State Space Reconstruction in UPPAAL: An Algorithm and its Proof

Jonas Rinast, Sibylle Schupp
Institute for 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*—Efficient state space reconstruction is necessary to carry out on-line model checking, a variant of model checking where parameters of a model are continually adjusted to remedy possible modeling faults. On-line model checking is, for example, useful in the medical domain where models of patient states and reactions are always inaccurate, but ensuring patient safety with model checking techniques is still desirable. In this paper, we propose a transformation reduction method based on use-definition chains to efficiently carry out the required state space reconstruction. We provide a formal specification of the general algorithm and proofs for its correctness. For evaluation we applied our reduction approach to the state space of the model checker UPPAAL. The experiments resulted in a reduction of the number of transformations necessary to reach a certain state by 42% on average.**

Keywords*—*State Space Reconstruction; On-line Model Checking; UPPAAL; Reference Counting; Use-Definition Chain.

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 medical staff. 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. Clearly, such a system must 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 for verifying that a system model conforms to its specification and thus may be applied to show the safety of such systems. 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, model checking 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 the human body is complex and varies between individuals, e.g., blood oxygen and heart rate depend on the patient's 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 continually 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.

Because parts of the previous state should be maintained and cross-correlations between state variables could be destroyed inadvertently a model adaptation step may first 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, a data flow analysis capturing relations of data sources and sinks, 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 reconstructed the original state space while yielding a reduction of the executed transformations in the range from 23% to 84%.

To summarize, in this paper we, first, contribute to the field of on-line model checking by presenting a transformation reduction algorithm together with its proof that may be used to reconstruct a particular model-checking state space, and, second, also contribute to the applicability of the UPPAAL model checker by providing a specialization of the algorithm together with its implementation evaluation.

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

This paper is an extended version of the paper published at VALID 2013 [1]. In contrast to the explanation of the reduction approach by way of a running example, this paper provides detailed information on the reduction by formalizing the approach and providing correctness proofs for the algorithm.

II. Related Work

The on-line model checking approach our reconstruction method is complementing and thus is closest to has recently been proposed by Li et al. [2][3]. 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. To our knowledge there are no other reconstruction methods for a particular UPPAAL state in the context of on-line model checking. However, the UPPAAL variant UPPAAL Tron, an on-line testing tool that can generate and execute test cases on-the-fly based on a timed automata system model, has been developed [4]. 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 [5]. 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 [6]. 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 [7]. 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 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 [8]. 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 [9] and Yi et al. developed the first implementation of the model-checker UPPAAL shortly after [10]. 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 [11]. Partial order reduction on the state space was employed by Bengtsson [12]. Larsen et al. reduced memory usage on-the-fly using an algorithm that exploits the control structure of models [13][14]. Further memory reductions were achieved by Bengtsson et al. with efficient state inclusion checks and compressed state-space representations [15]. Behrmann et al. provide an overview on current functionality and the usage of UPPAAL [16]. They also provide a more detailed presentation of UPPAAL's internal representations [17]. For a summary on timed automata, the semantics, used algorithms, data structures, and tools see [18]. Bengtsson's PhD thesis provides more in-detail information on difference bounded matrices [19].

III. On-Line Model Checking

This section 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 [18].

Generally, model checking explores the state space of a 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, jointly developed by Uppsala University, Sweden, and Aalborg University, Denmark [13][16]. 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.

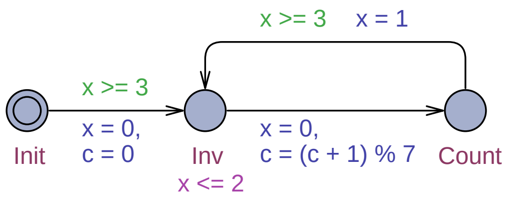

Figure 1. UPPAAL Model Example

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 they are unavailable. It reduces the modeling error by periodically 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 to give meaningful results. Although generally 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.

IV. UPPAAL'S STATE SPACE

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 [18][19]. 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 ∞, which indicates no bound. We denote the set of such entries by $\mathcal{K} = (\{\mathbb{Z} \times \{<, \leq\}\} \cup \infty)$. 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 $(n, <) < (n, \leq)$. Furthermore, 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: $\mathbf{M} \in \mathcal{K}^{|\mathcal{C}_0| \times |\mathcal{C}_0|}$.

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 corresponding DBM is

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

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 [19]:

- *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 ∞.
- *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 to 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: $x \in [1, 2]$

V. State Space Reconstruction

In many models a large number of past 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 influence the valuation of x. Therefore, instead of executing the transition sequence *Init* $\longrightarrow$ *Inv* $\longrightarrow$ *Count* totaling 7 transformations only 3 transformations are required to reach the same state space. The introduction of a new initial state and the direct transition to *Count* with an update `x = 0` is sufficient to recreate this time 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 remainder of this section first formally defines a transformation system and conditions for removing transformations from a sequence (Subsection V-A). Subsection V-B then presents our reduction algorithm based on use-definition chains. The application of the algorithm to the reconstruction problem in UPPAAL is discussed in Subsection V-C. Lastly, Subsection V-D summarizes the complete reconstruction process in UPPAAL.

A. Transformation Elimination Formalized

We now formally derive when transformations may be removed from a general transformation sequence. Examples of the individual parts of the formalization can be found in Subsection V-C, where we specialize the formalization to UPPAAL. Parts of our formalization and its specialization to UPPAAL are also published at FM 2014, where a more recent graph-based state space reconstruction algorithm using the same transformation abstraction is described [20].

Definition 1. *An* evaluation function *is a mapping*

$$e : \mathcal{V} \rightarrow \mathcal{D}$$

where $\mathcal{V}$ is a set of variables and $\mathcal{D}$ is the valuation domain of these variables.

We denote the set of all evaluation functions by $\mathcal{E}(\mathcal{V}, \mathcal{D})$.

