International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-022-00667-x

GENERAL

Regular

Bounded DBM-based clock state construction for timed automata in Uppaal

Sascha Lehmann[1] · **Sibylle Schupp**[1]

Accepted: 9 August 2022

Abstract
When the simulation of a system, or the verification of its model, needs to be resumed in an online context, we face the problem that a particular starting state needs to be reached or constructed, from which the process is then continued. For timed automata, especially the construction of a desired clock state, represented as a difference bound matrix (*DBM*), can be problematic, as only a limited set of DBM operations is available, which often does not include the ability to set DBM entries individually to the desired value. In online applications, we furthermore face strict timing requirements imposed on the generation process. In this paper, we present an approach to construct a target clock state in a model via sequences of DBM operations (as supported by the model checker *Uppaal*), for which we can guarantee bounded lengths, solving the present problem of ever-growing sequences over time. The approach forges new intermediate states and transitions based on an overapproximation of the target state, followed by a constraining phase, until the target state is reached. We prove that the construction sequence lengths are independent of the original trace lengths and are determined by the number of system clocks only, allowing for state construction in bounded time. Furthermore, we implement the (re-)construction routines and an extended *Uppaal* model simulator which provides the original operation sequences. Applying the approach to a test model suite as well as randomly generated DBM operation sequences, we empirically validate the theoretical result and the implementation.

Keywords Clock state construction · Difference bound matrix (DBM) · Timed automata · DBM overapproximation · Minimal constraint system (MCS)

1 Introduction

State (re-)construction, i.e., setting a system to a desired state, is a common task. Ranging from physical systems to programs and executable models, a system may need to be in some specific state to execute particular routines and test or verify a certain system behavior. The general task of constructing a state is as follows: Starting from a particular known (and usually static) state s_{init}, find a sequence of state transformations $S = (tr_1, \ldots, tr_n)$ (imposed by, e.g., actions in a physical system or transitions in an automaton), such that the sequence leads to a known target state s_{target}, i.e., find an S with $S(s_{\text{init}}) = s_{\text{target}}$. In the easiest case, the system is *static*, a valid "reference" transformation sequence S_{ref} to the target state is *known* from observing a previous system execution, and the construction process is *unconstrained* in time. Then, we can simply replay the given sequence S_{ref} to reach the target state. However, systems are usually more complex and impose constraints that render the trivial approach inapplicable in many cases:

- **C1** The system may change *dynamically* over time (e.g., as some actions become inapplicable in a changed environment); a previously valid sequence that correctly reached the target state may become invalid or lead to a wrong state due to changed actions.
- **C2** The reference transformation sequence may be *unknown* (e.g., as the system actions are unobservable, or as we want to construct a state not reached by a previous execution); such a sequence needs to be manually constructed.

✉ Sascha Lehmann
s.lehmann@tuhh.de

Sibylle Schupp
schupp@tuhh.de

[1] Hamburg University of Technology, Hamburg, Germany

Published online: 08 September 2022

C3 The feasible time frame may be *constrained* (e.g., if the construction is embedded in an online setting for ongoing monitoring and thus frequently repeated); the restoring sequence needs to be limited both in terms of its length and the complexity of its transformations.

As the trivial approach requires access to the observed reference sequence, which grows linearly with system progress, and further assumes an unchanged system to apply the formerly valid sequence to, a conflict with all three constraints becomes clear, and alternative construction approaches are needed.

A prominent example of executable models that are frequently initialized to updated states—and the motivation of our work—is the field of *online model checking*. *Model checking* in general proves that specific system properties hold and provides guarantees on the correct behavior of a system, and for checking timed systems in particular, the modeling formalism of *timed automata (TA)* is commonly used, which represents time as (zones of) real-valued clocks. For *online* model checking, a technique that incrementally verifies or falsifies properties of a model under simulation for limited time scopes, a starting model state that reflects the state of the real system is required, so that one can continue model checking from that state on. In this article, we approach the state construction problem for TAs regarding the initially described system constraints, i.e., for potentially dynamic models in an online setting.

Adding online constraints to the overall construction task, one can distinguish two scenarios based on the availability of a reference sequence:

S1 If the reference sequence S_{ref} is given, we need to find a transformation $red = (red_1, \ldots, red_m)$ that reduces S_{ref} to a bounded sequence S, but still reaches the target state when applied to the initial state, i.e., find a red with $S \stackrel{\text{def}}{=} red(S_{\text{ref}})$, $|S| \leq bound$ and $S(s_{\text{init}}) = S_{\text{ref}}(s_{\text{init}}) = s_{\text{target}}$.

S2 If S_{ref} is not given, we need to determine a bounded state transformation sequence S that leads to the target state, based on the characteristics of the initial and target state and the supported actions A, i.e., find an $S = f(s_{\text{init}}, s_{\text{target}}, A)$ with $|S| \leq bound$ and $S(s_{\text{init}}) = s_{\text{target}}$.

Regardless of the concrete construction approach, one is usually tied to a limited set of operations and actions supported by a system to lead from its initial state to the target state. However, depending on the concrete field of application, the requirements and restrictions for intermediate states and transitions between the initial and target state differ: In case of physical systems, we can only use given actions applied in the scope of the system semantics, e.g., move an entity in defined directions. Consequently, the visited states have to lie in the original system state space, and the executed transitions need to correspond to the supported actions. For executable models such as those used in model checking, these strict requirements may be relaxed. Compared to a physical system, we may not be bound to existing states and transitions to reach the target state; in fact, the strict requirement is only imposed on a model if its system description cannot be altered, or if a physical system is already executed alongside the model during construction of the starting state. Otherwise, we can adapt the model by introducing new states and transitions which allow reaching the target state faster, or—in case of adaptive models—enable state construction in the first place. Instead of requiring original actions exclusively, we are only bound to a set of atomic, model-checker specific operations then.

A general problem is that a model checker may not allow us to set the concrete clock state of a TA directly by assignment. Our work uses the model checker *Uppaal*, a modeling and verification tool developed in a collaboration of the *Uppsala* and *Aalborg* universities. Here, the set of possible operations is limited to the reset of individual model clocks, constraining of clock differences, and time delays. Furthermore, all its simulations start at a well-defined clock state, where all clocks are initially set to 0. On the one hand, the assignment restriction guarantees that any clock state which the model takes is indeed valid and reachable in terms of the underlying semantics. On the other hand, the restriction raises the need for another strategy to set the system to a desired clock state. For an online model checking interface, which relies on a repeated manual manipulation of the clock state in bounded time to iteratively restore the most current state for the next verification run, such a strategy is mandatory.

We already discovered in the beginning that the trivial approach neither meets time constraints due to growing sequence lengths over time, nor can it be used if the model changes and the prior observed transformation sequence thus becomes inapplicable. For the case of a changeable model, Rinast [31] introduced a graph-based approach that keeps track of visited states and traversed transitions and forges new transitions as shortcut between existing states on the fly. Thus, the states belong to the original state space, while the intermediate transitions are potentially composed of segments of existing transitions. Even though the approach improves on the trivial result, the dependency on original states and transition segments still leads to ever-growing state construction sequences if no suitable shortcuts are found.

To tackle the sequence growth problem, we propose a new approach that utilizes both forged intermediate states and transitions to reach a desired starting state via DBM operation sequences of guaranteed bounded length. This article makes the following contributions:

1. We propose and prove a clock state construction approach for timed automata based on DBM overapproximation and constraining (which we call *OC approach*), which guarantees bounded lengths of DBM operation sequences S that depend quadratically on the number of system clocks T only (i.e., $0 \leq |S| \leq 1+2*|T|+|T|*(|T|+1)$), and covers both the scenarios of known (**S1**) and unknown (**S2**) reference sequences.
2. We provide an implementation of all OC approach variants as well as the trivial and graph-based Rinast approach for comparison, alongside an extended simulator for Uppaal timed automata which exposes the applied DBM operation sequences required by our approach on the fly.
3. We perform a comparison of the new approach with existing alternatives, i.e., the trivial and graph-based approaches.

The proposed overapproximation and constraint strategy derives a DBM operation sequence that first generates a superzone of the target state, and then constrains it until that state is reached.

An example of a model representing a simple process illustrates the state construction problem. The example model as shown in Fig. 1) has three locations (`On`, `Off`, and `Execute`) and two clocks (`t_active` and `t_exec`). The model, which is initially in `On`, can either perform an execution if the system state is not `critical`, with a duration $d \in [2, 5]$ given by the guard `t_exec >= 2` and the invariant `t_exec <= 5`, or else restart the system via `Off`. Consider a path starting in `On`, which traverses the `Execute` location ten times, turns `Off` once, and then visits `Execute` another ten times. An execution of this path, assuming all clocks are initially set to 0, leads to the following difference bound matrix (cf. Sect. 3.3 for the definition of DBM):

$$\begin{array}{c} \\ t(0) \\ t_{\text{active}} \\ t_{\text{exec}} \end{array} \begin{array}{c} \begin{array}{ccc} t(0) & t_{\text{active}} & t_{\text{exec}} \end{array} \\ \begin{pmatrix} 0 & -20 & -2 \\ 50 & 0 & 45 \\ 5 & -18 & 0 \end{pmatrix} \end{array} \tag{1.1}$$

Different reasons may require us to restore the concrete model state, assuming the system was already running for some time: We may want to verify that the system remains uncritical for another amount of time steps; then, we do not want to perform verification from the original initial model state on, as what lies in the past was verified already, but from the most current model state. Or we may want to set the upper bound for an execution step to a lower value (e.g., 4) according to real observations. As the system was previously running with time value 5, and the (verified) current state might not lie in the reachable state space for the adapted system anymore, we need to reach that state in another way.

