. This pre-checking technique reduces the number of states to be explored. They claimed that the flexibility of monitoring is higher because their method explores before and after a given state, based on information gathered during the testing phase.

Tan et al. proposed a framework for generating test cases and monitoring hybrid systems simulated using the CHARON language for both design-level and implementation-level validation [9]. They provided a model of the monitor as a composition to the main test model. Each system condition is translated as a separate automaton and the automata are synchronized with each other. Our approach does not need to

generate a monitoring automata, since we use the DTRON tool as a monitor.

Salva and Cao discussed an approach to combine two monitoring methods, runtime verification and passive testing (an approach to passively observe the reactions of the IUT to test inputs) to check the conformance of web service compositions in the Cloud [10]. Monitoring models are generated to check whether an implementation conforms to its specification and meets the safety properties. They used ioSTS (input output Symbolic Transition System) models to automatically generate a monitor. The work dealt with deterministic specifications and thus it is not feasible for random behavior of real-time systems, which is needed in our case.

The most similar work was done by Larsen et al. They proposed a testing framework based on the TRON tool for online black-box testing of real-time embedded systems [11]. They showed that the tool is very efficient in terms of error detection and execution performance. The difference to our work is that their approach is targeted at online model-based testing, whereas we target runtime monitoring using log files.

III. Background

In this section we briefly explain the basic concepts of timed automata, conformance monitoring tools and frameworks.

A. UPPAAL Timed Automata

UPPAAL is a model-checker tool for modeling, simulation and verification of real-time systems using an extended version of timed automata called UPPAAL timed automata (UPTA) [5]. An UPTA model is a network of timed automata with *locations*, *edges*, synchronization *channels* (denoted as ! for emitting and ? for receiving synchronizations, respectively), *integer*, *boolean*, and *clock* variables. Edges can be constrained by predicates (over clocks or variables) known as *guards*, which define when the corresponding edge can be enabled. A location can be restricted over clock invariants, which specify how long the system can stay in a given location. On the edges, variables can be updated to new values, whereas clocks may be reset. If there is more than one enabled edge at a time, then one of them will be randomly selected. This means that UPPAAL supports non-deterministic modeling, which gives more freedom to design systems, especially in systems with random discrete events [12].

An UPTA model is composed of one or more *processes* (automata), which communicate via channel synchronizations and shared variables. Each process can have local clocks and variables, as well as access to shared variables. Processes are defined as *templates* from which they are instantiated.

B. DTRON

UPPAAL TRON is an input/output conformance testing tool for testing real-time systems [11]. A UPTA model typically consists of a system partition and of an environment partition. TRON utilizes the environment model to generate test inputs via randomized choice of input. It sends test inputs to an *adapter*, which is an interface between TRON and the IUT.

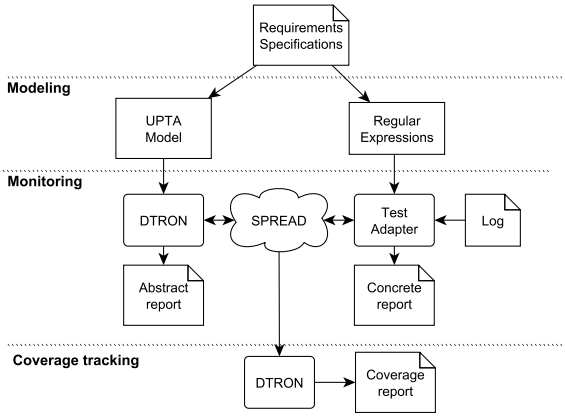

Fig. 1. Monitoring approach

The adapter receives the inputs and converts them to a format that is compatible with the IUT. It also transforms the outputs of the IUT to model-level output actions. Thus, the I/O conformance of the behavior of the IUT is observed by TRON. The feasibility of applying TRON for testing is shown in some practical case studies [13]–[15].

Distributed TRON (DTRON) [6] is an extension tool based on TRON which is capable of simulating and monitoring timed systems in a distributed fashion. This means that several DTRON instances and IUT adapters can be used in the same testing configuration. DTRON and its adapter are loosely coupled using a multicast network, called Spread [16], to exchange abstract messages with the adapter(s) or other DTRON instances. The observable channel synchronizations in a UPTA model are prefixed by $o\_$ and $i\_$, depending if they correspond to outputs or to inputs of the IUT.