Definition 2. *A* transformation *of the evaluation functions is a mapping*

$$t : \mathcal{E}(\mathcal{V}, \mathcal{D}) \rightarrow \mathcal{E}(\mathcal{V}, \mathcal{D})$$

Let $e_1 \xrightarrow{t_1} e_2 \xrightarrow{t_2} \dots \xrightarrow{t_{N-1}} e_N$ be a sequence of evaluation functions created by the transformations t_i where $\xrightarrow{t}$ indicates the application of a single transformation t. We denote the transformation sequence $t_1, t_2, \dots, t_{N-1}$ by T and the evaluation function sequence $e_1, e_2, \dots, e_N$ by E.

For the transformation reduction we are interested in finding a subsequence $T' \subseteq T$ such that $e_1 \overset{T'}{\Longrightarrow} e_N$ where $\overset{T}{\Longrightarrow}$ denotes the ordered application of a sequence of transformations T. Note that we use set notation for sequences although sequences are ordered and may contain duplicates. Such a transformation sequence T' then results in the same last evaluation function e_N but has potentially fewer transformations than T.

The reduction requires the specification of the transformations involved to capture their influence on the evaluation functions.

Definition 3. *An* evaluation calculation *is a mapping*

$$m : 2^{\mathcal{V}} \times \mathcal{E}(\mathcal{V}, \mathcal{D}) \to \mathcal{D}$$

where $m(V, e)$ computes a domain value by using exactly the evaluations $e(v)$ where $v \in V$.

For example, the evaluation calculation

$$\text{add} : (\{ x, y \}, e) \mapsto e(x) + e(y)$$

calculates the sum of the two variables x and y. We denote the set of all evaluation calculations by $\mathcal{C}(\mathcal{V}, \mathcal{D})$.

Definition 4. *The set of* specifications *is*

$$\mathcal{S} = \mathcal{V} \times 2^{\mathcal{V}} \times \mathcal{C}(\mathcal{V}, \mathcal{D})$$

We can specify transformations by providing a subset of $\mathcal{S}$ to the specification function:

Definition 5. *The* specification function *is a mapping*

$$\begin{aligned} s : 2^{\mathcal{S}} &\to (\mathcal{E}(\mathcal{V}, \mathcal{D}) \to \mathcal{E}(\mathcal{V}, \mathcal{D})) \\ S &\mapsto (e \mapsto e') \qquad \textit{where} \\ e'(x') &= \begin{cases} m(V, e) & \textit{if } (x, V, m) \in S \wedge x' = x \\ e(x') & \textit{otherwise} \end{cases} \end{aligned}$$

where $\forall (x, V, m), (x', V', m') \in S \; [x = x' \implies V = V' \wedge m = m']$

Definition 5 states that for each $x \in \mathcal{V}$ a specification set S may contain at most one element (x, V, m). The uniqueness of x in S ensures that the specification function $s(S)$ is well-defined as otherwise the definition for $e'(x')$ is ambiguous. We write (x, V_x, m_x) for the single element for x in S if it exists and further associate the evaluation calculation m_x with a term $m_x(V_x)$ that uses the variable symbols in V_x. We call the term $m_x(V_x)$ *irreducible* if there is no equivalent term in the underlying term algebra that uses fewer subterms, e.g., due to distributivity or the presence of zeros.

Proposition 1. *If there is no equality relationship between the variable symbols, expressed in the term algebra, then there exists a unique specification set S where all evaluation calculation terms are irreducible.*

Proof: Let S, S', $S \neq S'$ be two specification sets with irreducible evaluation calculation terms such that $t = s(S) = s(S')$. Then for all variables $x \in \mathcal{V}$ $[m_x(V_x) = m'_x(V'_x)]$, i.e., the terms are equivalent in the term algebra. Assume w.l.o.g. V_x contains a variable symbol y not contained in V'_x. Then either the term $m_x(V_x)$ can be rewritten in a way that eliminates y and $m_x(V_x)$ was not irreducible or the variable symbol y can not be eliminated. Then $m_x(V_x) = m'_x(V'_x)$ constitutes a non-trivial equality relation between the variable symbols. Both cases contradict the assumption and thus $S = S'$. ■

Definition 6. *The* specification set *of a transformation t is the unique set*

$$S(t) \subset \mathcal{S}$$

with irreducible evaluation calculation terms such that $t = s(S(t))$.

The specification set $S(t)$ of a transformation t captures all modifications to the input evaluation function as every member (x, V, m) defines which (x) and how (m) a variable is modified. Thus, providing a specification set for t fully characterizes t. Furthermore, using the specification set we can derive the write and read characteristics of the transformation.

Definition 7. *The* write set *of a transformation t is*

$$\mathcal{W}(t) = \bigcup_{(x,V,m) \in S(t)} \{ x \}$$

Definition 8. *The* read set *of a transformation t is*

$$\mathcal{R}(t) = \bigcup_{(x,V,m) \in S(t)} V$$

Moreover, to conveniently chain transformations together we define *compound transformations* in addition to the simple transformations.

Definition 9. *A* compound transformation *is a transformation*

$$t_c = t_1 \circ t_2 \circ \cdots \circ t_n$$

such that $e_1 \xrightarrow{t_c} e_{n+1} \equiv e_1 \xrightarrow{t_1} e_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} e_{n+1}$.

Note that $\circ$ denotes concatenation ($(a \circ b)(x) = b(a(x))$) in contrast to composition ($(a \circ b)(x) = a(b(x))$). For compound transformations the write and read sets can be derived from the individual transformations t_i.