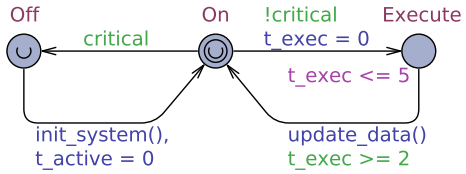


Fig. 1 Example process with two clocks (`t_active` and `t_exec`)

Setting the location and variable state is straight-forward; we can directly select *Init* locations which are active on model initialization, and set variables via direct assignments. The challenge though, as stated before, lies in the recovery of the clock state. If we had access to a trivial operation $SetValue$ that sets a specific entry of a DBM with system clocks T to any particular value and is directly callable in a model, we could simply set all clock state values on a single transition via at most $(|T|+1)^2$ applications of $SetValue$, and would be done. Unfortunately, such an operation is not supported by common TA model checkers due to their underlying semantics, and furthermore, the operations supported by a checker cannot be called directly and individually, as they are called in groups bound to the state and transition semantics of a TA. Thus, we have to compose or derive a supported sequence of DBM operations (and based on that, a corresponding sequence of automaton locations and edges)—which leads to this exact clock state DBM and does not require replaying the full execution path.

The article is structured as follows: We describe the related work in Sect. 2, followed by prerequisite knowledge of timed automata and DBMs in Sect. 3. Afterward, we introduce our approach in Sect. 4, where we refer back to the initial example, and describe the overapproximation and constraining phases of our approach in Sect. 5 and Sect. 6. Then, we transfer the approach to concrete Uppaal models in Sect. 7 and cover the tool implementation in Sect. 8, followed by the conducted experiments in Sect. 9. Finally, we provide a conclusion in Sect. 10.

2 Related work

In general terms, our work aims at the optimization of system state constructions, allowing model checkers like Uppaal to restore a given state more efficiently while relying on semantically supported operations only. Our work shares the motivation with fault tolerance and trace replay techniques commonly used to restore (past) system states, and draws technically on DBM operation sequences, their transformations, and DBM-based constraint solving.

In broad terms, our work contributes to research in state (re-)construction. Typically, research on this field can be found in debugging, testing, optimization, and in particular in

fault tolerance since the 1960s. In fault tolerance, one prominent technique is *checkpoint-recovery*, during which a system on failure is reinitialized to a complete snapshot and optionally updated along several incremental checkpoints. Over the past decades, it was used for state recovery both in software (e.g., for databases [28] and shared memory multiprocessor systems [38]) and cyber-physical hardware applications [22]. Compared to our approach, while often targeting entirely different domains unrelated to model checking (even though variants of checkpoint recovery are also used in model checking, e.g., for non-explicit storage of states in colored Petri nets [14]), such a technique is used to reduce the memory and runtime overhead to reach a certain state. For instance, using context-aware [25] and online [40] variants of checkpoint derivation, or optimized techniques such as two-state checkpointing in hard real-time systems [33], one tries to reduce the number and extent of checkpoints during recovery. For all these techniques, the initial state is the snapshot of the full state, which the system can be directly assigned to, while the target state is the one before the failure occurred, reachable via a sequence of incremental checkpoints and additional instructions after the final checkpoint. In our approach, the initial state is the starting state specified by the concrete model checker, and the target state is the most recently simulated state which we want to recover.

Other techniques, such as *full-system-restart* applied in cyber-physical systems [18] or *software rejuvenation* [17], rely on a restart of the affected system. The initial state then becomes the system state directly after the restart, and the goal remains getting back to the latest error-free state within limited time. The differences to our work are the application domain (these techniques are usually applied to concrete programs and actuator systems rather than abstract system models), the requirements (e.g., full restarts are only feasible in applications with non-exponential state spaces to keep the recovery time bounded), and the absence of clock state abstractions (i.e., clock zones) as used in timed automata.

A major challenge to make the aforementioned techniques feasible in practice is to find suitable reduction strategies for the involved instructions, so that not all original actions need to be replayed on recovery. Therefore, aside from fault tolerance, different forms of state reconstruction are used in software optimization, debugging, and testing. The aspect of instruction sequence reduction can be found, among others, in the field of *source code analysis and optimization* [9]. A common aspect of such an optimization is the identification of shorter instruction sequences to a specific state by omitting redundant instructions, or those leading to unread intermediate states. Likewise, in testing applications, traces of instructions observed during simulation are optimized for the purpose of reduced replay times or memory space usage, e.g., during virtual memory simulations [21].

For the problem of clock state (re-)construction in particular, only few works can be found. Closely related, in the domain of timed automata, Rinast approaches the recovery problem under the term *state space reconstruction* [31]. In his work, he constructs "shortcut" transitions to states observed during simulation, i.e., transitions from which DBM operations are removed that were overwritten by subsequent operations and thus were rendered redundant. As main use case of his work, applied in multiple medical case studies [30], he used the approach to reconstruct a target state via such shortcut transitions, visiting only a reduced (and preferably bounded) number of intermediate states. The approach requires that resets of all clocks in all model cycles exist to guarantee bounded (re-)construction lengths. Our approach omits this requirement and allows the construction of a specific state via a finite amount of transitions linearly proportional to the number of system clocks (cf. Sect. 7).

Several model checking techniques either rely on, or may benefit from, clock state (re-)construction techniques: *Incremental model checking* [27] verifies multiple iterations of a system design, and aims for reusing portions of previous checking results. Restoring a state, which is visited in one system, in another candidate system directly before reaching a differing component (compared to the former system), may reduce the time required to check incrementally designed system candidates. As the candidates are individual systems, a trivial transfer of one system state to another system is not generally possible, and thus, the state needs to be reconstructed. *Bounded model checking* [10] reduces the general model checking approach to limited scopes of k steps, and was applied to timed systems [4], including timed automata [37], in the past. Building on this boundedness, *online model checking* [39] verifies a system iteratively for limited future time scopes and derives temporarily valid guarantees. With optimized state (re-)construction techniques, we can faster restore the most recently verified state that fits new observations, from which another verification iteration is then started; otherwise, the recovery of such a state would require growing amounts of time.

Technically, we draw on DBM operation sequences, sequence transformations, and constraint solving. The concept of difference bound matrices (DBMs), the data structure which is used, e.g., for timed automata to represent the clock state, was covered in detail by Bengtsson et al. [8]. Our approach works on such a representation of clocks to restore the overall clock state. For that, it applies algebraic transformations to DBM operation sequences to reach DBM overapproximations and eliminate redundant constraints. Different DBM overapproximation techniques were already used in the literature, e.g., for the convex hull calculation of state unions [32] or zone extrapolation [6] in Uppaal, or for compact graph construction of real-time preemptive Petri net systems [1]. Compared to these techniques, tightness is

not required by our approach, as the constraint phase reaches a target DBM independent of the concrete overapproximation; however, more efficient constraint sequences may result from tightness requirements.

Our DBM_{target}-based overapproximation approach relies on constraint solving, and, more precisely, the selective discarding of constraints. In model checking, various data structures such as difference bound matrices (DBMs) [8], clock difference diagrams (CDDs) [24], or the more recent constraint matrix diagrams (CMDs) [13] are used to span the clock state by a set of constraints. Constraint solving can then, among others, be applied to such structures to manipulate the state space symbolically [29] and obtain valid clock value assignments, or to temporal logic formulae [15] to verify system properties. Furthermore, constraints need to be relaxed in certain cases to deal with inconsistencies of constraint systems, e.g., for constraint networks [16] or during model repair [3], often with the goal of minimal relaxations or consideration of constraint priorities or uncertainties in real world systems [12]. In our case, the relaxed constraints do not necessarily need to be minimal, but the constraints cannot be relaxed independent of each other, as they are bound to certain DBM operations (e.g., the future delay) semantically.

The constraint phase of our approach derives suitable sets of constraints that reduce the former overapproximated zone to the zone of the target DBM. An efficient approach is the use of minimal constraint systems (MCS), a technique introduced by Larsen et al. [23] for timed automata that is based on the transitive reduction in directed graphs [2] and commonly used as a compact representation of DBM data. Our approach builds upon the idea of MCS and extends it to regard constraints that are already fulfilled by the overapproximating DBM, which eliminates redundancy and results in shorter operation sequences.

In regard to tool dependencies and support, our work uses models for the model checking tool Uppaal [7]. Its implementation and formalisms impose specific requirements for our work, i.e., an operation-based construction of DBMs and a defined clock space initialization (all clocks set to 0 initially), respectively. The applicability of our approach is not limited to that particular model checker, though.

3 Automata and DBM prerequisites

In this section, we provide preliminary details on the definitions of the syntax and semantics of timed automata (Sect. 3.1), the model state (Sect. 3.2), DBMs and their operations (Sect. 3.3), the graph representation of DBMs (Sect. 3.4), sequences of DBM operations (Sect. 3.5), and a method for flow dependency analysis on DBM data (Sect. 3.6).

3.1 Timed automata

Our work uses timed automata as underlying modeling formalism. We define the syntax of timed automata as follows:

Definition 1 *(TA-Syntax)* A timed automaton (TA) is a tuple $\langle L, l_0, C, E, g, r, I\rangle$, where L is a finite set of locations, l_0 is the initial location, C is a finite set of (real-valued) clocks, $E \subseteq L \times L$ is a set of edges between locations, $g : E \to \Phi(C)$ is a mapping from edges to guards, $r : E \to R(C)$ is a mapping from edges to partial resets, and $I : L \to \Phi(C)$ is a mapping from locations to invariants. The set $\Phi(C)$ contains all possible conjunctions over constraints $t_a <> c$ and $t_a - t_b <> c$, with $t_a, t_b \in C$, $c \in \mathbb{N}$, and $<> \in \{<, \leq, =, \geq, >\}$. $R(C) = [C \rightharpoonup \mathbb{N}_0]$ denotes the set of all partial functions $\rho : C \rightharpoonup \mathbb{N}_0$, where each ρ represents a right-unique mapping from a subset of clocks to reset values (i.e., any natural number including 0, where the latter is the usual reset value).

Both edge guards and location invariants express constraints on the valuations of clocks C, which control when an edge can eventually be triggered, and for how long a location may remain active, respectively. Extending the TA formalism, an extended timed automaton (ETA) is a tuple $\langle L, l_0, C, V, E, g, r, I, a, L_U, L_C, Ch_{Bi}, Ch_{Br}\rangle$, where V is a set of variable, $a : E \to A(V)$ is a mapping from edges to variable assignments, L_U and L_C are urgent and committed locations, and Ch_{Bi} and Ch_{Br} are binary and broadcast channels. We define $A(V) = [V \rightharpoonup \mathbb{B} + \mathbb{Z}]$ as the set of all partial assignment functions $\alpha : V \rightharpoonup \mathbb{B} + \mathbb{Z}$, where each α represents a right-unique mapping from a subset of variables to boolean or integer values.