IV. Monitoring Approach

In our approach, different artifacts are derived from the given requirement specifications of the IUT and are used to monitor its behavior as illustrated in Figure 1. The approach is divided into two main stages:

Modeling: The system is specified as an UPTA model from the given requirements and specification documents. The model is partitioned into system and environment. Each partition can be composed of several processes.

Monitoring: We use the DTRON tool to monitor the functionality of the IUT against the UPTA model at runtime. The tool connects to the IUT via the Spread network and the adapter, and observes the output messages send by the adapter. We use a set of *mappings* to associate model fragments stemming from requirements with entries of the log file. The information is extracted using *regular expressions*. The result of mapping will be read by the adapter and converted into output messages sent to the UPTA model. Whenever a message is received by DTRON a synchronization of the prefixed channels occurs.

During a monitoring session, the DTRON tool produces an *abstract report*. In case DTRON detects unexpected behavior,

the report elaborates the reasons for the occurrence of the anomaly, and details the current state of the model. Further, the adapter generates a *concrete report* which provides more detailed information about the monitoring session than abstract report, by providing information about the output messages and their originating log entries. Besides, the report may display the values of different parameters or computed values based on the information from the log, which is otherwise not available at model level. A final verdict of the monitoring session is expressed at the end of the report.

Coverage tracking: In order to be able to provide information on which parts of the model have been covered by a given monitoring session or which acceptance criteria have been validated, the coverage level is reported live via the *Coverage Tracker* tool.

V. The Temperature Control System

We applied the proposed approach to a real-time *Temperature Control System* (TCS), which controls the temperature of a host device. The host device contains several sensors (between 20 and 40) and fans, both being installed in different physical locations. The TCS has access to the values of the sensors and can control the speed of the fans via a *Hardware Abstraction Layer* (HAL). As shown in Figure 2, the TCS consists of two components: *Temperature Monitor* (TM), for reading the temperature values, and *Fan Speed Calculator* (FSC), for calculating the target fan speed. The main task of TCS is to maintain the temperature of the host within certain bounds based on a predefined temperature profile.

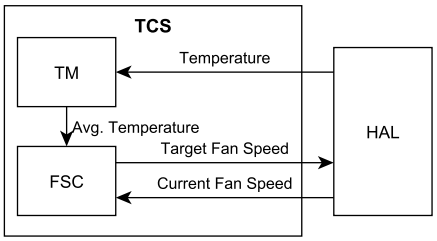

Fig. 2. The overview of Temperature Control System

The startup procedure of the TCS has two phases. First, the available temperature sensors and fans are detected and registered. Subsequently, TCS starts listening for temperature and fan speed readings from the HAL at regular time intervals, computes the target fan speed and communicates it back to HAL. The sensors are organized into three groups, each group having its own sensor polling time interval. Similarly, fans are also organized in a group and are controlled at once.

Whenever a temperature update for any sensor group in the system is received, the average temperature of the host system (i.e., avg_temp) is calculated as the average value of all sensors in the system, after performing error compensation and eliminating faulty values (i.e., with high deviation) from the pool of temperature values. Similarly, FSC computes regularly the target fan speed value (*target_speed*) to be sent to HAL for updating the fan speed. The value is based on the avg_temp, the current fan speed, previously computed target fan speed and the selected temperature control profile.

The host system of TCS continuously provides a detailed log of the status and message exchange of all its components, including the communication between TCS and HAL. The log file can be either prerecorded and downloaded from the host system, or it can be fetched on-the-fly during monitoring. In the latter case, a dedicated tool is used to regularly stream the log from the host system to the computer running the monitoring tool chain.

A. Modeling

The UPTA model of TCS is created from requirements and associated specification documents. TM and FSC are modeled as IUT, whereas HAL is modeled as environment. In our example, the sensor groups have identical behavior, except they have different polling intervals and number of sensors. The UPTA model comprises five UPTA processes: three for different temperature sensor groups, one for a fan group and one for the environment. However, since the processes of all temperature sensor groups are virtually identical, we show only one of them.

