# State Space Reconstruction for On-Line Model Checking with UPPAAL

Jonas Rinast, Sibylle Schupp
Institute of Software Systems
Hamburg University of Technology, Hamburg, Germany
Email: {jonas.rinast,schupp}@tuhh.de

Dieter Gollmann
Security in Distributed Applications
Hamburg University of Technology, Hamburg, Germany
Email: diego@tuhh.de

***Abstract*—On-line system verification requires the efficient reconstruction of the state space a model checker generates. This paper proposes an approach to reconstruct the current state of models of real-time systems, implements it in the Uppsala and Aalborg model checker (UPPAAL) and thus renders on-line model checking in UPPAAL possible. On-line model-checking can be employed if parameters of models need to be adjusted to real-world values in case models are inaccurate. Applications include closed-loop patient monitoring and care taking as patient models commonly fail to accurately model all interactions in the human body and thus cannot provide good long-term estimates to ensure the patient's safety. We exploit use-definition chains in state space transformations to reduce the amount of reconstruction transformations. During testing the method reduced the amount of transformations by 42% on average over all experiments.**

***Keywords*—*State Space Reconstruction; On-line Model Checking; UPPAAL***

## I. Introduction

Medical treatment facilities have grown to rely significantly on medical devices for monitoring and treatment. Most devices are still operated manually today and need to be configured, maintained, and supervised by a care taker. Recently, closed-loop monitoring and treatment of patients became a research topic as experience shows that human errors are prevalent. Patient-in-the-loop systems try to autonomously assess the patient's state using a monitoring device and if necessary treat the patient automatically, e.g., via a remote infusion pump. Such a system must clearly be shown to cause no harm to the patient. Safety must be ensured to prevent harm from the patient not only during normal operation but also in case emergency situations arise.

Model checking is a well developed technique to verify that a system model conforms to its specification and thus may be applied to show the safety of such system. However, to make meaningful conclusions about the system's behavior it is necessary to have detailed and accurate models of the individual components of the system. In the medical domain, the model checking approach is therefore severely hampered if the patient needs to be modeled accurately, e.g., to make estimates on a drug concentration in the patient. Generally, a patient model is likely to be inaccurate as the physiology of human beings is complex and varies between individuals, e.g. blood oxygen and heart rate depend on the patients condition. A generalized model will always miss individual characteristics. Patient-in-the-loop systems thus could be proved safe with such models but might still put patients at risk.

On-line model checking is a recent model-checking variant that relaxes the need for models to be accurate far into the future. On-line model checking provides safety assurances for short time frames only and renews these assurances continuously during operation. Appropriate models for the system thus only need to be correct for the short time frame they are used in. The renewal of safety assurances then is carried out on models adapted to the current system state to ensure the system's safety for the next time window. This on-line approach thus allows safety assessment at all times and provides means to react before safety violations occur.

A model adaptation step first needs to create an initialization sequence that recreates the previous model state before adjusting single values. The reconstruction is necessary to allow the simulation of the model to continue from the state it was interrupted in. This paper presents an automated state reconstruction approach for the Uppsala and Aalborg model checker (UPPAAL) that eliminates the need for custom reconstruction procedures for every application. The developed reconstruction method serves as a base for an on-line model checking interface with UPPAAL as the underlying verification engine.

Naively, the state space can be reconstructed by executing the same transition sequence that was used to create the state in the beginning. However, if the simulation has already run a significant time the executed transition sequence is likely to be long and only continues to grow over time. A more direct way to the desired state space is needed to keep the reconstruction process fast and on-line model checking feasible. For our reconstruction approach we adopted use-definition chains to eliminate transformations that have no effect on the final state space. Such transformations occur when their results are overwritten before they are read. Our reconstruction method has been applied to seven different test models. The method always correctly reconstructed the original state space while yielding a reduction of the executed transformations in the range from 23% to 84%.

Interestingly our on-line model checking interface could not only be used to automatically carry out on-line model checking. The interface also allows generic dynamic adaptation of model parameters and thus could be used with parameter learning algorithms or for calibrating the model.