Commonly, the semantics of TAs is defined via real-valued delays on transitions in a labeled transition system, which results in an uncountable state space. Using a zone-based abstraction (via convex polyhedra of clock constraints), one can reduce the state space to a finite set. We use the latter for our work and thus define the symbolic TA semantics similar to the definition of Behrmann et al. [5] and adapted for $\mathbb{N}_0$-resets, zero-initialized zones, and included variable states as follows:

Definition 2 *(TA-Symbolic Semantics)* Let $Z_0 = \bigwedge_{x \in C} x = 0$ be the initial zone, and $\forall x \in V : u_{v,0}(x) = 0$ be the initial variable valuation. The symbolic semantics of a timed automaton $\langle L, l_0, C, V, E, g, r, I, a\rangle$ is defined as a transition system $(S, s_0, \Rightarrow)$, called the simulation graph, where $S = L \times \Phi(C) \times U_v$ is the set of symbolic states, $s_0 = (l_0, Z_0 \wedge I(l_0), u_{v,0})$ is the initial state, $\Rightarrow = \{(s, s') \in S \times S \mid \exists e, t : s \overset{e}{\Rightarrow} t \overset{\delta}{\Rightarrow} s'\}$ is the transition relation, with:

- $(l, Z, u_v) \overset{\delta}{\Rightarrow} (l, (Z \wedge I(l))^{\uparrow} \wedge I(l), u_v)$

– $(l, Z, u_v) \overset{e}{\Rightarrow} (l', r_e(g(e) \wedge Z \wedge I(l)) \wedge I(l'), [a(e)]u_v)$ if $e = (l, l') \in E$,

where $Z^{\uparrow} = \{u+d \mid u \in Z \wedge d \in \mathbb{R}_{\geq 0}\}$ (the *future* operation), and $r_e(Z) = \{[r(e)]u \mid u \in Z\}$ (the *reset* operation). $u : C \to \mathbb{R}_{\geq 0}$ is a clock valuation function (note that the definitions consider guards and invariants as sets of clock valuations by abuse of notation), $u + d$ maps each clock $x \in C$ to the value $u(x) + d$, $d \in \mathbb{R}_{\geq 0}$, and $[r(e)]u$ denotes the clock valuation which maps a subset C_r of clocks C to natural numbers as defined by $r(e)$, and agrees with u over $C \backslash C_r$. Likewise, $u_v : V \to \mathbb{B} + \mathbb{Z}$ is a variable valuation function (where U_v is the set of all variable valuations), and $[a(e)]u_v$ denotes the variable valuation which maps a subset V_a of variables V to boolean or integer numbers as defined by $a(e)$, and agrees with u_v over $V \backslash V_a$.

For a definition of the semantics of networks of timed automata and channel synchronization, see [7].

3.2 Model state

An execution trace of a model is a sequence of transitions. During such an execution, each transition leads to a concrete *model state*. A state of an ETA consists of three components: (a) The location state covering the currently active location of each sub-automaton, (b) the variable state spanning all global and local `integer`, `bool`, etc. valuations that are used by components of the ETA, and (c) the clock state including all clock differences (e.g., for our work, represented as a DBM). The construction of a location and variable state is easily done: The targeted location can be set as initial location (l_{init}), and will be active on model initialization. The variables are set directly to their targeted values by assignment. The clock state, in contrast, needs to be constructed via a sequence of invariants, guards, resets, and other operations defined in the following sub-section and is the focus of this paper.

3.3 DBM and operations

Clocks can take values from a given interval, which is defined by the previous invariants, guards, and resets along a path through the model. In case of TAs, the concrete value intervals of these clocks are represented as DBMs [8], which have the following structure:

$$\begin{array}{c} \\ t(0) \\ t_1 \\ \vdots \\ t_n \end{array} \begin{array}{c} \begin{array}{cccc} t(0) & t_1 & \cdots & t_n \end{array} \\ \begin{pmatrix} 0 & c_{01} & \cdots & c_{0n} \\ c_{10} & 0 & \cdots & c_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n0} & c_{n1} & \cdots & 0 \end{pmatrix} \end{array} \tag{3.1}$$

In a DBM, the matrix entry in the row of clock t_i and column of clock t_j represents the upper bound of their clock value difference, i.e., $t_i - t_j \leq c_{ij}$, and the diagonal entries are naturally 0 (as $t_i - t_i = 0$). Note that a reference clock $t(0)$ is added, which has a constant value of 0, to turn any single clock constraint into a two-clock constraint $t_i - t(0) \leq c$ or $t(0) - t_i \leq c$. We denote the set of clocks of a DBM as $T_0(DBM)$ if the reference clock is included, and as $T(DBM)$ otherwise. Altogether, such a DBM represents the complete clock state.

The symbolic definition of TA semantics provides three types of operations that are applied to clock zones over transitions in the simulation graph: The $\uparrow$ operation delays the clock arbitrarily to include all futures, the logical $\wedge$ operation adds constraints to the clock zone, and the r_e operation resets selected clocks to natural numbers. In the context of DBMs in the scope of our work, we call these operations $DelayFuture$, $Constraint$, and $Reset$, respectively. A fourth operation is $Close$, which determines the tightest constraints representing the original zone, i.e., the transitive closure of constraints. While not required by the base semantics, that operation is commonly applied by model checkers after adding constraints to the zone, enabling a more efficient application of subsequent operations. The full set of operations applied during transitions through a model [5] is thus

$$\begin{aligned} OP = \{&DelayFuture, Reset(t_a, v), \\ &Constraint(t_a, t_b, v), Close(t_a, t_b)\}, \end{aligned} \tag{3.2}$$

which we will abbreviate as DF, R, C, and Cl, respectively, where appropriate. On the level of DBM transformations, the operations are formally defined as follows for a DBM with n clocks ($\forall i, j \in \mathbb{N} \cap [0, n]$):

$$DF(DBM)[i, j] = \begin{cases} \infty & \text{if } i \neq 0 \wedge j = 0 \\ DBM[i, j] & \text{otherwise} \end{cases} \tag{3.3}$$

$$R(t_a, v)(DBM)[i, j] = \begin{cases} DBM[i, 0] - v & \text{if } i \neq a \wedge j = a \\ DBM[0, j] + v & \text{if } i = a \wedge j \neq a \\ DBM[i, j] & \text{otherwise} \end{cases} \tag{3.4}$$

$$C(t_a, t_b, v)(DBM)[i, j] = \begin{cases} min(DBM[i, j], v) & \text{if } i = a \wedge j = b \\ DBM[i, j] & \text{otherwise} \end{cases} \tag{3.5}$$

$$\begin{aligned} Cl(t_a, t_b)(DBM)[i, j] = \min(&DBM[i, j], \\ &DBM[i, a] + DBM[a, b] + DBM[b, j]) \end{aligned} \tag{3.6}$$

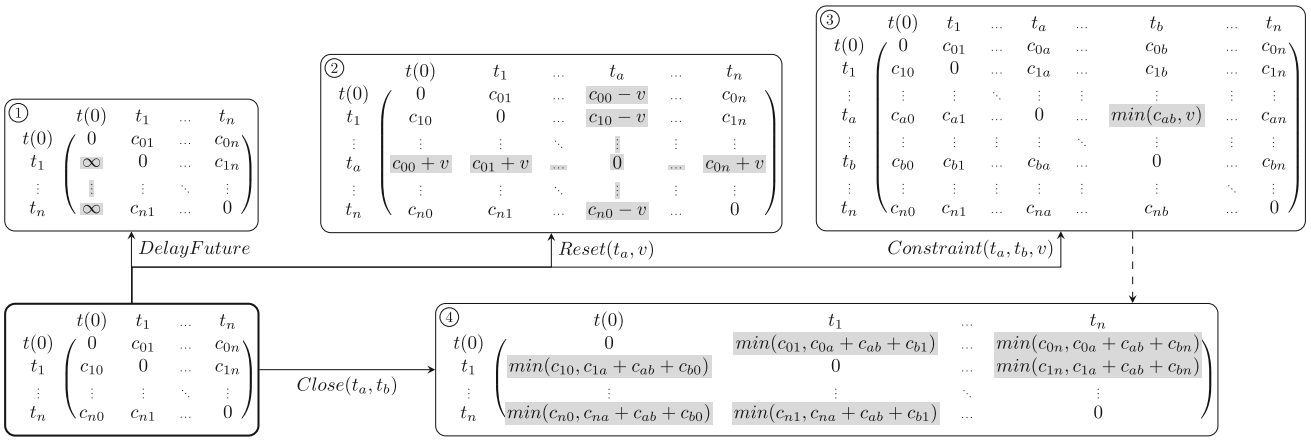


Fig. 2 An overview of all DBM operations $op \in OP$

A graphical overview of the operations in OP (Eq. 3.2) is shown in Fig. 2. All clocks are infinitely delayed into the future whenever a new (neither urgent nor committed) set of locations is reached, before the new invariant constraints of the target locations can be applied. The **DelayFuture** operation removes the upper bound of all DBM clocks by setting their DBM entries in the first column to ∞ (see Fig. 2(1)).

The **Reset**($\mathbf{t_a}$, **v**) operation resets a single clock t_a to the value v. The operation adapts all DBM entries in the row and column of t_a (see Fig. 2(2)), i.e., the difference between that clock and all other clocks, according to the reset value.

The **Constraint**($\mathbf{t_a}$, $\mathbf{t_b}$, **v**) operation constrains the DBM zone by updating a single DBM entry $DBM[t_a, t_b]$ to the minimum of its value and v (see Fig. 2(3)). As $Constraint$ is the only operation in OP which does not preserve the closedness of the input DBM zone, a following $Close(t_a, t_b)$ call is required.