Definition 10. *The* write set *of a compound transformation t_c is*

$$\mathcal{W}(t_c) = \bigcup_i \mathcal{W}(t_i)$$

Definition 11. *The* read set *of a compound transformation t_c is*

$$\mathcal{R}(t_c) = \bigcup_i (\mathcal{R}(t_i) \setminus \bigcup_{j<i} \mathcal{W}(t_j))$$

Using these definitions two cases exist where a transformation t_i may be removed from the transformation sequence T without changing e_N. The cases can be described in the following way:

1) *no write*
$\mathcal{W}(t_i) = \emptyset$
The transformation t_i may be removed if the write set $\mathcal{W}(t_i)$ is empty.
Proof: As $\mathcal{W}(t_i) = \emptyset \implies S(t_i) = \emptyset \implies t_i = I$ by definition we find $e_i \xrightarrow{t_i} e_{i+1} \implies e_i = e_{i+1}$ and thus $e_i \xrightarrow{t_i} e_{i+1} \xrightarrow{t_{i+1}} e_{i+2} = e_i \xrightarrow{t_{i+1}} e_{i+2}$. ■

2) *no read overwritten*
$\forall x \in \mathcal{W}(t_i) \; [j > i \wedge x \in \mathcal{R}(t_j) \implies \exists k \; [i < k < j \wedge x \in \mathcal{W}(t_k)]]$
The transformation t_i may be removed if for all following transformations t_j that read a variable written by t_i there is a transformation t_k between t_i and t_j that writes that variable.
Proof: Let $e'_i \xrightarrow{I} e'_{i+1} \to \cdots \to e'_M$ be the transition sequence E' that replaces t_i with the identity transformation I in $e_i \xrightarrow{t_i} e_{i+1} \to \cdots \to e_M$. Note

that the new transformation sequence is equivalent to removing t_i in the old one. Now if we assume $e_M \neq e'_M$ then there must be a variable x such that $e_M(x) \neq e'_M(x)$. It follows that there must be a transformation t_j, $j < M$, such that $e_j(x) = e'_j(x)$ and $e_{j+1}(x) \neq e'_{j+1}(x)$ because $e_i = e'_i$. Hence, the variable x is written by t_j, i.e., $x \in \mathcal{W}(t_j)$. As $x \in \mathcal{W}(t_j)$, $e_{j+1}(x) = m(V, e_j)$ for $(x, V, m) \in S(t_j)$ and there must be a variable $v \in \mathcal{R}(t_j)$ such that $e_j(v) \neq e'_j(v)$ to satisfy $m(V, e_j) \neq m(V, e'_j)$. If $j = i$, we have a contradiction to the assumption $e_i = e'_i$ and are finished. Otherwise, we apply the same argument to the variable v and would get another transformation t_k, $k < j$, and a variable $w \in \mathcal{R}(t_k)$ such that $e_k(w) \neq e'_k(w)$. This process leads to a contradiction to our assumption in at most $M - i$ iterations. ■

B. Use-Definition Chain Reduction

The key idea of our approach is the construction of use-definition chains to identify transformations satisfying the requirements presented in Subsection V-A. 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.

We now propose an algorithm that removes the transformations in question. Our algorithm consists of two smaller algorithms: the APPLY algorithm and the ELIMINATE algorithm. The APPLY algorithm is used to perform a transformation and the ELIMINATE algorithm removes unnecessary transformations in a sequence of applied transformations. Usage of the algorithms is assumed as follows: for a sequence of transformations T first the APPLY algorithm is called on all transformations in order, then the ELIMINATE algorithm is executed to obtain the reduced transformation sequence. Note that a transformation sequence T could be split into two subsequent sequences T_1 and T_2 and the ELIMINATE algorithm could be run on the sequence T_1 as soon as all transformations in T_1 have been applied. Thus, the ELIMINATE algorithm can be run with an appropriate transformation sequence after every execution of APPLY , which achieves on-the-fly removal. However, for formalization purposes, we assume the algorithm execution sequence given in Figure 2 where the initial mappings c_1, r_1 and u_1 satisfy $\forall t \in T\, [c_1(t) = 0 \wedge u_1(t) = \emptyset] \wedge \forall x \in \mathcal{V} [r_1(x) = \bot] \wedge c_1(\bot) = |\mathcal{V}|$. The algorithm generates the following sequences:

- Transformation sequence $T = t_1, t_2, \ldots, t_{N-1}$
- Evaluation function sequence $E = e_1, e_2, \ldots, e_N$
- Reference counter sequence $C = c_1, c_2, \ldots, c_N$

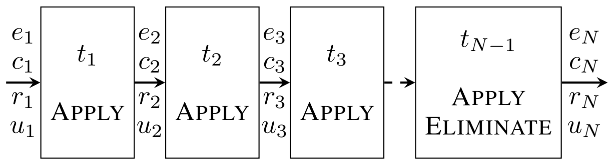

Figure 2. Algorithm Execution Sequence

- Responsibility sequence $R = r_1, r_2, \ldots, r_N$
- Use-definition sequence $U = u_1, u_2, \ldots, u_N$

where a *reference counter* is a mapping $c : T \cup \{\bot\} \to \mathbb{N}_0$, a *responsibility mapping* is a mapping $r : \mathcal{V} \to T \cup \{\bot\}$, and an *use-definition mapping* is a mapping $u : T \to 2^T$.

Figure 3 shows the APPLY algorithm. The algorithm takes the to-be-applied transformation t, an evaluation function e, a reference counter c, a responsibility mapping r, and a use-definition mapping u as parameters. Except t all the parameters are stored locally in lines 2 to 5 to allow internal manipulations and the results are returned in lines 17 to 20. The algorithm can be divided into two parts, one handling the reads of the transformation t and one handling its writes. Lines 6 to 11 process the variables read by t. For every variable x_i it is determined if the transformation responsible for its current valuation $r(x_i)$ is already used by t in line 7. If $r(x_i)$ was not marked used yet it is marked used in line 9 and the reference counter is increased accordingly in line 8. Lines 12 to 16 then handle the writes of t. First the evaluation function is updated to map x_i to the value $m_i(V_i, e)$ as specified by the transformation. Then line 14 reduces the reference counter of the transformation previously responsible for the value of x_i and line 15 updates the responsibility mapping such that now the transformation t is responsible for the value of x_i. Lastly in the return section in line 18 the reference counter is set to $|S(t)|$ to show that t is now responsible for $|S(t)|$ variable valuations.