The rest of the paper is organized as follows: Section II shortly introduces model checking, on-line model checking, and the model checker UPPAAL. Section III first provides necessary information on UPPAAL's state space and its transformations and then explains our reconstruction approach. Section IV presents our evaluation results. Section V gives an overview on related literature and, lastly, Section VI summarizes the paper and suggests further research.

## II. On-Line Model Checking

This section shortly introduces model checking and its on-line variant, on-line model checking. The technique is shown by way of example using the model checker UPPAAL; for a formal specification of UPPAAL see [1].

Generally, the model checking approach explores the state space of the given system model in a symbolic fashion to check whether the state space satisfies certain properties. Such properties are mostly derived from a requirement specification for the system, e.g., one could check whether or not a certain system state is actually reachable. The modeling and property languages vary greatly depending on the model-checking tool. Tools for various programming languages coexist with dedicated tools that support their own modeling language. Dedicated tools often use finite state automata as a base formalism for their models. UPPAAL is such a well-established, dedicated model checking tool, which was jointly developed by Uppsala University, Sweden, and Aalborg University, Denmark [2], [3]. It is based on the formalism of timed automata: an extension of finite state automata with clock variables to allow modeling of time constraints. A finite state automaton defines a transition system by defining locations and edges that connect these locations. Edges are fired to execute a transition from one location to another. The system state in this case is the current location of the automaton and the possible valuations of the clock variables.

Figure 1 shows the example model that will be used to demonstrate the proposed state space reconstruction method. The model consists of three locations, *Init*, *Inv*, and *Count*, where *Init* is the initial location indicated by the double circle. The model uses two variables: $x$, a clock variable, and $c$, a bounded integer variable. Clock variables are special variables that synchronously advance indefinitely unless they are bounded by one or more invariants on the current locations. The location *Inv* has such an invariant, `x <= 2`, to bound the clock $x$, thus the value of $x$ in *Inv* can be any value between its value when it entered the location and 2. The model has a single transition from the initial location to *Inv*. This transition is annotated with a guard, `x >= 3`, and an update, `x = 0, c = 0`. Guards are used to enable and disable edges depending on the current state. Here, the clock $x$ needs to be greater or equal to 3 for the edge to be enabled. Only then can it be fired and a transition occurs. Indeed, as there is no invariant on $x$ on the initial location the edge is enabled for values greater or equal to 3. Upon firing of the edge the update is executed: the clock $x$ and the bounded integer $c$ are both reset to 0. The edge from *Count* to *Inv* is nearly identical to the previous edge: when $x$ is greater or equal to 3 the edge may be fired but $x$ is reset to 1 instead of 0 and $c$ is not modified. As a consequence, the value of $x$ in *Inv* is between 0 and 2 when the location is first entered and between 1 and 2 on every subsequent visit. The transition between *Init* and *Count* has no guard and shows that an update may consist of a complex expression: the update `c = (c + 1) % 7` increases $c$ by 1 modulo 7.

As explained in the introduction, model checking relies on accurate long term models. On-line model checking is a variant of classic model-checking that eliminates the need for such models and thus may be applied when such models are unavailable. It reduces the modeling error by periodically

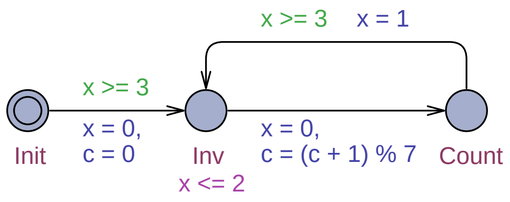

Figure 1. UPPAAL Model Example

adjusting the current state to the observed real state, e.g., by setting a model value to the exact value measured by a sensor attached to a patient. For example, if we consider the model in Figure 1 one could assume that the counter variable $c$ is modeling some patient's parameter. If that parameter in reality occasionally jumps the model is inaccurate and needs to be adjusted by setting $c$ to the correct value. On-line model checking performs the adjustments and thus the jumps do not need to be modeled accurately. Note that errors may still be present in the system under on-line model checking but the method predicts them in advance to react to them. On-line model checking requires the model analysis to finish before the next update interval. Though the main work is done by the model checker the reconstruction still consumes some time, which our method reduces compared to the naive automatic reconstruction approach.

## III. State Space Reconstruction

In this section, we summarize the required information on UPPAAL's state space in Subsection III-A. Then, Subsection III-B presents our state space reconstruction process.