The **Close**($\mathbf{t_a}$, $\mathbf{t_b}$) operation transforms the DBM to its closed form (see Fig. 2(4)) in case that only one single constraint operation $Constraint(t_a, t_b, v)$ was applied beforehand (cf. [11]). A general form, $Close()$, exists, which recalculates all shortest paths between pairs of clocks from scratch. For a more compact presentation of the construction sequences, we will use a single **Close** operation after sequences of constraints instead of selective **Close**($\mathbf{t_a}$, $\mathbf{t_b}$) operations after each constraint.

3.4 DBMs and graphs

A DBM can be represented as weighted complete digraph [8] as shown in Fig. 3. In such a graph, each edge (t_i, t_j) represents the clock difference $t_j - t_i$, and the edge weight represents the value of $DBM[j, i]$. Self-edges, which represent the diagonal DBM entries with 0 weight, are usually omitted. The weight of a path is the sum of weights of edges included in the path. Three facts become important

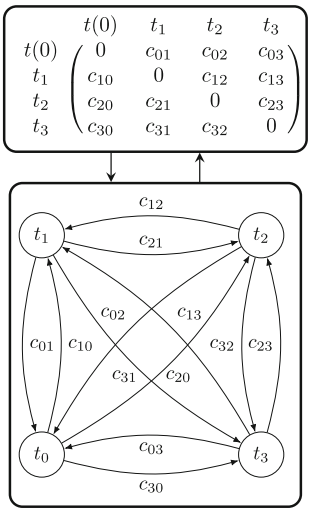


Fig. 3 Transformation between DBM and graph

in the graph-based version of our DBM overapproximation approach:

Fact 1 (Emptyness and weights) *A DBM represents a non-empty zone iff its graph G has no negative-weight cycles* [8].

Fact 2 (Minimum edge costs) *Given a DBM, each edge $e = (t_i, t_j)$ in its graph G has minimum edge costs* [23], *i.e., given a weight function w, $w(e) \leq w(p)$ holds for each longer path $p = ((t_i, t_k), \ldots, (t_l, t_j))$ from t_i to t_j, iff that DBM is in closed form (i.e., represents the transitive closure for a set of constraints).*

Fact 3 (Cycles with ∞-weight edges) *The total path cost of a cycle with at least one ∞-weight edge is ∞.*

Finally, we point out the following path types:

A **simple cycle** in a digraph is a cycle where no vertex except for the start vertex is repeated (as the cycle starts and ends in the same vertex).

A **Hamiltonian path** is a path that visits each vertex of a graph exactly once.
A **Hamiltonian cycle** is a simple cycle over all vertices of a graph.
A **cycle chord** of a cycle in a digraph is an edge between two (non-immediately) consecutive vertices of the cycle, i.e., a "short-cut" between two cycle vertices, such as (t_1, t_3) for the cycle (t_1, t_2), (t_2, t_3), (t_3, t_1).
A **path prefix** of a path p is a subpath starting with the same vertex as p.
A **prefix cycle** of a cycle c is a subcycle over a path prefix of c followed by an edge to the start vertex of c.
An **all-positive-prefix path** p in a weighted digraph is path in which all path prefixes (including p itself) have positive weights.

3.5 DBM operation sequences

We denote the sets of all possible R, C, and Cl operations as follows:

$$\begin{aligned}\mathcal{R}(DBM) &= \{R(t_a, v) \mid t_a \in T(DBM), v \in \mathbb{N}_0\}\\ \mathcal{C}(DBM) &= \{C(t_a, t_b, v) \mid t_a, t_b \in T_0(DBM), v \in \mathbb{Z}\}\\ \mathcal{CL}(DBM) &= \{Cl(t_a, t_b) \mid t_a, t_b \in T_0(DBM)\}\\ \mathcal{OP}(DBM) &= \mathcal{R}(DBM) \cup \mathcal{C}(DBM) \cup \mathcal{CL}(DBM)\\ &\quad \cup \{DF\}\end{aligned}$$

Multiple operations form a sequence, which we denote as $S = (op_1, op_2, \ldots, op_n)$, with $op_i \in \mathcal{OP}(DBM)$, and for which we define its application to a DBM based on function composition as:

$$apply(S, DBM) = (op_n \circ op_{n-1} \circ \ldots \circ op_1)(DBM). \quad (3.7)$$

We define the merging of two operation sequences $S_1 = (op_1, \ldots, op_m)$ and $S_2 = (op_{m+1}, \ldots, op_n)$ as

$$S_1 \oplus S_2 = (op_1, \ldots, op_m, op_{m+1}, \ldots, op_n). \quad (3.8)$$

Furthermore, we define the set of all merged sequences over the cartesian product of two sequence sets $\mathcal{S}_a = \{S_{a,1}, \ldots, S_{a,n}\}$ and $\mathcal{S}_b = \{S_{b,1}, \ldots, S_{b,m}\}$ as

$$\mathcal{S}_a \otimes \mathcal{S}_b = \bigcup_{(S_1, S_2) \in \mathcal{S}_a \times \mathcal{S}_b} S_1 \oplus S_2, \quad (3.9)$$

and, based on $\otimes$ (written as $\otimes_i$ indicating the i-th occurrence of the operator), define for a single sequence set $\mathcal{S}$

$$\bigotimes_k \mathcal{S} = \mathcal{S} \otimes_1 \mathcal{S} \otimes_2 \ldots \otimes_{k-1} \mathcal{S}. \quad (3.10)$$

Finally, we denote the subsequence relation between a sequence $S = (x_1, ..., x_n)$ and a subsequence $S' = (x_{i_k})_{k \in [1,n]}, i_1 < i_2 < \ldots$ as $S' | S$.
For a single operation op, we define

$$\begin{aligned} op^1 &= \{(op)\}, & op^* &= \{(), (op), (op, op), \ldots\}\\ op? &= \{()\} \cup op^1 & op^+ &= \{(op), (op, op), \ldots\}\end{aligned}$$

for sets of sequences where op is applied once (op^1), zero or more times (op^*), zero or one time ($op?$), or one or more times (op^+), and for a set OP of operations, we define

$$\begin{aligned} OP^1 &= \{(op_1), \ldots, (op_n)\}, op_i \in OP\\ OP^* &= \bigcup_{i=0}^{\infty} \bigotimes_i OP^1, \quad OP^+ = \bigcup_{i=1}^{\infty} \bigotimes_i OP^1\\ OP? &= \{(), (op_1), \ldots, (op_n)\} = \{()\} \cup OP^1\end{aligned}$$

to denote the sets of sequences S with length $|S| = 1$ (OP^1), $|S| \geq 0$ (OP^*), $|S| \geq 1$ (OP^+), or $|S| \in [0, 1]$ ($OP?$).

Finally, we define the following particular types of sequences which become relevant in our approach:

- A **reset-all** sequence for a DBM is a sequence $S \in \mathcal{R}(DBM)^*$ in which each clock $t \in T(DBM)$ is reset exactly once.
- A **reset-df-all** sequence for a DBM is a sequence $S \in (\mathcal{R}(DBM)^1 \otimes DF^1)^*$ in which each clock $t \in T(DBM)$ is reset exactly once, and each reset is followed by a DF.
- A **reset-(df)-all** sequence for a DBM is a sequence $S \in (\mathcal{R}(DBM)^1 \otimes DF?)^*$ in which each clock $t \in T(DBM)$ is reset exactly once, and each reset may be followed by a DF; note that the sets of reset-all and reset-df-all sequences are subsets of the set of all reset(-df)-all sequences.

In our O(DBM) approach (cf. Sect. 5.2), a major focus lies on the order of resets in operation sequences. In general, the two DBM entries $DBM[i, j]$ (representing $t_i - t_j \leq DBM[i, j]$) and $DBM[j, i]$ (representing $t_j - t_i \leq DBM[j, i]$, i.e., $t_i - t_j \geq -DBM[j, i]$), span the interval of possible difference values $t_i - t_j \in [-DBM[j, i], DBM[i, j]]$. For such an interval, the following fact holds:

Fact 4 (Order of clock difference intervals) *Given a DBM reached via an operation sequence $S \in \mathcal{OP}(DBM)^*$, for each pair of clocks $t_i, t_j \in T(DBM)$, with $t_i - t_j \in [l, u]$, lower bound $l = -DBM[j, i]$, and upper bound $u = DBM[i, j]$, we can distinguish between four cases:*

$$l \geq 0 \wedge u > 0, \quad \textit{iff } t_i \textit{ reset before } t_j \quad (3.11)$$

$$l < 0 \wedge u \leq 0, \quad \textit{iff } t_i \textit{ reset after } t_j \quad (3.12)$$

$$l = 0 \wedge u = 0, \quad \textit{iff } t_i, t_j \textit{ reset ``simultaneously''} \tag{3.13}$$
$$l < 0 \wedge u > 0, \quad \textit{iff order ``not explicitly known''}, \tag{3.14}$$

where "reset before" and "reset after" imply that a potential delay happened between both resets, while "simultaneously" means without intermediate delay.

Note that the fourth case can only occur if we reset to values other than 0, as only then, a reset may set a clock to a value greater than another clock which was actually reset before, so that the reset order is "not explicitly known" from the interval bounds alone.