Figure 4 shows the ELIMINATE algorithm. The algorithm takes a sequence of transformations T, a reference counter c, and a use-definition mapping u as parameters where all transformations in T must already have been applied with the APPLY algorithm. In lines 2 to 3 the transformation sequence and the reference counter are stored locally and the results of the algorithm are returned in lines 16 to 17. In between, in lines 4 to 15, a fix point is calculated by removing as many transformations from T as possible. The algorithm checks for every transformation t in T if the reference counter $c(t)$ evaluates to zero in line 7. If a transformation t satisfies $c(t) = 0$ the algorithm at first removes t from T in line 8, then adjusts the reference counter by decreasing all counters of transformations used by t in lines 9 to 11, and lastly schedules another iteration of the fix point calculation in line 12 as the modifications to the reference counters may induce additional removals.

We now show the correctness of the algorithm. We assume that there implicitly is a transformation t_e appended to the transformation sequence T that satisfies $\mathcal{R}(t_e) = \mathcal{V}$ to indicate that the final calculated values are actually read and need to be recreated correctly. Otherwise, the whole transformation

```
 1: procedure APPLY($t, e, c, r, u$)
 2:     $e_l \leftarrow e$
 3:     $c_l \leftarrow c$
 4:     $r_l \leftarrow r$
 5:     $u_l \leftarrow u$
 6:     for all $x_i \in \mathcal{R}(t)$ do
 7:         if $r(x_i) \notin u_l(t)$ then
 8:             $c_l \leftarrow c_l[r(x_i) \mapsto c_l(r(x_i))+1]$
 9:             $u_l \leftarrow u_l[t \mapsto u_l(t) \cup \{ r(x_i) \}]$
10:         end if
11:     end for
12:     for all $(x_i, V_i, m_i) \in S(t)$ do
13:         $e_l \leftarrow e_l[x_i \mapsto m_i(V_i, e_l)]$
14:         $c_l \leftarrow c_l[r(x_i) \mapsto c_l(r(x_i)) - 1]$
15:         $r_l \leftarrow r_l[x_i \mapsto t]$
16:     end for
17:     $e' \leftarrow e_l$
18:     $c' \leftarrow c_l[t \mapsto |S(t)|]$
19:     $r' \leftarrow r_l$
20:     $u' \leftarrow u_l$
21: end procedure
```

Figure 3. APPLY Algorithm

```
Require: $\forall t \in T[\text{APPLY}(t, ...)]$
 1: procedure ELIMINATE($T, c, u$)
 2:     $T_l \leftarrow T$
 3:     $c_l \leftarrow c$
 4:     repeat
 5:         $b \leftarrow true$
 6:         for all $t \in T_l$ do
 7:             if $c_l(t) = 0$ then
 8:                 $T_l \leftarrow T_l \setminus \{ t \}$
 9:                 for all $s \in u(t)$ do
10:                     $c_l \leftarrow c_l[s \mapsto c_l(s)-1]$
11:                 end for
12:                 $b \leftarrow false$
13:             end if
14:         end for
15:     until $b = true$
16:     $T' \leftarrow T_l$
17:     $c' \leftarrow c_l$
18: end procedure
```

Figure 4. ELIMINATE Algorithm

sequence T could be removed as no data is consumed. As a transformation t can only be removed in line 8 of the ELIMINATE algorithm, which is only executed if $c(t) = 0$, the following implications need to be shown to prove the algorithm.

1) $\mathcal{W}(t) = \emptyset \implies \forall c \in C\,[c(t) = 0] \quad (\Rightarrow)$
   If a transformation t does not write any variable then the transformation may be removed at any time.
2) $\forall x \in \mathcal{W}(t_i)\,[\forall t_j \in T\,[i < j \wedge x \in \mathcal{R}(t_j) \implies \exists t_k \in T\,[i < k < j \wedge x \in \mathcal{W}(t_k)]]] \implies \forall c_l \in C\,[l > \max k \implies c_l(t_i) = 0] \quad (\Rightarrow)$
   If every variable written by a transformation t_i is overwritten by transformations t_k before it is read by a transformation t_j then the transformation may be removed as soon as the last overwriting transformation t_k is performed.
3) $i < l \wedge c_l(t_i) = 0 \implies \mathcal{W}(t_i) = \emptyset \vee \forall x \in \mathcal{W}(t_i)[\nexists t_j \in T[i < j \wedge x \in \mathcal{R}(t_j) \wedge c_l(t_j) \neq 0 \wedge \forall t_k \in T\,[i < k < j \implies x \notin \mathcal{W}(t_k)]]] \quad (\Leftarrow)$
   If a transformation t_i may be removed the transformation either does not write any variable or every variable written by it is overwritten overwritten by transformations t_k without being read beforehand.
   Derivation Note: Application of identities yields $\forall x \in \mathcal{W}(t_i)\,[\forall t_j \in T\,[i < j \wedge x \in \mathcal{R}(t_j) \implies \exists t_k \in T\,[i < k < j \wedge x \in \mathcal{W}(t_k)]]] \Leftrightarrow \forall x \in \mathcal{W}(t_i)\,[\nexists t_j \in T\,[i < j \wedge x \in \mathcal{R}(t_j) \wedge \forall t_k \in T\,[i < k < j \implies x \notin \mathcal{W}(t_k)]]]$. Adding $c_l(t_j) \neq 0$ only makes sure the transformation can not be removed, i.e., it really exists.

We first derive two lemmas that characterize execution dependencies of certain lines in the algorithms to break the proof down into smaller parts.