### A. *UPPAAL's State Space and its Transformations*

UPPAAL's state space can be divided into three parts: the time state, the location state, and the data state. The location and the data state are straightforward: every data variable is assigned exactly one value for the data state and the location state consists of the current location vector, i.e., a vector that contains the current location for every automaton instance. The time state is more complicated as it needs to capture all possible valuations for every clock in the model as well as all relations between the clocks. *Difference bound matrices* (DBM) are a common and simple representation method for such time states [1], [4]. By introducing a static zero clock in addition to all the clocks in the model ($\mathcal{C}_0 = \mathcal{C} \cup \mathbf{0}$, $\mathcal{C}$ the set of all clocks) all necessary clock constraints can be written in the form $x - y \preceq n$ where $x$ and $y$ are clocks ($x, y \in \mathcal{C}_0$), $\preceq$ is a comparator ($\preceq \in \{<, \leq\}$), and $n$ is an integer ($n \in \mathbb{Z}$). A value in a difference bound matrix then is a tuple of an integer and a comparator $(n, \preceq)$, $n \in \mathbb{Z}$, $\preceq \in \{<, \leq\}$ or the special symbol $\infty$, which indicates no bound. An order on the entries is given by $(n, \preceq) < \infty$, $(n_1, \preceq_1) < (n_2, \preceq_2)$ if $n_1 < n_2$, and $(n, <) < (n, \leq)$. Addition is defined as follows: $(n, \preceq) + \infty = \infty$, $(m, \leq) + (n, \leq) = (m + n, \leq)$, and $(m, <) + (n, \preceq) = (m + n, <)$. A difference bound matrix thus contains one bound, either including or excluding, for every pair of clocks $\mathbf{M} = (\{\mathbb{Z} \times \{<, \leq\}\} \cup \infty)^{|\mathcal{C}_0| \times |\mathcal{C}_0|}$.

As an example a clock constraint system with two clocks $a$ and $b$ and the constraints $a \in [2, 4)$, $b > 5$, and $b - a \geq 3$ is transformed to the canonical constraints $a - \mathbf{0} < 4$, $\mathbf{0} - a \leq -2$,

$b - \mathbf{0} < \infty$, $\mathbf{0} - b < -5$, $a - b \leq -3$, and $b - a < \infty$. The matching DBM is

$$\begin{array}{c|c|cc|} & \mathbf{0} & a & b \\ \mathbf{0} & 0 & (-2, \leq) & (-5, <) \\ \hline a & (4, <) & 0 & (-3, \leq) \\ b & \infty & \infty & 0 \end{array}$$

During simulation of an UPPAAL model its transitions are repeatedly executed. Every transition generally has multiple effects on the time state and each such effect corresponds to a transformation of the difference bound matrix that represents the current time state. The following summary lists the DBM transformations necessary to traverse the state space [4]:

- *Clock Reset* A clock reset is performed when an edge is fired that has an update for a clock variable (`x = n`). A clock reset sets the upper and lower bound on the clock $x$ to the given value and depending constraints, i.e., constraints on a clock difference involving $x$ are adjusted. This corresponds to modifying the matrix row and column for the clock $x$.
- *Constraint Introduction* A constraint introduction is performed if either a firing edge has a guard on a clock or an invariant on a clock is present in a current location and the bound is more restrictive than the current constraint on the involved clock. In that case the relevant matrix entry is set to the new constraint and for all other entries in the matrix it is checked whether the new bound induces stricter bounds.
- *Bound Elimination* Bound elimination is performed when a new location is entered. All bounds on clock constraints of the form $c - \mathbf{0} < n$ are removed, i.e., the upper bounds on clocks are removed. Bound elimination is equivalent to setting the first matrix column except the top-most value to $\infty$.
- *Intersection* An intersection is performed if a state is constrained by multiple constraints. In that case all constraints are applied individually and their results are intersected to obtain the final result. Intersecting multiple DBMs is achieved by finding the minimum value for every matrix entry from all intersecting matrices.
- *Urgency Introduction* An urgency introduction is performed if an urgent or committed location is entered or an entered location has an outgoing, enabled transition that synchronizes on an urgent channel. Unlike the previous transformations, urgency is a modeling construct specific for UPPAAL to prevent time from passing. An urgency introduction is semantically equivalent to introducing a fresh clock on the incoming edge and adding a new invariant on that clock with a bound of 0 to the location. An urgency introduction thus can be derived from a clock reset and a constraint introduction.