For DBMs reached via reset-df-all sequences (as shown in Fig. 4), we note that each entry $DBM[i, j]$ becomes either $v_i - v_j$ or ∞ based on the reset order, and the following fact holds:

Fact 5 (DBM value after reset-df-all sequence) *Given a DBM with n clocks ($\forall i, j \in \mathbb{N} \cap [0, n]$), a reset-df-all sequence S sets each entry $DBM[i, j]$ to $v_i - v_j$ for which t_i is reset after t_j, and to ∞ otherwise:*

$$S(DBM)[i, j] = \begin{cases} v_i - v_j & \textit{if } t_i \textit{ reset after } t_j \\ \infty & \textit{otherwise} \end{cases} \tag{3.15}$$

Proof Fact 5 can be proven with the sets $wr(DF)$ and $wr(R(t_a, v))$ (see Eqs. 3.16 and 3.17 in Sect. 3.6). An entry $DBM[i, j]$, $i, j \neq 0$ is overwritten twice by a reset-df-all sequence; once by $R(t_i, v_i)$, for which the entry $DBM[i, j]$ lies in the row of t_i and is overwritten with the value $DBM[0, j] + v_i$, and once by $R(t_j, v_j)$, for which $DBM[i, j]$ lies in the column of t_j and is overwritten with the value $DBM[i, 0] - v_j$. We also notice that the DF operations set the values $DBM[i, 0], i \neq 0$, to ∞ between the resets. It follows that if t_i is reset after t_j, $DBM[i, j]$ becomes $DBM[0, j] + v_i$ by $R(t_i, v_i)$, with $DBM[0, j] = -v_j$ due to the previous reset $R(t_j, v_j)$, resulting in $DBM[i, j] = v_i - v_j$. Conversely, if t_i is reset before t_j, $DBM[i, j]$ becomes $DBM[i, 0] - v_j$ by $R(t_j, v_j)$, with $DBM[i, 0] = \infty$ due to the previous DF, resulting in $DBM[i, j] = \infty$. □

For the relation between reset orders in a reset-df-all sequence and paths in a graph G of a DBM, we note the following facts:

Fact 6 (Hamiltonian cycles and reset order) *For a DBM and its graph G, a one-to-one relation exists between Hamiltonian cycles p_H in G and total reset orders of clocks $T(DBM)$.*

Fact 7 (∞-edges after reset-df-all sequence) *Given a DBM and its graph G, a reset-df-all sequence sets all edges $(t_i, t_j) \in E(G)$ to ∞ for which the clocks t_i and t_j are not in reset order.*

Fact 8 (Hamiltonian path over non-∞ edges) *Given a DBM and its graph G resulting from a reset-df-all sequence, only the Hamiltonian path p_H over all vertices $t_i \in V(G)$ in reset order and all its sub-paths via cycle chords traverse no edges with ∞ costs.*

Proof Fact 6 holds as in a complete digraph with exactly one edge per ordered vertex pair, only one unique cycle exists that traverses all vertices in a particular order. □

Proof Fact 7 directly follows from Fact 5. □

Proof Fact 8 holds as all other paths contain at least one edge in reverse reset order, for which Fact 7 holds. □

3.6 Flow dependency of DBM operations

To prove that particular operations in a sequence can be omitted without changing the resulting DBM, we need to analyze at which points individual DBM entries are read or written. For that, we define the function $rd : \mathcal{OP}(DBM) \to \mathbb{N} \to \mathbb{N} \to 2^{\mathbb{N}\times\mathbb{N}}$, which returns the read set of an operation op for a particular entry $DBM[i, j]$, i.e., the set of index pairs (a, b) of all entries $DBM[a, b]$ read by op for the calculation of $DBM[i, j]$. Furthermore, we define the function $wr : \mathcal{OP}(DBM) \to 2^{\mathbb{N}\times\mathbb{N}}$, which returns the write set of the operation op, i.e., the set of index pairs (a, b) of all entries $DBM[a, b]$ written to by op. For the supported set of operations $OP = \{DF, R(t_a, v), C(t_a, t_b, v), Cl(t_a, t_b)\}$, rd and wr are defined as follows:

$$rd(DF, i, j) = \emptyset$$
$$wr(DF) = \{(i, 0) \mid i \in \mathbb{N} \cap [1, n]\} \tag{3.16}$$
$$rd(R(t_a, v), i, j) = \begin{cases} \{(i, 0)\} & \text{if } i \neq a \wedge j = a \\ \{(0, j)\} & \text{if } i = a \wedge j \neq a \\ \emptyset & \text{otherwise} \end{cases}$$
$$wr(R(t_a, v)) = \{(i, a) \mid i \in \mathbb{N} \cap [0, n], i \neq a\} \cup \{(a, j) \mid j \in \mathbb{N} \cap [0, n], j \neq a\} \tag{3.17}$$
$$rd(C(t_a, t_b, v), i, j) = \begin{cases} \{(i, j)\} & \text{if } i = a \wedge j = b \\ \emptyset & \text{otherwise} \end{cases}$$
$$wr(C(t_a, t_b, v)) = \{(a, b)\} \tag{3.18}$$
$$rd(Cl(t_a, t_b), i, j) = \begin{cases} \{(i, j), (i, a), (a, b), (b, j)\} & \text{if } i \neq j \\ \emptyset & \text{otherwise} \end{cases}$$
$$wr(Cl(t_a, t_b)) = \{(i, j) \mid i, j \in \mathbb{N} \cap [0, n], i \neq j\} \tag{3.19}$$

Based on the rd and wr functions, we define the flow dependency function $fd : \mathcal{OP}(DBM) \to 2^{\mathbb{N}\times\mathbb{N}} \to 2^{\mathbb{N}\times\mathbb{N}}$, which determines how the dependencies from previous oper-

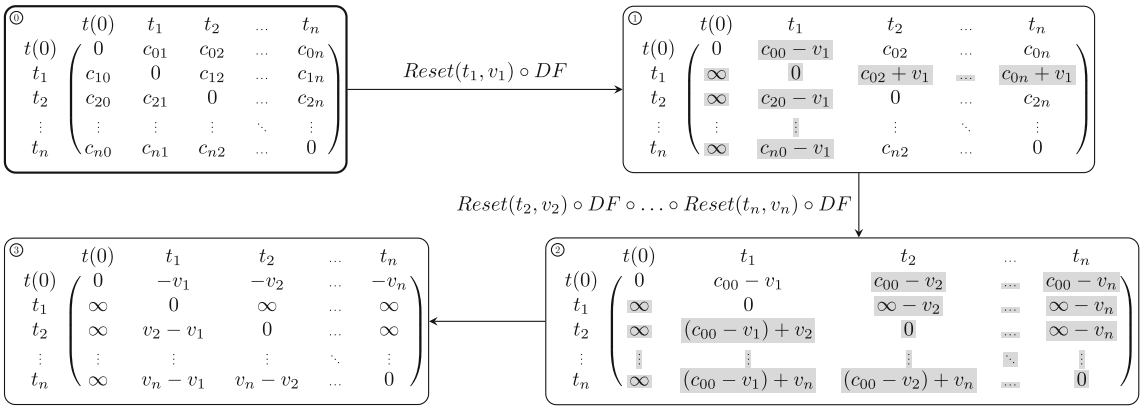


Fig. 4 A sequence of all clock resets with intermediate future delays

ations are forwarded by an operation op:

$$fd(op, I) := (I \backslash wr(op)) \cup I_{op}, \tag{3.20}$$
$$\text{with } I_{op} = \{(i, j) \mid (i, j) \in wr(op), rd(op, i, j) \cap I \neq \emptyset\}$$
$$fd(S, I) := (fd(op_n) \circ \ldots \circ fd(op_1))(I) \tag{3.21}$$

In other words, to determine how the dependency on a set I of marked index pairs is maintained by an operation op, the function fd removes all index pairs from I whose DBM values are overwritten by op (i.e., $I \backslash w(op)$), and (re-)adds all those index pairs (i.e., the set I_{op}) whose new DBM values calculated by op depend on marked entries of the original I. That way, we get a new set of marked index pairs based on the original marked indices and the dependencies from these indices forwarded by op. To determine if an operation op_1 can be omitted from a sequence S, we initialize I with the write set of op_1, and recursively apply fd to I for each following operation until the resulting index set becomes $\emptyset$.

To prove flow independency on infinite sets of operation sequences composed from a finite set of operations as required in Sect. 5.1.2, we use the following properties:

$$\forall op \in OP : I_1 \subseteq I_2 \implies fd(op, I_1) \subseteq fd(op, I_2) \tag{3.22}$$

$$\forall S \in \mathcal{OP}(DBM)^* : I_1 \subseteq I_2 \implies fd(S, I_1) \subseteq fd(S, I_2), \tag{3.23}$$

from which follows that if a set of marked indices becomes $\emptyset$ after applying some sequence S, a subset of that index set will certainly become $\emptyset$ as well after applying S. Thus, it is sufficient to check the superset only.

4 DBM-based clock state construction

The main task of this work is to determine a sequence of operations that allows reaching an originally observed model clock state. In this section, we describe the general idea of our approach (Sect. 4.1), allowing state construction in limited time, and revisit the example from the introduction to compare concrete state construction approaches (4.2).

4.1 Approach description

In Sect. 1, we introduced the general state construction task, i.e., finding a sequence S that transform a starting state s_{init} into a target state s_{target}, as well as the boundedness requirement $|S| \leq bound$ for some strict $bound$ imposed by online settings. Our approach splits that state construction task into two phases, the O-phase and the C-phase, where the former constructs an overapproximation of the target state, and the latter constraints that overapproximation until it equals the target state:

- **O-Phase**: Given a starting state s_{init} and a target state s_{target}, determine a bounded sequence S_{approx} that transforms s_{init} into an overapproximation of s_{target}, i.e., find an S_{approx} with $S_{\text{approx}}(s_{\text{init}}) \supseteq s_{\text{target}}$.
- **C-Phase**: Given a target state s_{target} and an overapproximating state $s_{\text{approx}} \supseteq s_{\text{target}}$, determine a bounded sequence S_{constr} that transforms s_{approx} into s_{target}, i.e., find an S_{constr} with $S_{\text{constr}}(s_{\text{approx}}) = s_{\text{target}}$.

The main advantage of this constructive approach is, as we will show later, that it is possible to generate operation sequences with bounded lengths for both phases, which do not grow throughout a simulation, but keep a constant length depending only on the number of clocks involved.

Combining both phases leads to a formulation of the overall state construction task for our approach:

- **Bounded state construction via OC-approach**: Given a starting state s_{init} and a target state s_{target}, determine a bounded sequence $S = S_{\text{approx}} \oplus S_{\text{constr}}$ that transforms

s_{init} into s_{target}, where S_{approx} leads to an overapproximation of s_{target} which is then constrained by S_{constr} to s_{target}, i.e., find an $S = S_{\text{approx}} \oplus S_{\text{constr}}$ with $|S| \leq bound$, $S_{\text{approx}}(s_{\text{init}}) \stackrel{\text{def}}{=} s_{\text{approx}} \supseteq s_{\text{target}}$, $S_{\text{constr}}(s_{\text{approx}}) = s_{\text{target}}$, and thus, $S(s_{\text{init}}) = s_{\text{target}}$.