Lemma 1. *Every execution of line 10 in the* ELIMINATE *algorithm is preceded by an execution of line 8 of the* APPLY *algorithm where $r(x) = s$ for some variable x, i.e., for every decrease of $c_j(s)$ in line 10 there is an increase of $c_i(s)$ in line 8 where $i < j$.*

Proof: An execution of line 10 in the ELIMINATE algorithm for a transformation s implies $s \in u(t)$ for some transformation t because of line 9. As only line 9 of the APPLY algorithm modifies u an execution of it is implied where $r(x) = s$ for some variable x. Additionally, line 10 may not be executed multiple times with the same transformation s for a single execution of line 8 because $u(t)$ is a set and therefore does not contain duplicates of s and the transformation t is removed from T_l in line 8 rendering a subsequent access to $u(t)$ impossible. It follows that every execution of line 10 in the ELIMINATE algorithm is paired with a preceeding execution of line 8 in the APPLY algorithm. ■

Lemma 2. *Before the execution of the* APPLY *algorithm for a transformation t there are zero variables x_i that satisfy $r(x_i) = t$. After its execution there are at all times at most $|S(t)|$ variables x_i that satisfy $r(x_i) = t$ for a transformation t.*

Proof: Only line 15 of the APPLY algorithm may modify $r(x)$. A valuation satisfying $r(x) = t$ can only be established when it is executed for the transformation t. In that case line 15 is executed $|S(t)|$ times due to line 12 and because $\forall (x_i, V_i, m_i), (x_j, V_j, m_j) \in S(t)\,[x_i \neq x_j]$ there are exactly $|S(t)|$ variables x_i that satisfy $r(x_i) = t$ after the execution of the APPLY algorithm for t. Because subsequent executions of line 15 always result in valuations $r(x) \neq t$ for a variable x it follows that the amount of variables x_i that satisfy $r(x_i) = t$ may only be reduced. The proposition follows by taking into consideration that initially $\forall x \in \mathcal{V}\,[r(x) = \bot]$. ■

Corollary 1. *Line 14 of the* APPLY *algorithm may be executed for a transformation t at most $|S(t)|$ times, i.e., line 14 reduces $c(t)$ by one at most $|S(t)|$ times.*

Proof: An execution of line 14 for a transformation t implies that $r(x) = t$ for a variable x. Additionally, the execution is always followed by an execution of line 15, which sets $r(x) \neq t$. It follows that every execution of line 14 for a transformation t implies a reduction of the amount of variables that satisfy $r(x) = t$ by one. It follows that line 14 may at most be executed $|S(t)|$ times for a transformation t due to Lemma 2. ■

We now prove the algorithm correct by showing that the presented requirements hold.

1) $\mathcal{W}(t) = \emptyset \implies \forall c \in C\,[c(t) = 0]$

   Proof: There are four occurrences where reference counters may be modified: in lines 8, 14 and 18 of the APPLY algorithm and in line 10 of the ELIMINATE algorithm.

   a) APPLY algorithm, line 8
      Due to Lemma 2 $\mathcal{W}(t) = \emptyset \implies S(t) = \emptyset \implies \forall r \in R\,[\nexists x \in \mathcal{V}\,[r(x) = t]]$ and thus line 8 cannot modify $c(t)$.
   b) APPLY algorithm, line 14
      Due to Lemma 2 $\mathcal{W}(t) = \emptyset \implies S(t) = \emptyset \implies \forall r \in R\,[\nexists x \in \mathcal{V}\,[r(x) = t]]$ and thus line 14 cannot modify $c(t)$.
   c) APPLY algorithm, line 18
      As $\mathcal{W}(t) = \emptyset \implies S(t) = \emptyset$ line 18 sets $c(t)$ to $|S(t)| = 0$ if APPLY is executed for t and $c(t)$ is not modified otherwise.
   d) ELIMINATE algorithm, line 10
      As line 8 cannot modify $c(t)$ (see above) line 10 cannot modify $c(t)$ due to Lemma 1.

   According to the case analysis it follows that $c'(t) = c(t)$ or $c'(t) = 0$ for every application of the APPLY or ELIMINATE algorithm. Using that $c_1(t) = 0$ it follows that $\mathcal{W}(t) = \emptyset \implies \forall c \in C[c(t) = 0]$ by induction. ■

2) $\forall x \in \mathcal{W}(t_i)\,[\forall t_j \in T\,[i < j \wedge x \in \mathcal{R}(t_j) \implies \exists t_k \in T\,[i < k < j \wedge x \in \mathcal{W}(t_k)]]] \implies \forall c_l \in C\,[l > \max k \implies c_l(t_i) = 0]$

   Proof: There are four occurrences where reference counters may be modified: in lines 8, 14 and 18 of the APPLY algorithm and in line 10 of the ELIMINATE algorithm. We consider the transformation t_i.