All observable channel synchronizations are labeled with $o\_$ prefix, specifying that all the channels in question are output from the IUT. The direction of the communication between the IUT and its environment is differentiated based on the direction of the channel synchronization: the *!* symbol and the *?* symbol for sending and receiving from the environment.For example, when the IUT sends a request to the environment process, the synchronization is modeled as $o\_S1\_req!$ in the IUT model; when a response is received from the environment by a IUT process it is modeled as a receiving channel synchronization $o\_S1\_res?$. For each sensor and fan group, we declare a boolean variable which shows whether the corresponding group is registered and, respectively, polling. These variables are used in defining reachability verification rules.

In our modeling process, we had to take several design decisions, related to the complexity of the UPTA model.

Decision 1: Defer complex mathematical computations to the adapter. Ideally, the calculation of the average temperature and of the target fan speed should be done in the UPTA model. However, due to the following factors, we decided to defer it to the adapter:

- the calculation of average temperature and target fan speed require floating-point arithmetic operations which are not supported by UPPAAL;
- due to the large number of variables involved in the computation and to the fact that even if bounded their values span large intervals, the computation easily results in a state space explosion in UPPAAL which manifest in the test system running out of memory.

Decision 2: Model all temperature indications in a group with only one channel synchronization. At the implementation level, HAL communicates the temperature readings of each sensor in a given group as distinct messages

in a non-deterministic order. Modeling a similar behavior in UPTA will increase the model complexity, as it will require one channel synchronization for each sensor in a group. Thus, we model the temperature readings of a sensor group as one single message. We defer to the adapter the task of collecting and validating temperature and fan speed readings.

1) Temperature Sensor process: The process for the temperature sensor group 1 is shown in Figure 3. Initially, TCS sends the request to initiate the registration ($o\_S1\_req!$) of sensor group 1 and waits for a response message ($o\_S1\_res?$) from the environment (HAL). The response message will always be accompanied by a preliminary temperature indication (o_S1_ind) for each sensor in the group.

Following a successful registration ($S1\_reg = true$), the sensor will enter in the polling mode ($S1Poll = true$) where the temperature of the sensor group will be indicated every polling interval ($S1\_poll\_Int$). The average temperature will be calculated in the adapter and not sent to the tester any longer, as it will be discussed later on.

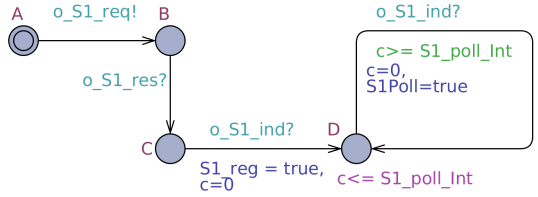

Fig. 3. TM model for sensor group *S1*

2) Fan process: The behavior of the fans (Figure 4) is similar to the one of the sensors: on startup, the maximum fan speed is requested from HAL ($o\_fan1\_max\_speed\_req!$) followed by a confirmation response ($o\_fan1\_max\_speed\_res?$). Subsequently, FCS checks if it is able to control the fans by setting their speed to a specific value ($o\_fan1\_speed\_req!$) and waits for a confirmation that the fan speed request has been received ($o\_fan1\_speed\_res!$). Immediately after, the HAL reports the current fan speed for the group ($o\_fan1\_speed\_ind?$). When the procedure is completed, the fan registration is considered complete ($FANreg = true$) and the fan enters the polling mode ($FANPoll = true$).

Further, FSC calculates the target fan speed every polling cycle based on the previous target fan speed, the current fan speed, and the current average temperature. The reason for this is that a fan cannot always accelerate or decelerate its speed to the new target speed instantly or within one polling cycle, but it requires a certain amount of time or several polling cycles to achieve the new target speed. Similarly in the Fan process, every time the fan speed indication is received, the target fan speed is calculated (in our case in the adapter) and communicated to HAL via the ($o\_fan1\_speed\_req!$). This cycle repeats every polling interval.

3) Environment model: The environment model (Figure 5) is a canonical model containing all the counterparts of the channel synchronizations in the TCS processes. Any synchronization can occur at any moment, if the state of the UPTA model allows it.