Returning to the example model (Figure 1) the individual transitions can now be broken down into their respective transformations. The initial location *Init* induces a bound elimination on the initial state where all clocks are set to zero. The transition from *Init* to *Inv* yields a constraint introduction for the guard (`x >= 3`) and a subsequent clock reset (`x = 0, c = 0`). The reset of the bounded integer $c$ is ignored here as $c$ is part of the data state. The location *Inv* results in a bound elimination and a following constraint introduction to accommodate the invariant (`x <= 2`). The transition from *Inv* to *Count* simply induces a single clock reset transformation before the location *Count* eliminates the bound on the state space again. Lastly, the transition from *Count* to *Inv* introduces the same kind of transformations as the transition from *Init* to *Inv*: both perform a constraint introduction and a clock reset. The values computed for the clock variable $x$ are as follows:

1) Location *Init*
   a) Initial: $x = 0$
   b) Bound Elimination: $x \in [0, \infty)$
2) Transition *Init* $\longrightarrow$ *Inv*
   a) Constraint Introduction: $x \in [3, \infty)$
   b) Clock Reset: $x = 0$
3) Location *Inv*
   a) Bound Elimination: $x \in [0, \infty)$
   b) Constraint Introduction: $x \in [0, 2]$
4) Transition *Inv* $\longrightarrow$ *Count*
   a) Clock Reset: $x = 0$
5) Location *Count*
   a) Bound Elimination: $x \in [0, \infty)$
6) Transition *Count* $\longrightarrow$ *Inv*
   a) Constraint Introduction: $x \in [3, \infty)$
   b) Clock Reset: $x = 1$
7) Location *Inv*
   a) Bound Elimination: $x \in [1, \infty)$
   b) Constraint Introduction: $x \in [1, 2]$

## B. *Reconstructing UPPAAL States*

In many models a large number of previous transitions do not have an impact on the current state space. In the example model (Figure 1) this behavior can be observed: in the location *Count* the clock $x$ is in the range $[0, \infty)$. This valuation was completely created by the clock reset of the ingoing edge and the bound elimination of the location itself. Previous state space transformations do not have any influence on the valuation of $x$. Therefore, instead of executing the transition sequence *Init* $\longrightarrow$ *Inv* $\longrightarrow$ *Count* totaling 7 transformations only 3 transformations are required. The introduction of a new initial state and the direct transition to *Count* with an update `x = 0` is sufficient to recreate the state space.

During reconstruction it is thus beneficial to exploit the fact that effects of certain state space transformations are overwritten by subsequent transformations. The key idea of our approach is the construction of use-definition chains to identify transformations that may be removed. A use-definition chain is a data structure that provides information about the origins of variable values: for every use of a variable the chain contains definitions that have influenced the variable and ultimately lead to the current value. Our idea is to adapt the definition-use chain technique from static data flow analysis on a program's source code to the state space reconstruction: every entry in the model's difference bound matrix is treated as variable and thus is observed for uses and modifications. DBM entries are only modified by applying a state space transformation on the DBM. We thus analyzed the read and write access to matrix entries for every transformation to derive the use-definition chains where the transformations are the basic operations.

In the following, our reconstruction approach is presented using the clock reset transformation as a leading example. First, we discuss the derivation of use-definition chains from the transformation. Then we lay out the use of reference counters as a memory structure to identify removable transformations. Lastly, we give a short overview on model synthesis based on the shortened transformation sequence. Altogether the adaptation of the use-definition chain approach to the reconstruction process results in the following steps:

1) *Initialization* Canonize model by introducing general starting points for later synthesis, extract necessary information from the model.
2) *Simulation* Select transitions in the model according to intended behavior, execute and store them. Simultaneously break them down into matching state space transformations and apply them internally to construct the use-definition chains of the transformations. Remove unnecessary transformations on-the-fly using reference counters for the transitions derived from the use-definition chains.
3) *Synthesis* Group the sequence of reduced transformations to form transitions and add the transitions to a newly created model obtained from the original one. Match the last transition to the current location state and update the data state on that transition.