We mentioned in Sect. 3 that setting the variable and location state is trivially solved for *Uppaal* TAs, and thus, our approach is focused on the construction of the clock state. Thus, in our concrete case of state construction for clock states in TAs, each state s represents a DBM clock state, and the sequences S are composed from a limited set of operations OP supported by the TA formalism, i.e., we apply $s = DBM$ and require $S \in \mathcal{OP}(DBM)^*$.

4.2 Introductory example

Recall the example model of a simple process in Fig. 1, as well as the sample trace and reached DBM (i.e., the target clock state) described in Sect. 1. In this sub-section, we show the differences between possible clock-state construction approaches, i.e., the trivial approach, the graph-based approach [30], and our OC approach.

For the example, applying the trivial approach leads to unbound sequences along all model paths on the long run. The construction sequence is equal to the executed model path and thus grows linearly with the amount of transitions taken. The graph-based approach by Rinast keeps track of all model states that were reached, and finds shortcuts between these states, which can be taken as replacement for longer paths leading to the exact same state during construction. The approach limits the size of the construction path as long as at some point, re-reaching a previous state is guaranteed along all paths, which requires a recurrent reset of all clocks along such a path. In the example, the resulting states of all paths that periodically also take the transitions over `Off`, which leads to a reset of the clock `t_active`, can be constructed by a path shorter than the original. Otherwise, if only transitions over `Execute` are taken consecutively, the clock `t_active` increases, and thus, no model states are repeated.

Using our approach, we can either refer to the example trace and reduce it to a bounded-length sequence, or construct one independent from the actual execution path, considering only the target state. In both cases, the size depends only on the number of clocks involved, instead of the actual, growing execution path length; this boundedness is the major theoretical contribution of our work.

Using the three approaches, we get the construction sequences shown in Fig. 5. Applying the trivial approach (Fig. 5a), we get a sequence of 43 locations, which have to be traversed before we reach the targeted state. This sequence will keep growing over time, without an upper bound in length. Using the approach by Rinast, the first sequence of 10 `Execute` steps can be discarded, as the clock state after the `Off` section is identical to the one at the initial model state. This identified shortcut is used to perform the construction only via the second round of `Execute` steps. Nevertheless, the `Off` location is not necessarily reached frequently, and therefore, paths exist for which this approach does not return a length-bounded construction sequence. Finally, the approach introduced in this work leads to a construction sequence which requires 10 DBM operations, which are transformed into a sequence of 5 locations. In general, we can guarantee that our approach always requires at most $1+2*|T|+|T|*(|T|+1) = 1+2*2+2*3 = 11$ operations for the clock state construction of this model, based on the number of $|T| = 2$ clocks.

5 Overapproximation phase (O-phase)

In the first phase, the **O-phase**, we determine a sequence S_{approx}, which, when applied to DBM_{init}, results in an approximation $\text{DBM}_{\text{approx}} \in \mathcal{DBM}_{\text{approx}}$, where $\mathcal{DBM}_{\text{approx}}$ is the set of all overapproximations of DBM_{target}:

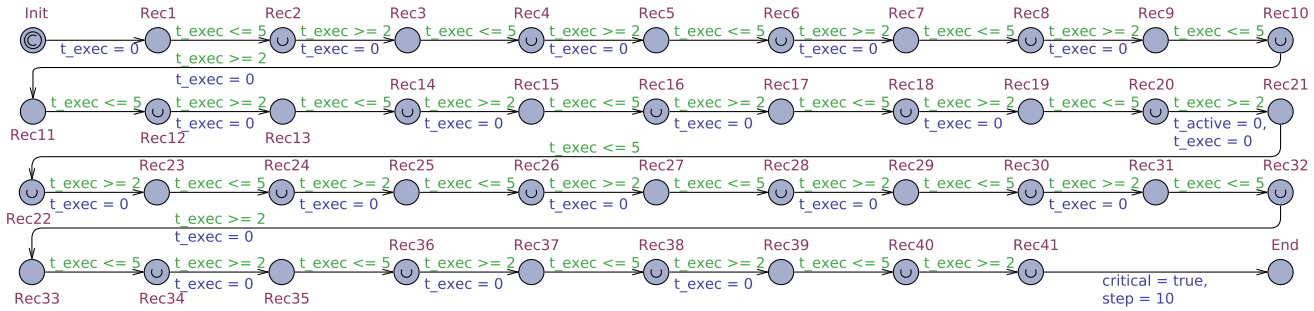
$$DBM_{\text{approx}} \stackrel{\text{def}}{=} S_{\text{approx}}(DBM_{\text{init}}) \supseteq DBM_{\text{target}} \tag{5.1}$$

This overapproximation is required, so that we can constrain DBM_{approx} selectively in a second step to finally obtain DBM_{target}. Generally, a DBM_a overapproximates another DBM_b if the first forms a *superzone* of the other. A DBM forms such a superzone if the following condition holds:

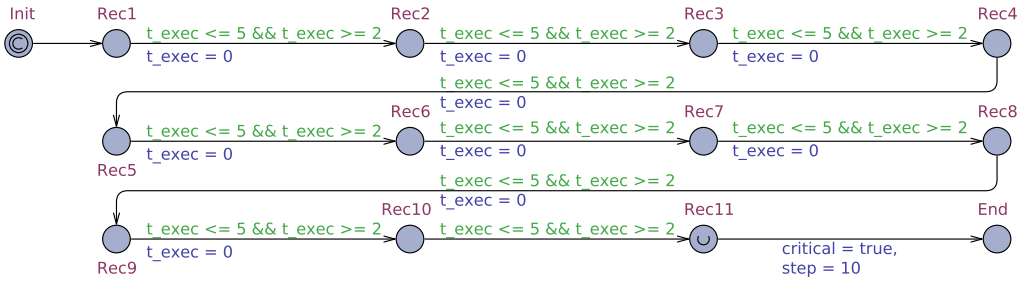
$$\begin{aligned} DBM_a \supseteq DBM_b \\ \Longleftrightarrow \\ \forall 0 \leq i \leq |T| : \forall 0 \leq j \leq |T| : \\ DBM_a[i,j] \geq DBM_b[i,j] \end{aligned} \tag{5.2}$$

Similarly, DBM_a is called a subzone of DBM_b (denoted as $DBM_a \subseteq DBM_b$) iff $DBM_a[i,j] \leq DBM_b[i,j]$ for all DBM entries.

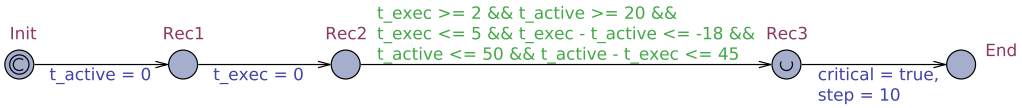
In the simplest case, DBM_{init} is already an overapproximation of DBM_{target} ($DBM_{\text{init}} \in \mathcal{DBM}_{\text{approx}}$), and the O-phase returns an empty transformation sequence $S = ()$. In that case, we can directly continue with the C-phase procedure. Otherwise, if DBM_{init} does not overapproximate DBM_{target}, there exist entries in DBM_{init} which, following from Eq. (5.2), are smaller than the corresponding entries of DBM_{target}. To obtain an overapproximation, those entries need to be overwritten by values which are equal or greater than the ones in DBM_{target}. As mentioned before in Sect. 1, a $SetValue$ operation which would set the DBM entry in the



(a) Trivial approach



(b) Rinast approach



(c) OC approach

Fig. 5 Construction sequences of the three approaches

row of t_i and column of t_j to c (with $c \geq DBM_{\text{target}}[i, j]$) is not defined in OP, nor is it supported by model checkers such as Uppaal. Therefore, we have to determine a suitable sequence of supported non-constraining operations (i.e., $Reset$ and DF) that leads to DBM_{approx}. In the next two sub-sections, we introduce two approaches to find such a sequence S_{approx}: If a reference sequence that leads from DBM_{init} to DBM_{target} is given, we show how to reduce the sequence to obtain S_{approx}. If such a sequence is not known beforehand, we derive an overapproximating sequence from DBM_{target} only.

5.1 O(SEQ): overapproximation based on given sequence *S*

If an original sequence from DBM_{init} to DBM_{target} is already given, e.g., because DBM_{target} was reached via an observable simulation, the sequence construction task is the following: For an operation sequence S_{ref} that transforms DBM_{init} into DBM_{target}, construct a bounded sequence S_{approx} which overapproximates DBM_{target}.

$$DBM_{\text{approx}} \overset{\text{def}}{=} S_{\text{approx}}(DBM_{\text{init}}) \supseteq DBM_{\text{target}}, \quad \text{with } S_{\text{approx}} | S_{\text{ref}} \tag{5.3}$$

In particular, considering the set of all possible overapproximation sequences $\mathcal{S}_{\text{approx}}$, we obtain an $S_{\text{approx}} \in \mathcal{S}_{\text{approx}}$ by repeated elimination of selected operations from S_{ref}. As result, we get a sequence which has, independent of the original sequence size, a bounded length. In fact, we show that it is always possible to reach an overapproximation by at most $|T|$ $Reset$ and $|T| + 1$ DF operations, where T is the set of clocks of the particular model.

We split the argumentation for the individual sequence reductions into two steps, recalling the operations OP described in Sect. 3.3: First, we show that we can eliminate all $Constraint$ and $Close$ operations from S_{ref}, where each elimination results in a shorter sequence that leads from DBM_{init} to an overapproximation of DBM_{target}. Second, we show that we can furthermore eliminate $Reset$ and DF operations that are redundant or already overwritten, which leads to a length-bounded S_{approx}.