   a) APPLY algorithm, line 8
      Assume a transformation t_j modifies $c(t_i)$ in line 8. Then due to Lemma 2 it follows that $i < j$ and $\exists x \in \mathcal{V}\,[r(x) = t_i]$. It follows that $\exists x \in \mathcal{V}\,[x \in \mathcal{R}(t_j) \wedge x \in \mathcal{W}(t_i)]$ must be satisfied. Thus, the inner premise in the proposition holds and there must be a transformation t_k satisfying $i < k < j$ and $x \in \mathcal{W}(t_k)$. The execution of the APPLY algorithm for the transformation t_k, however, then results in $r(x) \neq t_i$ (see proof of Lemma 2) and t_j can no longer modify $c(t_i)$. The premise thus prevents line 8 from modifying $c(t_i)$.
   b) APPLY algorithm, line 14
      As we are only interested in modifications to $c(t_i)$ we only need to consider executions for transformations $t_j, j > i$ due to Lemma 2 as line 14 requires $\exists x \in \mathcal{V}[r(x) = t_i]$ to modify $c(t_i)$. According to Corollary 1 line 14 may reduce $c(t_i)$ maximally by $|S(t_i)|$. The maximum reduction occurs if $\forall x \in \mathcal{W}(t_i)\,[\exists t_j \in T\,[i < j \wedge x \in \mathcal{W}(t_j)]]$ (see proof of Corollary 1). According to the premise of the proposition this requirement is satisfied if $\forall x \in \mathcal{W}(t_i)\,[\exists t_j \in T\,[i < j \wedge x \in \mathcal{R}(t_j)]]$. This requirement, however, is always satisfied because of the implicit transformation t_e at the end of the transformation sequence, which satisfies $\mathcal{R}(t_e) = \mathcal{V}$. It follows that the premise of the proposition implies the existence of a set of transformations $T_r \subseteq T$, which satisfies $\mathcal{W}(t_i) \subseteq \bigcup_{t \in T_r} \mathcal{W}(t)$ and, thus, line 14 reduces $c(t_i)$ by $|S(t_i)|$ in total.
   c) APPLY algorithm, line 18
      As we are only interested in modifications to $c(t_i)$ we only need to consider the execution for t_i. In that case line 18 sets $c(t_i)$ to $|\mathcal{W}(t_i)|$ as $|\mathcal{W}(t_i)| = |S(t_i)|$.
   d) ELIMINATE algorithm, line 10
      Due to Lemma 1 there is no execution of line 10 that modifies $c(t_i)$ as there is no modification of $c(t_i)$ in line 8 in the APPLY algorithm (see above).

   According to the case analysis $c(t_i)$ is modified in the following way: at first line 18 sets $c(t_i)$ to $|S(t_i)|$. Then the transformations from the set T_r are executed and lead to a monotonously descending $c(t_i)$ value (no reads). The execution of the last transformation from T_r results in a $c(t_i) = 0$. This transformation is the transformation with the highest k of the transformations t_k in the proposition as all transformations t_k satisfy $\mathcal{W}(t_i) \cap \mathcal{W}(t_k) \neq \emptyset$. Thus $\forall c_l \in C\,[l > \max k \implies c_l(t_i) = 0]$ is satisfied and the proposition holds. ■

3) $i < l \wedge c_l(t_i) = 0 \implies \mathcal{W}(t_i) = \emptyset \vee \forall x \in \mathcal{W}(t_i)[\nexists t_j \in T[i < j \wedge x \in \mathcal{R}(t_j) \wedge c_l(t_j) \neq 0 \wedge \forall t_k \in T\,[i < k < j \implies x \notin \mathcal{W}(t_k)]]]$

   Proof: There are three occurrences where reference counters may be set or reduced to zero after t_i is processed. Lines 14 and 18 of the APPLY algorithm and line 10 of the ELIMINATE algorithm potentially modify $c_l(t_i)$ in such a way.

   a) APPLY algorithm, line 14
      In this case we prove $i < l \wedge c_l(t_i) = 0 \implies \forall x \in \mathcal{W}(t_i)\,[\nexists t_j \in T\,[i < j \wedge x \in \mathcal{R}(t_j) \wedge c_l(t_j) \neq 0 \wedge \forall t_k \in T\,[i < k < j \implies x \notin \mathcal{W}(t_k)]]]$ by contraposition. Assume x to be a variable satisfying $x \in \mathcal{W}(t_i)$ and assume t_j to be a transformation satisfying $i < j \wedge x \in \mathcal{R}(t_j) \wedge c_l(t_j) \neq 0 \wedge \forall t_k \in T[i < k < j \implies x \notin \mathcal{W}(t_k)]$. When executed t_j increases $c_j(t_i)$ in line 8 as $x \in \mathcal{R}(t_j)$ and no intermediate transformation t_k invalidates $r(x) = t_i$. Then for the transformation t_{l-1} to reduce $c_l(t_i)$ to zero in line 14 it is necessary to revert the increase by an execution of line 10 in the ELIMINATE algorithm before t_{l-1} is executed due to Corollary 1. Due to Lemma 1 the reduction must result from $c(t_j)$ being reduced to zero (ELIMINATE algorithm, line

7) as otherwise additional increases would have happened in line 8 beforehand. Assume t_m, $j < m < l-1$ to be the transformation that reduced $c_{m+1}(t_j)$ to zero to revert the increase of $c_j(t_i)$. We find that $c_{m+1}(t_j) = 0 \implies c_l(t_j) = 0$ unless $c(t_j)$ is increased again between the execution of t_m and t_{l-1}. However, a reduction of $c(t)$ to zero implies $\nexists x \in \mathcal{V}\,[r(x) = t]$ and thus such an increase is impossible. It follows that if t_j exists $c_l(t_i)$ can not be reduced to zero.

b) APPLY algorithm, line 18
Line 18 may only modify $c_l(t_i)$ if APPLY is executed for t_i. In that case $c_l(t_i)$ is set to zero if $|S(t_i)| = 0$. As $|S(t_i)| = 0 \implies \mathcal{W}(t_i) = \emptyset$ it follows that in this case $c_l(t_i) = 0 \implies \mathcal{W}(t_i) = \emptyset$, which is part of the proposition.

c) ELIMINATE algorithm, line 10
If line 10 reduces $c_l(t_i)$ to zero then there must have been a reduction of a different reference counter to zero beforehand as line 7 requires $c(t) = 0$. As this argument holds recursively a reduction in line 10 ultimately implies a reduction due to either line 14 or line 18 in the APPLY algorithm. Thus, a reduction in line 10 implies the implications for those lines found above.

Combining the case analysis results with the premise $i < l \wedge c_l(t_i)$ results in all cases in either $\mathcal{W}(t_i) = \emptyset$ or $\forall x \in \mathcal{W}(t_i)\,[\nexists t_j \in T\,[i < j \wedge x \in \mathcal{R}(t_j) \wedge c_l(t_j) \neq 0 \wedge \forall t_k \in T\,[i < k < j \implies x \notin \mathcal{W}(t_k)]]]$ yielding the proposition. ■

C. Algorithm Application to UPPAAL

We now specialize the general formalization to UPPAAL's state space and transformation system such that we may apply the presented reduction method to UPPAAL models. The reduction is only relevant for the time state of UPPAAL because the data and location states may be modified directly during on-line model checking. Only the time state has constraints that may be invalidated by individual changes to the time state.