Starting points for the reconstruction algorithm are the algorithms for the original transformations. Figure 2 lists as an example an algorithm for the clock reset transformation on the difference bound matrix $D$ according to Johan Bengtsson [4]. Examination of the algorithm yields that all values in the row and column that are associated with the reset clock are written and all values in the top-most row and left-most column are read: lines 3 and 4 of the algorithm write $D_{ij}$ and $D_{ji}$ and read $D_{i0}$ and $D_{0i}$. Note that index $j$ is always greater than 0 as it is a real clock and not the $\mathbf{0}$-clock. Therefore, the reset transformation creates a use for every value in the top-most row and left-most column and a definition for every value in the row and column for the clock in question. Taking into consideration that the value $D_{jj}$ will always evaluate to zero and no definition needs to be generated we construct a modified algorithm that captures the definitions and uses generated by the transformation. Figure 3 shows the modified algorithm. It has an additional parameter $T$, which is a matrix that contains the transformations that are responsible for the current DBM values. Additionally to the functionality of the original algorithm the new algorithm updates this matrix and creates the necessary definition and use information: in lines 6 and 7 we store that the reset transformation uses the transformations $T_{0i}$ and $T_{i0}$ and lines 8 and 9 update the matrix to show the reset transformation is now responsible for the values $D_{ji}$ and $D_{ij}$.

We designed the transformation matrix data structure for the use-definition chains to allow on-the-fly removal of unnecessary transformations: as soon as a transformation is overwritten in the matrix a following transformation cannot have a dependency on that transformation. Thus, the transformation may be directly removed if no intermediate transformation depends on it. If intermediate transformations exist the transformation can be deleted as soon as those are removed. To accurately track needed transformations the data structure uses reference counters: every transition is assigned a counter to indicate how often it is referenced and every transformation updates this counter. The benefit of the on-the-fly removal is reduced memory usage for the data structure and shorter processing time during transformation execution.

```
1:  procedure RESET($D, x_j = m$)
2:      for $i \leftarrow 0, n$ do
3:          $D_{ji} \leftarrow (m, \leq) + D_{0i}$
4:          $D_{ij} \leftarrow D_{i0} + (-m, \leq)$
5:      end for
6:  end procedure
```

Figure 2. Reset Transformation Algorithm [4]

```
1:  procedure RESET($D, T, x_j = m$)
2:      for $i \leftarrow 0, n$ do
3:          if $i \neq j$ then
4:              $D_{ij} \leftarrow D_{i0} + (-m, \leq)$
5:              $D_{ji} \leftarrow (m, \leq) + D_{0i}$
6:              Use($T_{0i}$)
7:              Use($T_{i0}$)
8:              $T_{ji} \leftarrow$ this
9:              $T_{ij} \leftarrow$ this
10:         end if
11:     end for
12: end procedure
```

Figure 3. Modified Reset Transformation Algorithm

We analyzed all relevant DBM transformations for their reads and writes and adapted the algorithms to update the transformation matrix and the reference counters accordingly. Special attention had to be given to the intersection algorithm: if two transformations are applied to a DBM and only one of them writes a certain value but the previous value is the stronger bound the transformation that did not modify the matrix is responsible for the resulting entry. This behavior needs to be introduced during the intersection algorithm as the original transformation only creates relations for entries it can potentially modify. Also the reference counters have to be propagated accordingly. We encapsulate read-write relations of transformations in special linker objects such that other transformations may influence them later on and the effects of the transformation on the data structure may be deferred to appropriate times to manage the reference counters.