5.1.1 Constraint and close elimination

In this step, we consecutively remove $Constraint$ and $Close$ operations from S_{ref}. For the proof that all such operations can be removed while obtaining an overapproximation of DBM_{target}, we have to prove the following three lemmas (cf. Fig. 6):

Lemma 5.1 (Subzone via Constraint and Close) *For a given DBM, applying a C or Cl operation to the DBM leads to a subzone of that DBM:*

$$C(DBM) \subseteq DBM$$
$$Cl(DBM) \subseteq DBM$$

Lemma 5.2 (Chaining of Constraint and Close Eliminations) *Given a DBM and a sequence $S = (\mathcal{C}(DBM) \cup \mathcal{CL}(DBM))^*$, the subsequence S_{el}, (i.e., $S_{el}|S$), obtained by repeated elimination of operations from S results in $S_{el}(DBM) \supseteq S(DBM)$.*

Lemma 5.3 (Preserved superzone relation) *The superzone relation is preserved by the operations $DelayFuture$ and $Reset(t_a, v)$, i.e., for $op \in (\{DF\} \cup \mathcal{R}(DBM))$:*

$$DBM_a \supseteq DBM_b \implies op(DBM_a) \supseteq op(DBM_b)$$

Proof Lemma 5.1 is easily shown based on the definition of a superzone (Eq. 5.2) and the min operation applied by C and Cl (Fig. 2). For both operations, entries $DBM[a, b]$ are only changed by min operations, which take the original entry value c_{ab} as argument (e.g., $min(c_{ab}, v)$ for the $Constraint$ operation). Therefore, each affected entry can only be decreased or remains equal, resulting in a subzone of the original DBM. □

Proof For Lemma 5.2, we need to show that a zone that is obtained via a reduced sequence with multiple C and Cl operations eliminated, is a superzone of the DBM obtained via the original sequence, i.e., show monotonicity of the "greater-than" relation for C and Cl operations. This is the case if a superzone DBM_{S2} obtained for a superzone DBM_{S1} of a DBM is also a superzone of DBM, i.e., for any DBM_a, DBM_b, DBM_c, the *transitivity* property holds:

$$DBM_a \supseteq DBM_b \wedge DBM_b \supseteq DBM_c \implies DBM_a \supseteq DBM_c \tag{5.4}$$

It is straightforward to show that this implication holds, based on the definition of a superzone in Eq. (5.2): From $DBM_a \supseteq DBM_b$ follows for all DBM entries that $DBM_a[i, j] \geq DBM_b[i, j]$. Correspondingly, $DBM_b \supseteq DBM_c$ implies that $DBM_b[i, j] \geq DBM_c[i, j]$ for all DBM entries. As a result, $DBM_a[i, j] \geq DBM_b[i, j] \geq DBM_c[i, j]$, and thus, via the transitivity property of "greater-than," $DBM_a[i, j] \geq DBM_c[i, j]$. In conclusion, following the reverse implication of Eq. (5.2), we get $DBM_a \supseteq DBM_c$. □

Proof For Lemma 5.3, we need to show monotonicity of DF and R, that is, the "greater-than" relation between the entries of two DBMs does not change when applied to both of them. This property ensures that when we obtain a superzone by C and Cl elimination, following operations keep the superzone property intact. Again, that property is easily shown: DF sets all the entries of the first column to ∞, so that for two DBMs (DBM_a, DBM_b), with $DBM_a \supseteq DBM_b$, the corresponding entries either remain unchanged or are both set to ∞ when $DF(DBM_a)$ and $DF(DBM_b)$ are applied. The $Reset(t_a, v)$ operation sets all row values $DBM[a, j]$ to $DBM[0, j] + v$, and all column values $DBM[i, a]$ to $DBM[i, 0] - v$. So again, if $DBM_a \supseteq DBM_b$, those corresponding entries are set to values for which the "greater-than" relation holds ($DBM[0, j]$ or $DBM[i, 0]$), shifted by a constant value v. Therefore, for both DF and $Reset(t_a, v)$, the superzone relation is preserved. □

Following from the three lemmas, we can derive two sequence transformation rules that preserve the subzone relation:

$$apply(S_1 \oplus (C) \oplus S_2, DBM) \subseteq apply(S_1 \oplus S_2, DBM), \tag{5.5}$$
$$\text{with } C \in \mathcal{C}(DBM),\ S_1, S_2 \in \mathcal{OP}(DBM)^*$$
$$apply(S_1 \oplus (Cl) \oplus S_2, DBM) \subseteq apply(S_1 \oplus S_2, DBM), \tag{5.6}$$
$$\text{with } Cl \in \mathcal{CL}(DBM),\ S_1, S_2 \in \mathcal{OP}(DBM)^*$$

We can thus conclude for the resulting DBMs of both sequences:

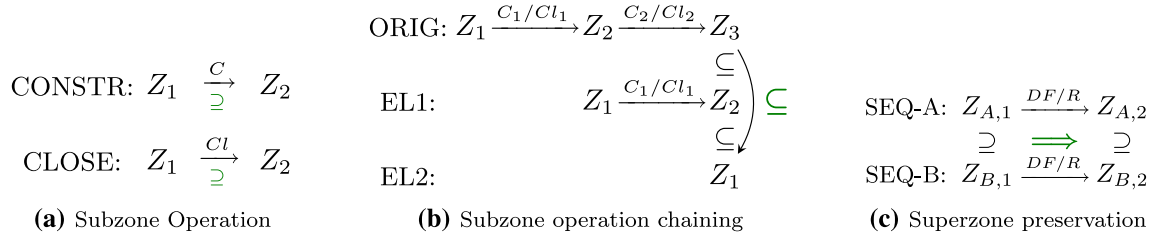
Proposition 5.1 (Overapproximation by elimination of C/CL) *For any sequence $S \in \mathcal{OP}(DBM)^*$ and its subsequence $S_{\text{sub}}|S$ obtained by elimination of all C and Cl operations, $S_{\text{sub}}(DBM) \supseteq S(DBM)$ holds.*

At this point, the reduced operation sequence S_{sub} has the following form:

$$S_{\text{sub}} \in (DF^* \otimes R(t_i, v_i)^1 \otimes DF^* \otimes R(t_j, v_j)^1 \otimes \ldots \otimes R(t_n, v_n)^1 \otimes DF^*)$$

Example 5.1 (Example of C and Cl elimination) Given a zero-initialized DBM_{init}, i.e., all entries are initially 0, and the following reference sequence:

$$S_{\text{ref}} = (DF, C(t_1, t(0), 5), Cl, R(t_1, 0), R(t_2, 0), DF, C(t(0), t_2, -3), Cl, R(t_1, 0), R(t_3, 0)),$$



(a) Subzone Operation **(b)** Subzone operation chaining **(c)** Superzone preservation

Fig. 6 Clock zone relations for constraint elimination

which leads to the target DBM

$$DBM_{\text{target}} = \begin{array}{c} \\ t(0) \\ t_1 \\ t_2 \\ t_3 \end{array} \begin{array}{c} \begin{array}{cccc} t(0) & t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 0 & 0 & -3 & 0 \\ 0 & 0 & -3 & 0 \\ \infty & \infty & 0 & \infty \\ 0 & 0 & -3 & 0 \end{pmatrix} \end{array}$$

After elimination of C and Cl operations, which are $C(t_1, t(0), 5)$, $C(t(0), t_2, -3)$, and their corresponding $Close$ operations, the approximating sequence becomes:

$$S_{\text{approx}} = (DF, R(t_1, 0), R(t_2, 0), DF, R(t_1, 0), R(t_3, 0)),$$

which leads to the overapproximating DBM

$$DBM_{\text{approx}} = S_{\text{approx}}(DBM_{\text{init}}) = \begin{array}{c} \\ t(0) \\ t_1 \\ t_2 \\ t_3 \end{array} \begin{array}{c} \begin{array}{cccc} t(0) & t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \infty & \infty & 0 & \infty \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

5.1.2 Reset and delay-future elimination

Now that we derived a reduced operation sequence by removing all $Constraint$ and $Close$ operations, which results in a superzone of DBM_{target}, we show that we can further reduce the sequence S_{approx} by removing specific $Reset$ and $DelayFuture$ operations while retaining the relation $DBM_{\text{approx}} \supseteq DBM_{\text{target}}$. For that, we use the reduction rules formulated in the following two lemmas:

Lemma 5.4 (DF sequence reduction) *A sequence of DF operations $S = DF^+$ can be reduced to a single DF operation without affecting the outcome of applying S to a DBM:*

$$apply(DF^+, DBM) = apply(DF, DBM) \tag{5.7}$$

Lemma 5.5 (Reset elimination in reset-df sequences) *Given a DBM and an operation sequence $(R(t_a, v_{a,1})) \oplus S_{RD,\text{sub}} \oplus (R(t_a, v_{a,2}))$ with $S_{RD,\text{sub}} \in (\mathcal{R}(DBM)_{\setminus\{a\}} \cup \{DF\})^*$, i.e., a sequence that starts and ends with a reset of the same clock t_a (to arbitrary values), and between both resets, only contains DF operations and resets of clocks t_i, with $i \neq a$, we can omit the first reset of t_a without affecting the outcome of applying S to a DBM:*

$$\begin{aligned} &apply((R(t_a, v_{a,1})) \oplus S_{RD,sub} \oplus (R(t_a, v_{a,2})), DBM) \\ &\quad = apply(S_{RD,sub} \oplus (R(t_a, v_{a,2})), DBM), \\ &\quad with\ t_a \in T(DBM) \end{aligned} \tag{5.8}$$

Proof Lemma 5.4 is easily proven; as a DF operation only assigns the first-column entries $DBM[i, 0], i \neq 0$ to ∞, subsequent DF operations keep the DBM unchanged. □

Proof We prove Lemma 5.5 based on the flow dependency function in Eq. (3.20). Given a DBM with $n = |T(DBM)|$, we first initialize I_0:

$$\begin{aligned} I_0 &\overset{\text{def}}{=} w(R(t_a, v_{a,1})) \\ &= \{(a, 0), \ldots, (a, n), (0, a), \ldots, (n, a)\} \end{aligned}$$

Then, we either apply a first DF (resulting in $I_{1,1}$) or $R(t_i, v_i), i \neq a$ (resulting in $I_{1,2}$):

$$\begin{aligned} I_{1,1} &= fd(DF, I_0) = (I_0 \setminus w(DF)) \cup \emptyset \\ &= \{(a, 1), \ldots, (a, n), (0, a), \ldots, (n, a)\} \subseteq I_0 \\ I_{1,2} &= fd(R(t_i, v_i), I_0) = (I_0 \setminus w(R(t_i, v_i))) \cup I_{op} \\ &= (I_0 \setminus \{(i, 0), \ldots, (i, n), (0, i), \ldots, (n, i)\}) \\ &\quad \cup \{(i, a), (a, i)\} \\ &= I_0 \end{aligned}$$