Consider an UPPAAL model $\mathcal{M}$ with the clock set $\mathcal{C}$. The time state of $\mathcal{M}$ can be represented with a difference bound matrix containing $|\mathcal{C}_0|^2$ entries. Therefore, in our specialization the variable set $\mathcal{V}$ contains that many variables: $|\mathcal{V}| = |\mathcal{C}_0|^2$. The domain of the variables ($\mathcal{D}$) is $\mathcal{K}$ because every variable represents an DBM entry. We refer to the variables by $\mathrm{DBM}_{r,c}$ where r denotes the row number and c denotes the column number. Next, we define UPPAAL's DBM transformations for our formalization. As presented in Section IV, the relevant transformations are the *Clock Reset* (RESET(x, v)), the Constraint Introduction (CONSTRAINT$(x, y, v, \preceq)$), and the *Bound Elimination* (UP):

RESET(x, v)	Sets the clock variable x to the value v and adjusts constraints on that clock accordingly
CONSTRAINT$(x, y, v, \preceq)$	Introduces a new upper bound on a clock or on a difference of clocks and propagates dependencies
UP	Removes the upper bounds on every clock but not on differences of clocks

Four specification calculations are necessary to specify these transformations, where two assign values based on constants and two calculate minima. The first two are the $\mathrm{assign}(v)$ calculation and the $\mathrm{add}(v)$ calculation: $\mathrm{assign}(v)$ simply assigns the constant value v to a variable; $\mathrm{add}(v)$ calculates the sum of the constant value v and the current evaluation of a variable and assigns it:

$$\begin{aligned} \mathrm{assign} : \mathcal{K} &\to (2^{\mathcal{V}} \times \mathcal{E}(\mathcal{V}, \mathcal{D}) \to \mathcal{K}) \\ v &\mapsto ((\emptyset, e) \mapsto v) \\ \mathrm{add} : \mathcal{K} &\to (2^{\mathcal{V}} \times \mathcal{E}(\mathcal{V}, \mathcal{D}) \to \mathcal{K}) \\ v &\mapsto ((\{\, x \,\}, e) \mapsto e(x) + v) \end{aligned}$$

The minima calculations are $\mathrm{minassign}(v)$ and $\mathrm{minadd}()$. Assigning a variable with $\mathrm{minassign}(v)$ results in an evaluation equal to the minimum of v and the evaluation of the comparing variable. $\mathrm{minadd}()$ checks whether the sum of two variable evaluations is smaller than a third evaluation and if so assigns the sum:

$$\begin{aligned} \mathrm{minassign} : \mathcal{K} &\to (2^{\mathcal{V}} \times \mathcal{E}(\mathcal{V}, \mathcal{D}) \to \mathcal{K}) \\ v &\mapsto ((\{\, x \,\}, e) \mapsto \min(e(x), v)) \\ \mathrm{minadd} : 2^{\mathcal{V}} \times \mathcal{E}(\mathcal{V}, \mathcal{D}) &\to \mathcal{K} \\ (\{\, x, y, z \,\}, e) &\mapsto \min(e(x), e(y) + e(z)) \end{aligned}$$

Providing adequate specification sets $S(t)$ with these specification calculates now allows defining the transformations UP, RESET(x, v), and CONSTRAINT$(x, y, v, \preceq)$. For convenience we use i_x and i_y for the indices of the clocks x and y in the DBMs. The UP transformation begins straight-forward: setting all values in the first DBM column except the top-most one to ∞ removes the upper bounds on the clocks:

$$S(\textsc{Up}) = \{\, (\mathrm{DBM}_{i,1}, \emptyset, \mathrm{assign}(\infty)) \mid 1 < i \leq |\mathcal{C}_0| \,\}$$

The RESET(x, v) transformation performs two actions. First, it sets x to v, i.e., it sets both bounds to v. Then constraints in the clock's row and column are adjusted. A compound transformation models this behavior:

$$\begin{aligned} \textsc{Reset}(x, v) &= t_s \circ t_p \\ S(t_s) &= \{\, (\mathrm{DBM}_{i_x,1}, \emptyset, \mathrm{assign}((v, \leq))), \\ &\quad (\mathrm{DBM}_{1,i_x}, \emptyset, \mathrm{assign}((-v, \leq))) \,\} \\ S(t_p) &= \{\, (\mathrm{DBM}_{i_x,i}, \{\, \mathrm{DBM}_{1,i} \,\}, \mathrm{add}((v, \leq))), \\ &\quad (\mathrm{DBM}_{i,i_x}, \{\, \mathrm{DBM}_{i,1} \,\}, \mathrm{add}((-v, \leq))) \\ &\quad \mid 1 < i \leq |\mathcal{C}_0| \,\} \end{aligned}$$