The synthesis of the actual UPPAAL model from the calculated transformation sequence has to take into consideration that UPPAAL allows a single automaton to be instantiated multiple times with possibly different parameters. During initialization of the reconstruction we therefore analyze the model definitions for automaton instantiation and save the relevant parameters. Also, as the location space needs to be correctly reconstructed an automaton that is instantiated multiple times has multiple initializations transitions for every instantiation. We use a single bounded integer variable in conjunction with appropriate guards to correctly order these transitions. Another important aspect of the synthesis step is that the model initialization needs to be self-contained, i.e., the ini-

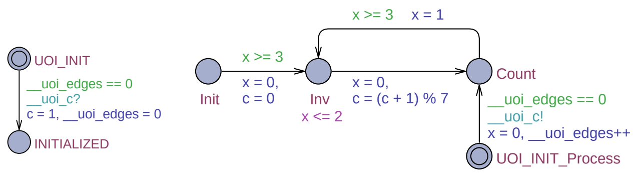

Figure 4. Reconstructed example model

TABLE I. EVALUATION RESULTS

| Model | Transitions | | | Transformations | | |
|---|---|---|---|---|---|---|
| | Before | After | Reduction | Before | After | Reduction |
| 2doors | 100 | 65.89 | 34.1% | 364.7 | 254.46 | 30.2% |
| bridge | 100 | 68.21 | 31.8% | 188.39 | 144.09 | 23.5% |
| train-gate | 100 | 66.18 | 33.8% | 320.09 | 214.17 | 33.1% |
| fischer | 100 | 91.27 | 8.7% | 345.33 | 249.46 | 27.7% |
| csmacd2 | 100 | 100 | 0% | 709.71 | 434.19 | 38.8% |
| csmacd32 | 75.58 | 75.58 | 0% | 1818.6 | 327.49 | 79.7% |
| tdma | 100 | 68.16 | 31.8% | 719.88 | 240.11 | 66.6% |
| 2doors | 1000 | 627.9 | 37.2% | 3722.3 | 2612.9 | 29.8% |
| bridge | 1000 | 641.3 | 35.9% | 1882.8 | 1436.4 | 23.7% |
| train-gate | 1000 | 606.1 | 39.4% | 3200.1 | 2194.1 | 31.4% |
| fischer | 1000 | 853 | 14.7% | 3455.3 | 2486.8 | 28% |
| csmacd2 | 1000 | 1000 | 0% | 7238.1 | 4375.5 | 39.5% |
| csmacd32 | 619.6 | 619.6 | 0% | 22491.1 | 2540.3 | 84% |
| tdma | 1000 | 663.1 | 33.7% | 6446.3 | 2651.5 | 58.9% |

tialization of multiple automata needs to finish synchronously to prevent parts of the model from advancing prematurely. As the initialization transitions per automaton may differ in length we employ a broadcast channel to synchronize the last transition to the original model. We use these final transitions to initialize the data variables as well. In case global variables are present an additional init automaton is introduced for their initialization. Figure 4 shows the reconstruction model (right) for the example model (Figure 1) after 2 transitions. The additional initialization automaton (left) sets the global, bounded integer $c$ to 1. The clock $x$ is set to 0 and the location is correctly initialized to *Count* after an initial first transition. The reconstructed model only needs to execute a single transition in contrast to the original model, which uses two, to reach the correct state. For transformations the reconstructed model uses 3 time state transformations while the original model needs 7.

## IV. EVALUATION

We evaluated our use-definition reconstruction method by applying it to seven different UPPAAL models and comparing it to the naive reconstruction approach. The models *2doors*, *bridge*, *train-gate*, and *fischer* are part of the UPPAAL example model suite. The *csmacd* models and *tdma* were taken from case studies. We ran two test sets for every model. The first test executed 100 times 100 random transitions of the model before reconstructing the state. The second test set executed 1000 random transitions 10 times. For the *csmacd32* model it was not always possible to execute the maximum number of transitions during simulation as the model exhibits deadlock states. Table I shows our evaluation results. In the top half the results of the first test set and in the bottom half the results of the second test set are shown. All values are averages over the respective test runs but their variances are small. In our experiments the reduction of transformations is between 23% and 84% while the reduction of transitions is between 0% and 39.4%. This difference mainly stems from the fact that to delete a single transition all induced transformations need to be removed. However, our model synthesis algorithm still is unoptimized and sometimes produces unnecessary transitions. In cases where the transition reduction is higher than the transformation reduction the removal of transformations made it possible to merge multiple transitions. Interestingly, the *cs-macd* models contain use-definition chains spanning the whole simulation, which prevent removal of transitions though many transformations are irrelevant to the state. Future work will need to address this issue, e.g., by also evaluating concrete state values. Regarding total execution time, our adjustments have a small impact as the model checking procedure consumes most of the time. Also, compared to the model checking part our approach scales well with the complexity of the used models.