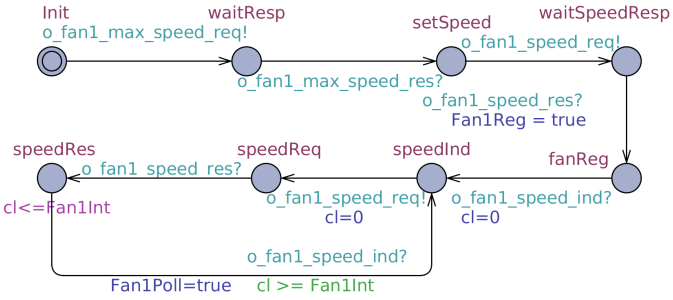

Fig. 4. FSC model: fan speed controller

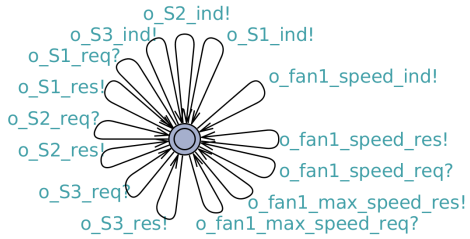

Fig. 5. Environment model

When generating traces from UPPAAL models with tools like TRON and DTRON, the environment model is usually the one driving the trace generation. In our case, since we only observe the behaviour of the IUT, the execution of the UPTA model will be driven by the output messages sent by the adapter according to the log file.

B. Monitoring

Monitoring of the TCS has three goals: a) verify that different messages in the system are delivered in the proper order and that they satisfy the real-time constraints, b) check that the TCS is able to keep the host system within the specified temperature limits and, c) check that TCS is able to control the fans according to the specified temperature profile.

In our test setup, the first goal is addressed by the DTRON tool which verifies that the order of the messages and their timings conform to the one specified in the UPTA model. Due to the limitations of UPPAAL, discussed previously, the other two tasks are accomplished by the adapter (see Figure 1). Thus, the adapter accomplishes three functionalities:

- it parses the log file, selects the relevant log entries based on a regular expression, and distributes the information as abstract messages on the Spread network. In this case, the verdict assignment is deferred to DTRON;
- it collects temperature readings and fan speed values from the selected messages, and computes the target fan speed and the average temperature. In addition, it verifies that the average temperature is in the allowed range and that the actual fan speed does not deviate from the target fan speed beyond a given threshold;

- it generates an on-the-fly report including a verdict of the monitoring session.

The deviation between the calculated target fan speed and the actual target fan speed is calculated as follows:

$$Deviation = \frac{Current\_Fan\_Speed - Target\_Fan\_Speed}{Target\_Fan\_Speed} \times 100$$

Whenever this deviation exceeds a certain threshold (e.g., 10%) it means that the TCS is not able to cope with the requested target speed and it should be signalled as a failure. There are two possible reasons for this kind of failure: 1) hardware failure (e.g., sensors, fans, or both are malfunctioning) and 2) system temperature increased very steeply such that the TCS could not cool down the system within acceptable time period.

C. Coverage tracking

As mentioned in the introduction, monitoring is a passive form of testing. It cannot control the behavior of the IUT via test inputs; it can only observe its behavior. However, in both cases it is important to be able to recognize the coverage level of the model, as a quantitative measure of *how much and which part of the system has been tested/monitored during a session*.

With respect to the TCS model, we are interested in two kinds of coverage: *acceptance criteria coverage* (how many of the coverage criteria have been validated out of the total) and *edge coverage* (how much of the UTPA model has been visited). Thus, we annotate the model with additional tracking (aka *trap*) variables, which do not affect the original behavior of the UPTA model. Thus, we used two types of tracking variables: edge variables (added automatically to each edge) and requirements variables (added manually based on domain knowledge). The approach has been discussed in [17].

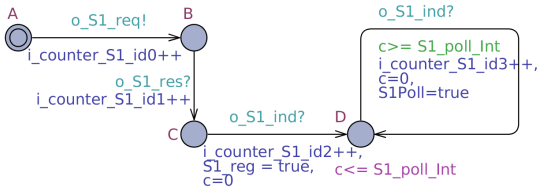

Fig. 6. Sensor 1 group model with tracking variables

Figure 6 gives an example on how tracking variables have been used. The *i_counter_id* variables are used for edge coverage, whereas variables *S1Poll* and *S1_reg* monitor that different requirements are met.