At last, the CONSTRAINT$(x, y, v, \preceq)$ transformation first introduces the constraint $x - y \preceq v$ and then propagates it to depending constraints. Note that the propagation itself also is divided into multiple transformations as subsequent transformations require previous calculations:

$$\begin{aligned}\textsc{Constraint}(x, y, v, \preceq) = t_c \circ& \\ & t_{1,1} \circ \cdots \circ t_{1,|\mathcal{C}_0|} \circ \\ & t_{2,1} \circ \cdots \circ t_{2,|\mathcal{C}_0|} \circ \\ & \vdots \\ & t_{|\mathcal{C}_0|,1} \circ \cdots \circ t_{|\mathcal{C}_0|,|\mathcal{C}_0|} \\ t_{i,j} = t_{i,j,1} \circ t_{i,j,2}&\end{aligned}$$

$$\begin{aligned}S(t_c) &= \{\, (\mathrm{DBM}_{i_x,i_y}, \{\, \mathrm{DBM}_{i_x,i_y} \,\}, \mathrm{minassign}((v, \preceq))) \,\} \\ S(t_{i,j,1}) &= \{\, (\mathrm{DBM}_{i,j}, \{\, \mathrm{DBM}_{i,j}, \mathrm{DBM}_{i,i_x}, \mathrm{DBM}_{i_x,j} \,\}, \\ &\quad \mathrm{minadd}) \,\} \\ S(t_{i,j,2}) &= \{\, (\mathrm{DBM}_{i,j}, \{\, \mathrm{DBM}_{i,j}, \mathrm{DBM}_{i,i_y}, \mathrm{DBM}_{i_y,j} \,\}, \\ &\quad \mathrm{minadd}) \,\}\end{aligned}$$

D. Reconstruction Summarized

The complete state space reconstruction process consists of three steps:

1) *Initialization* Canonize model by introducing general starting points for later model synthesis, extract necessary information from the model, e.g., clock and variable definitions.
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 use the APPLY algorithm to internally construct the use-definition chains of the transformations. Then use the ELIMINATE algorithm to remove unnecessary transformations on-the-fly by evaluating the reference counters.
3) *Synthesis* Group the sequence of reduced transformations to form valid transitions and add the transitions to a newly created model obtained from the original one. The last transitions connect to the current locations when the reconstruction was initiated. Those transitions also update the data state.

Note that the synthesis of the actual UPPAAL model from the reduced 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 initialization 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 5 shows the reconstruction model for the example model (Figure 1) after 2 transitions on the right side. The additional initialization automaton is shown on the left. It 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 DBM transformations the reconstructed model uses three transformations while the original model needs seven.

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%

VI. 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 [21][22]. 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 *csmacd* 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

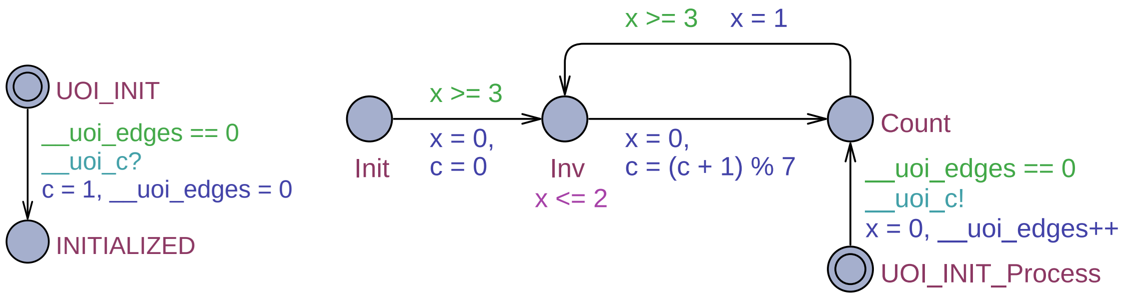

Figure 5. Reconstructed example model

of the time. Also, compared to the model checking part our approach scales well with the complexity of the used models.

VII. Conclusion and Future Work

In this paper, we addressed the problem of state space 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. An algorithm for the chain construction with reference counters was presented. It is able to identify and remove transformations in the transformation sequence that do not have an impact on the final state space. We provided proofs for the algorithm itself and the requirements for the removal of a transformation without altering the final state. The reconstruction process was implemented in a prototype implementation 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%.

The prototype implementation is part of our UPPAAL on-line model checking interface that is currently in development. Interestingly, this interface could not only be used to automatically carry out on-line model checking. The interface also allows generic dynamic adaptation of model parameters. In the future enhancing on-line model checking by combining it with parameter learning algorithms and model calibration methods might broaden the applicability of model checking even further.

However, the proposed reconstruction method still yields infeasible reconstruction sequences for real-time on-line model checking in general 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 also need to 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. Rinast, S. Schupp, and D. Gollmann, “State space reconstruction for on-line model checking with UPPAAL,” in VALID 2013, The Fifth Internation Conference on Advances in System Testing and Validation Lifecycle, 2013, pp. 21–26.

[2] T. Li, Q. Wang, F. Tan, L. Bu, J.-n. Cao, X. Liu, Y. Wang, and R. Zheng, “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.

[3] T. Li, F. Tan, Q. Wang, L. Bu, J.-N. Cao, and X. Liu, “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.

[4] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, “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.

[5] 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.

[6] 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.

[7] 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.

[8] D. Arney, M. Pajic, J. M. Goldman, I. Lee, R. Mangharam, and O. Sokolsky, “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.

[9] R. Alur and D. L. Dill, “A theory of timed automata,” Theoretical Computer Science, vol. 126, no. 2, 1994, pp. 183–235.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, “Compact data struc-

tures and state-space reduction for model-checking real-time systems," Real-Time Systems, vol. 25, no. 2-3, 2003, pp. 255–275.

[15] 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.

[16] 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.

[17] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi, "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.

[18] 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.

[19] J. Bengtsson, "Clocks, dbms and states in timed systems," Ph.D. dissertation, Uppsala University, 2002.

[20] J. Rinast, S. Schupp, and D. Gollmann, "A graph-based transformation reduction to reach UPPAAL states faster," in 19th International Symposium on Formal Methods 2014 (FM2014), ser. Lecture Notes in Computer Science. Springer Verlag, 2014, pp. 547–562.

[21] S. Yovine, "KRONOS: a verification tool for real-time systems," International Journal on Software Tools for Technology Transfer, vol. 1, no. 1-2, 1997, pp. 123–133.

[22] H. Lönn and P. Pettersson, "Formal verification of a tdma protocol start-up mechanism," in Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS '97). Taipei: Ieee, 1997, pp. 235–242.