## V. RELATED WORK

The on-line model checking approach our reconstruction method is complementing and thus closest to has recently been proposed by Li et al. [5], [6]. They employ a hybrid automata model to ensure correct usage of a laser scalpel during laser tracheotomy to prevent burns to the patient. Yet, the necessary model initialization and reconstruction step is a custom solution and is not presented in detail. In the context of on-line model checking with UPPAAL, the UPPAAL variant UPPAAL Tron has been developed [7]. UPPAAL Tron is an on-line testing tool that can generate and execute test cases on-the-fly based on a timed automata system model. While the tool focus lies on input/output testing using a static system model the fact that the underlying model is an UPPAAL model means that our reconstruction approach might be beneficial for tests when the system model is inaccurate or still needs to be developed. Other related work falls in two categories: different ways to use or implement on-line model checking, and different ways to optimize state space exploration and representation in model checkers.

Qi et al. propose an on-line model checking approach to evaluate safety and liveness properties in C/C++ web service systems [8]. Their focus lies on consistency checks for distributed states to debug a system from known source code. Reconstruction is not an issue because the source code is static during execution. Easwaran et al. use a control-theoretic approach to the general runtime verification problem [9]. They introduce a steering component featuring a model to predict execution traces. Their approach uses a fixed prediction model while our reconstruction is for adapting inaccurate models. Sauter et al. address the prediction of system properties using previously gathered time series of measurements, e.g., taken by sensors [10]. They propose a split into an on-line and an off-line computation and to precompute expensive parts of the prediction step to reduce on-line work load. While their scenario of adapting using sensor measurements is applicable to our medical scenario with inaccurate patient models they focus on the verification load problem while we address the model inaccuracy. Harel et al. propose usage of model checking during the behavior and requirement specification step during development. Instead of interactively guiding the system to derive requirements a model checker executes the model and generally finds more adequate requirements. While their approach employs on-line model checking their goal thus lies on early requirement development. In contrast our approach is useful in adaptation of deployed systems to ensure safety. Arney et al. present a recent patient-in-the-loop case study for automatic monitoring and treatment where UPPAAL and Simulink models were developed to verify safety questions beforehand [11]. They monitor heart rate and blood oxygen levels of the patient and automatically control drug infusion via a remote pump. On-line model-checking could benefit this scenario as currently a generalized patient model is employed and drug absorption rates may vary per patient.

Alur and Dill introduced timed automata and the underlying theory in 1994 [12] and Yi et al. developed the first implementation of the model-checker UPPAAL shortly after [13]. Many improvements have been made to the model-checking approach over the years. Larsen et al. proposed symbolic and compositional approaches to reduce the state-space explosion problem [14]. Partial order reduction on the state space was employed by Bengtsson [15]. Larsen et al. reduced memory usage on-the-fly using an algorithm that exploits the control structure of models [2], [16]. Further memory reductions were achieved by Bengtsson et al. with efficient state inclusion checks and compressed state-space representations [17]. Behrmann et al. provide an overview on current functionality and the usage of UPPAAL [3]. They also provide a more detailed presentation of UPPAAL's internal representations [18]. For a summary on timed automata, the semantics, used algorithms, data structures, and tools see [1]. Bengtsson's PhD thesis provides more in-detail information on difference bounded matrices [4].

## VI. Conclusion and Future Work

In this paper we addressed the problem of state reconstruction of UPPAAL models in the context of on-line model checking. Our reconstruction method uses use-definition chains to track influence of individual transformations on the state space during model simulation. With the chains constructed and the additional use of reference counters we are able to identify and remove transformations in the transformation sequence that do not have an impact on the final state space. A prototype implementation was developed and compared to the naive reconstruction approach, which does not remove any transformations. Seven UPPAAL models from different sources were analyzed and our approach reduced the amount of transformations necessary for reconstruction by 23% to 84% and reduced model transitions by up to 39.4%.

In general, the proposed reconstruction method still yields infeasible reconstruction sequences for real-time on-line model checking as the reconstruction sequence length still grows over time. A reconstruction sequence of constant length is desirable to ensure real-time properties. Future research thus could focus on further optimizing the proposed reconstruction method. For example, the proposed method currently only relates transformations according to read and write accesses. Concrete variable values are not taken into account. Transformations that produce the same values could be removed, but are currently not. Experience during development has shown that such transformations occur often especially in periodic use-definition chains that arise due to cycles in the model. Removal of them could improve the reconstruction sequence significantly by breaking such cycles.