A separate UPTA process (Figure 7) is used to compute which acceptance criteria (grouping several requirements) are validated. For instance, when all sensor groups are registered ($S1Poll$ AND $S2Poll$ AND $S3Poll$ is True) it means that acceptance criteria *"All sensor groups should start reporting values after they are registered"* is satisfied, which is denoted by setting the $i\_counter\_us\_1\_ac\_2$ variable to *True*.

A *Coverage Tracker Tool* [17], takes advantage of the distributed nature of DTRON by connecting to the Spread network and receiving periodically from DTRON (based on

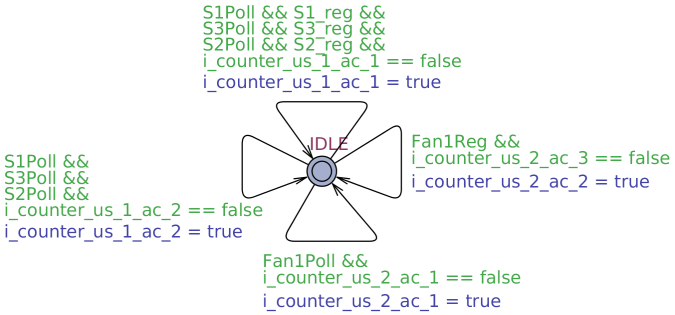

Fig. 7. Model fragment observing acceptance criteria

customizable time interval) the values of the tracking variables from the model. After each update the tool updates statistics for all tracking variables based on the selected coverage criteria and includes them in a *Coverage Report*. A sample report is shown in Listing 1, where we have a total of four acceptance criteria. The coverage level for acceptance criteria is 50% since only 2 out of 4 acceptance criteria have been validated. However, the edge coverage level is 100% for Sensor 1 group process because all the edges in the process were traversed at least once. Due to space reasons, we removed the other information from the report.

Listing 1. Coverage Report

```
***                                                              ***
Test started:    04-06-2014 15:03:42
Report output time:      04-06-2014 15:04:22
Tester id:       0

Summary Table:
Edge Coverage:                                   100.0 %
Acceptance criteria Coverage:                    50.00 %


Details

Coverage Details:
Total Edge Coverage                              100.0 %
**********************************************
Template: S1
        Requirement: S1_id0                      1
        Requirement: S1_id1                      1
        Requirement: S1_id2                      1
        Requirement: S1_id3                      1
Edge Coverage:           100.0 %

Total Acceptance criteria Coverage               50.00 %
**********************************************
Template: All templates
        Requirement: us_1_ac_1                   0
        Requirement: us_1_ac_2                   0
        Requirement: us_2_ac_1                   1
        Requirement: us_2_ac_2                   1
```

In the case of the TCS system, the coverage level of the UTPA model reaches 100% rather quick, i.e., after all sensors and fans have registered and the sensors passed through the polling cycle at least once.

VI. Evaluation and discussion

In order to evaluate our approach, we performed two types of validation: manual mutation of a sample log file and the use of prerecorded production logs.

A. Mutation testing

For mutation testing purposes, we selected a log file including events of the TCS activity for around three hours and containing roughly 33,000 entries. In this log, we manually applied 1-level mutations (one mutation at a time). We used the following operands: delete an entry, duplicate an entry, change the position of an entry, modify parameter values and delete parameter values inside an entry. In total, we operated 25 mutations. In 92% of the cases, the mutants were detected and *killed*. Two of the mutants were left *alive*. The first one was swapping the entries reporting values of different sensors within the same polling interval, which basically had no impact neither on the message sequence in DTRON nor on the calculated average values. A second mutant left alive was changing the value of a temperature reading of one sensor. However, since the value was an outlier value compared to the other sensor values in the same group, it was removed by the algorithm of computing the average temperature and thus, it did not have an impact on the average temperature.

B. Production logs

In the second validation experiment, we analysed several prerecorded production logs with length up to five hours. Two anomalies have been detected.

- after four hours, it was detected that several sensors did not report their readings in the specified time interval;
- the fan speed was slowly deviating from the target value with the passage of time. After thorough investigation, we concluded that the specification document of the IUT had an omission in the description of how the target fan speed calculation.