## References

[1] J. Bengtsson and W. Yi, "Timed Automata: Semantics, Algorithms and Tools," in Lectures on Concurrency and Petri Nets, J. Desel, W. Reisig, and G. Rozenberg, Eds. Springer Berlin Heidelberg, 2004, ch. 3, pp. 87–124.

[2] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Efficient verification of real-time systems: compact data structure and state-space reduction," in Real-Time Systems Symposium, San Francisco, CA, USA, 1997, pp. 14–24.

[3] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal 4.0," Department of Computer Science, Aalborg University, Aalborg, Denmark, Tech. Rep., 2006.

[4] J. Bengtsson, "Clocks, DBMs and States in Timed Systems," Ph.D. dissertation, Uppsala University, 2002.

[5] T. Li et al., "From offline long-run to online short-run: Exploring a new approach of hybrid systems model checking for mdpnp," in 2011 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability (HCMDSS-MDPnP 2011), 2011.

[6] T. Li et al., "From Offline toward Real-Time: A Hybrid Systems Model Checking and CPS Co-design Approach for Medical Device Plug-and-Play (MDPnP)," in Proceedings of the 3rd ACM/IEEE International Conference on Cyber-Physical Systems - ICCPS '12. Beijing, China: IEEE, April 2012, pp. 13–22.

[7] 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. Springer Berlin Heidelberg, 2008, pp. 77–117.

[8] Z. Qi, A. Liang, H. Guan, M. Wu, and Z. Zhang, "A Hybrid Model Checking and Runtime Monitoring Method for C++ Web Services," in 2009 Fifth International Joint Conference on INC, IMS and IDC. Seoul, South Korea: IEEE, 2009, pp. 745–750.

[9] A. Easwaran, S. Kannan, and O. Sokolsky, "Steering of Discrete Event Systems: Control Theory Approach," Electronic Notes in Theoretical Computer Science, vol. 144, no. 4, 2006, pp. 21–39.

[10] G. Sauter, H. Dierks, M. Fränzle, and M. R. Hansen, "Light-weight hybrid model checking facilitating online prediction of temporal properties," in 21st Nordic Workshop on Programming Theory, NWPT 09, vol. 2, Lyngby, Denmark, 2009.

[11] D. Arney et al., "Toward patient safety in closed-loop medical device systems," in Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems - ICCPS '10. Stockholm, Sweden: ACM New York, NY, USA, 2010, pp. 139–148.

[12] R. Alur and D. L. Dill, "A Theory of Timed Automata," Theoretical Computer Science, vol. 126, no. 2, 1994, pp. 183–235.

[13] W. Yi, P. Pettersson, and M. Daniels, "Automatic verification of real-time communicating systems by constraint-solving," in 7th International Conference on Formal Description Techniques, D. Hogrefe and S. Leue, Eds., 1994, pp. 223–238.

[14] K. G. Larsen, P. Pettersson, and W. Yi, "Compositional and symbolic model-checking of real-time systems," in Real-Time Systems Symposium, Pisa, Italy, 1995, pp. 76–87.

[15] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi, "Partial order reductions for timed systems," in CONCUR'98 Concurrency Theory, D. Sangiorgi and R. de Simone, Eds. Springer Berlin Heidelberg, 1998, pp. 485–500.

[16] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Compact Data Structures and State-Space Reduction for Model-Checking Real-Time Systems," Real-Time Systems, vol. 25, no. 2-3, 2003, pp. 255–275.

[17] J. Bengtsson, "Reducing memory usage in symbolic state-space exploration for timed systems," Department of Information Technology, Uppsala University, Uppsala, Sweden, Tech. Rep. May, 2001.

[18] G. Behrmann et al., "UPPAAL Implementation Secrets," in Formal Techniques in Real-Time and Fault-Tolerant Systems, W. Damm and E.-R. Olderog, Eds. Oldenburg, Germany: Springer-Verlag Berlin, 2002, pp. 3–22.