These findings are in line with the observations made in [2], which argues that monitoring can detect failures that otherwise remain undetected by traditional testing techniques, especially when failures can occur at random times and are triggered by non-deterministic environments.

C. Benefits and limitations

Our approach divides the verification tasks between DTRON and the adapter. DTRON is used for runtime verification of the sequence and real-time constraints, whereas the adapter validates complex numerical computation which is not properly supported by DTRON and UPPPAL. We do not see this as a major drawback since we take advantage on the one hand, of the real-time verification capabilities of DTRON, and on the other hand, of the flexibly of a Python adapter in computing different conditions and in customizing the monitoring report.

The state space explosion was one of the main problems when trying include the computations in the model. This was due not only to the large number of clock (5) and integer (20) variables in the model, but also to their large ranges, e.g. [-50, 200] for sensors or [0-300] for fans. With the complete set of variables in the model, the state space explosion will manifest rather quickly and the tool would run out of memory.

After deferring the calculations to the adapter, the symbolic state space of the model became manageable. For instance, the UPTA model with three sensor processes, one fan process, and the environment model resulted in 1005 symbolic states and very limited memory consumption. The maximum number of symbolic state was relatively smaller when using DTRON, since the later only evaluates the next immediate symbolic states that the model can transition to from the current state, instead of evaluating the entire state space. This, in theory, allows for even more scalability.

Our approach gives one the possibility to track the coverage level of the model with respect to acceptance criteria and model structure. However, tracking the coverage level does not come for free, as introducing tracking variables and additional processes (and clocks) in the UPTA model has an impact on the state space.

One drawback of monitoring a system in real-time is that the length of the monitoring session can become quite long in order to observe relevant system reactions. The same would apply in using prerecorded logs: the duration of the monitoring session will be equal with the duration of the log file. This can be an impediment in the current time-constrained continuous integration and continuous deployment processes, as the validation of the system via monitoring would provide feed-back too slow. To address this problem several optimizations were performed.

1) Targeted monitoring: In on-line mode, we have scoped the process in two parts: smoke monitoring and long-term monitoring. The smoke monitoring is performed after each integration cycle, when the latest version of the IUT is started and monitored in a controlled execution environment. The session typically lasts for 10 minutes during which the detection and registration of the sensors and fans are detected, they are registered and several sensor readings are received at specified intervals. This smoke session is sufficient to validate the acceptance criteria specified in Figure 7 and to have all the edges covered in the model. Long-term monitoring is performed for periods ranging from one hour to several days with the purpose of validating if the system remains stable and it is able to regulate the temperature according to its specification. This implies checking that sensor readings are still received at specified intervals and the fan speed calculations do not deviate more than a certain value.

In case of prerecorded logs, the adapter was implemented so that at the start of the monitoring process it takes as parameter a time interval for which the monitoring should be performed. During the monitoring session the adapter skips all log entries outside the provided interval. This approach allows us to focus the monitoring on certain interesting parts of the log, for instance when the temperature of the environment increased/decreased rapidly or when certain relevant temperatures have been reached.

2) Time scaling: Another approach to reduce the time needed for monitoring prerecorded logs is to speed up the entire monitoring framework with a *scaling factor*, as follows:
• at adapter level - the adapter extracts messages from the log as they arrive, enqueues and distributes them to DTRON based on their timestamp, with microsecond accuracy. We use the scaling factor to reduce the time interval between sending the messages to DTRON.
• at DTRON level - to start a monitoring session with DTRON, one has to provide a mandatory parameter called *time unit* which specifies the duration of a clock tick in the UPTA model, expressed in microseconds. For instance, a value of 1 000 000 specifies that a clock tick in the UPTA model takes one second. We use the scaling factor the divide the time unit as well. For a scaling factor of 10, the clock tick will take 0.1 seconds in the UPTA model.

This approach allowed us to reduce the monitoring time with the value of the scaling factor. For instance, with a scaling factor of 10, we were able to process a five hour log file in around 30 minutes and detect the same anomalies that were discussed previously. One should note though, that the maximum value of the scaling factor strongly depends of the time precision of the model, of the log file, and of the real-time capabilities of the operating system running the monitoring tool chain.

VII. Conclusions

We have presented an application of the UPPAAL and DTRON tools to monitoring a real-time system in a practical setting. The behavior of the implementation under test is specified using UPPAAL timed automata and DTRON is used as a monitor to verify that the execution traces of the system conform to its UPTA model. Our approach allows us not only to check the correct real-time execution of the IUT, but also to validate it with respect to different computed values.

An adapter is used to interface with the IUT, which in our case is a execution log file. The adapter also takes over parts of the monitoring tasks related to complex numerical computations, which are not supported well by UPPAAL and related tools.

The presented approach allows one to track system requirements and acceptance criteria to the UTPA model and to follow their coverage level. Also, traditional structural coverage criteria, such as edge coverage, are used to report on the model coverage level achieved by a given log.

One limitation in the modeling process was the scalability, especially when having numerical computations. However, the problem has been solved by delegating a part of the functionality to the adapter.

The practical evaluation showed that our method is able to detect errors that were not otherwise detected by traditional testing techniques. We also showed how different optimizations of the approach were performed in order to shorten the feed-back cycles or to focus the monitoring to relevant parts of the log.

References

[1] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[2] K. Okumoto and A. L. Goel, "Optimum release time for software systems based on reliability and cost criteria," *Journal of Systems and Software*, vol. 1, no. 0, pp. 315 – 318, 1980. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0164121279900335

[3] M. M. Chupilko and A. S. Kamkin, "Runtime verification based on executable models: On-the-fly matching of timed traces," in *MBT*, ser. EPTCS, A. K. Petrenko and H. Schlingloff, Eds., vol. 111, 2013, pp. 67–81. [Online]. Available: http://dblp.uni-trier.de/db/series/eptcs/eptcs111.html#abs-1303-1010

[4] D. Dvorak and B. Kuipers, "Model-based monitoring of dynamic systems," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'89. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 1238–1243. [Online]. Available: http://dl.acm.org/citation.cfm?id=1623891.1623953

[5] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *Int. Journal on Software Tools for Technology Transfer*, vol. 1, pp. 134–152, 1997.

[6] A. Anier, J. Vain, and E. Halling, "Provably correct test generation for online testing of timed systems," in *The 11th International Baltic Conference on DB and IS*. IOC-Press, May 2014.

[7] N. Delgado, A. Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859–872, 2004.

[8] Y. Zhao and F. Rammig, "Model-based runtime verification framework," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 1, pp. 179 – 193, 2009, proceedings of the Sixth International Workshop on Formal Engineering approches to Software Components and Architectures (FESCA 2009). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066109003892

[9] L. Tan, J. Kim, O. Sokolsky, and I. Lee, "Model-based testing and monitoring for hybrid embedded systems," in *Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on*. IEEE, 2004, pp. 487–492.

[10] S. Salva and T.-D. Cao, "A model-based testing approach combining passive conformance testing and runtime verification: Application to web service compositions deployed in clouds," in *Software Engineering Research, Management and Applications*, ser. Studies in Computational Intelligence, R. Lee, Ed. Springer International Publishing, 2014, vol. 496, pp. 99–116. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-00948-3_7

[11] K. G. Larsen *et al.*, "testing real-time embedded software using uppaal-tron: An industrial case study."

[12] A. Hessel *et al.*, "Testing real-time systems using uppaal," in *Formal Methods and Testing*, R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 77–117. [Online]. Available: http://dl.acm.org/citation.cfm?id=1806209.1806212

[13] A. Anier and J. Vain, "Timed automata based provably correct robot control," in *Electronics Conference (BEC), 2010 12th Biennial Baltic*, Oct 2010, pp. 201–204.

[14] L. de Assis Barbosa *et al.*, "On the automatic generation of timed automata models from isa 5.2 diagrams," in *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, Sept 2007, pp. 406–412.

[15] C. Rutz and J. Schmaltz, "An experience report on an industrial case-study about timed model-based testing with uppaal-tron," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, March 2011, pp. 39–46.

[16] The Spread Toolkit. [Online]. Available: http://www.spread.org/

[17] M. Koskinen *et al.*, "Combining model-based testing and continuous integration," in *ICSEA 2013, The Eighth International Conference on Software Engineering Advances*, 2013, pp. 65–71.