As $I_{1,1} \subseteq I_0$ and $I_{1,2} = I_0$ hold, we can make use of Eq. (3.23) and consider I_0 the "worst-case" dependency after any sequence $S_{RD,sub} \in (\mathcal{R}(DBM)_{\setminus\{a\}} \cup \{DF\})^*$. In other words, using I_0 means that in the most dependent case resulting from any sequence $S_{RD,sub}$, still only the entries in the row and column of t_a depend on values written by $R(t_a, v_{a,1})$. Finally, we apply $R(t_a, v_{a,2})$ and get:

$$I_2 = fd(R(t_a, v_{a,2}), I_0) = (I_0 \setminus w(R(t_a, v_{a,2}))) \cup I_{op}$$

$$= (I_0 \backslash \{(a,0),\ldots,(a,n),(0,a),\ldots,(n,a)\}) \cup \emptyset$$
$$= \emptyset$$

From $I_2 = \emptyset$, we can conclude that no DBM entry is affected by the operation $R(t_a, v_{a,1})$ anymore after we applied $R(t_a, v_{a,2})$, and thus, $R(t_a, v_{a,1})$ can be omitted from the sequence. □

We can now apply rule (5.8) repeatedly, followed by rule (5.7) where applicable, which leads to the following lemma:

Lemma 5.6 (Reset-df sequence reduction) *Any sequence* $S \in (\mathcal{R}(DBM) \cup \{DF\})^*$, *can be reduced to a sequence* $S_{red} \in (DF?\otimes(R_1)\otimes DF?\otimes\ldots\otimes(R_m)\otimes DF?)$, $R_i \in \mathcal{R}(DBM)$, *with* $\forall R(t_a, v_a), R(t_b, v_b) \in S_{red} : t_a \neq t_b$ *by eliminating each but the last reset of every clock which is reset in S, and removing all but one DF operation in each subsequence of DF operations.*

Example 5.2 (Example of R and DF elimination) Based on the intermediate overapproximation sequence S_{approx} obtained in Example 5.1, eliminating the overwritten operations, i.e., the operation $R(t_1, 0)$, results in the final overapproximation sequence

$$S_{\text{approx}} = (DF, R(t_2,0), DF, R(t_1,0), R(t_3,0)),$$

which leads to the overapproximation of DBM_{target}:

$$DBM_{\text{approx}} = \begin{array}{c} \\ t(0) \\ t_1 \\ t_2 \\ t_3 \end{array} \begin{array}{c} \begin{array}{cccc} t(0) & t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \infty & \infty & 0 & \infty \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

5.1.3 Operation sequence reduction: summary

In summary, for the O(SEQ) approach based on a given sequence S, we proved that all $Constraint$ and $Close$ operations can be removed from a sequence of operations, if we aim for a DBM_{approx} which overapproximates DBM_{target} (cf. Sect. 5.1.1). Furthermore, we can reduce the resulting sequence by eliminating subsequent occurrences of DF operations and resets of clocks which are reset again later in the sequence (cf. Sect. 5.1.2). Therefore, we have proved the following for the overapproximation of DBM states:

Proposition 5.2 (Overapproximation sequences) *Given a DBM, any sequence* $S \in \mathcal{OP}(DBM)^*$, *i.e., any arbitrary sequence of DF, R, C, and Cl operations, can be reduced to a sequence* $S_{appr} \in (DF? \otimes (R_1) \otimes DF? \otimes \ldots \otimes (R_m)\otimes DF?)$, $R_i \in \mathcal{R}(DBM)$, *with* $\forall R(t_a, v_a), R(t_b, v_b) \in S_{appr} : t_a \neq t_b$, *i.e., a sequence of DF and R operations in which each clock is reset at most once, and DF operations occur at most once between resets, such that* $S_{appr}(DBM) \supseteq S(DBM)$ *holds.*

Overall, we can conclude that for each sequence of operations in OP for any DBM, there exists a reduced sequence with at most $|T(DBM)|$ $Reset$ and $|T(DBM)|+1$ $DelayFuture$ operations.

5.2 O(DBM): overapproximation sequence based on DBM_{target}

In this section, we approach the second scenario **S2** from Sect. 1, i.e., the case that S_{ref} is not given and an overapproximation sequence S_{approx} (for which the relation $S_{\text{approx}}(DBM_{\text{init}}) \supseteq DBM_{\text{target}}$ holds) must be derived from characteristics of DBM_{init}, DBM_{target}, and the operation set OP alone.

The following two lemmas provide the base for our approach:

Lemma 5.7 (Approximation sequence existence) *Given a zero-initialized* DBM_{init}, *for each* $DBM_{\text{target}} = S(DBM_{\text{init}})$ *reached by a sequence* $S = \mathcal{OP}(DBM_{\text{init}})^*$, *there exists a sequence* $S_{\text{approx}} = (DF, R_1, DF, R_2, \ldots, R_n, DF)$, $R_i \in \mathcal{R}(DBM_{\text{init}})$, *with* $S_{\text{approx}}(DBM_{\text{init}}) \supseteq DBM_{\text{target}}$.

Lemma 5.8 (reset-(df)-all to reset-df-all) *If a reset-(df)-all sequence overapproximates a DBM, the reset-df-all sequences obtained by enforcing a DF after each reset overapproximates the DBM as well.*

Proof Lemma 5.7 holds as zero-initialization implies a prefix sequence $S_R = (R(t_1,0),\ldots,R(t_n,0))$, with $n = |T(DBM)|$, which turns each sequence S into $S_{ext} = S_R \oplus S$. It follows that each clock is reset at least once, and due to Proposition 5.2, we can reduce S_{ext} to S_{approx}, with $S_{\text{approx}}(DBM_{\text{init}}) \supseteq DBM_{\text{target}}$. □

Proof Lemma 5.8 holds as DF operations only set particular entries to ∞ (cf. Eq. (3.3)) and thus always lead to superzones of the original DBM. □

The challenge is to determine a reset-(df)-all sequence without access to an S_{ref} to reduce. The potentially suitable reset-(df)-all sequences differ in three characteristics: The order of resets (C1), the reset values (C2), and the occurrence of DF operations between the resets (C3); the set of such sequences is infinite. Due to Lemma 5.8, we can remove C3 by limiting the search space to reset-df-all sequences. The search space becomes finite for classes of automata that restrict C2, including the formalism of TAs as defined in [5], which supports resets to 0 (i.e., 0-resets) only; the *Uppaal* tool allows arbitrary positive reset values (i.e., $\mathbb{N}_0$-resets)

during simulation, but limits support to 0-resets during verification. In the remainder of this section, we approach DBM overapproximation for both the case of 0-resets (Sect. 5.2.1) and $\mathbb{N}_0$-resets (Sect. 5.2.2).

5.2.1 O(DBM) approach for 0-resets

The approach for 0-resets is straightforward, as we can directly infer reset orders from the two-clock constraints $DBM[i,j], i \neq j \neq 0$ (cf. Fact 4). From Fact 4, we know that $DBM[i,j]$ is strictly positive iff t_i is reset before t_j. Thus, a possible algorithmic approach to determine a valid reset order is to sum the amount of positive values for each column j, and order the clocks t_j accordingly in ascending order of the sums. This approach builds on the fact that the higher the amount of positive values in a column j, the more clocks were reset before t_j, so that the clock with the lowest column sum was reset first, while the clock with the highest sum was reset last. If clocks are reset simultaneously (i.e., without intermediate delay), the same number of clocks must be reset before them, and thus, their sums of positive column values are equal; the order of these clocks can be chosen arbitrarily.

The order is formally expressed as follows: Given a DBM with $n = |T(DBM)|$, the set $PSUM = \{(c_1, t_1), \ldots, (c_n, t_n)\}, t_j \in T(DBM), c_j = |\{i \mid i \in \mathbb{N} \cap [1,n], DBM[i,j] > 0\}|$ and any valid order $\leq_C$: $PSUM \times PSUM$, with $(c_i, t_i) \leq_C (c_j, t_j) \iff c_i \leq c_j$, we can derive a corresponding reset order $Ord_R : T(DBM) \times T(DBM)$, with

$$Ord_R(t_i, t_j) \iff \exists (c_i, t_i), (c_j, t_j) \in PSUM : (c_i, t_i) \leq_C (c_j, t_j), \tag{5.9}$$

The resulting orderings Ord_R only differ in the order of their "simultaneously" reset clocks. Each reset-df-all sequence that follows such order is guaranteed to lead to an overapproximation of DBM_{target}.

5.2.2 O(DBM) approach for $\mathbb{N}_0$-resets

While the reset order is the only variable for the case of 0-resets, neither the reset order nor the reset values are known for the case of $\mathbb{N}_0$-resets. In the following, we provide a method to derive overapproximating reset-df-all sequences with $\mathbb{N}_0$-resets, and split the argumentation into two parts:

P1 We show how reset values are obtained if the reset order is given.

P2 We show how reset orders are obtained for which valid reset valuations exist.

Determine reset values (P1) The process for the determination of reset values is shown in Fig. 7. Given a reset-df-all sequence S with unknown reset values (Fig. 7-(1)), we derive a constraint system on reset values from the relation between the entries of the overapproximating and target DBM (Fig. 7-(2)). Following from Fact 5, the constraints are either $v_i - v_j \geq DBM_{\text{target}}[i,j]$ or $\infty \geq DBM_{\text{target}}[i,j]$ (Fig. 7-(3a–c)). As the constraints $DBM_{\text{target}}[i,j] \leq \infty$ trivially hold, they can be omitted from the constraint system. Furthermore, $\mathbb{N}_0$-resets impose constraints $v_i \geq 0$ for each $i \in \mathbb{N} \cap [1, |T(DBM)|]$ (Fig. 7-(4)). The adapted constraint system, which solely consists of constraints on some particular (pairs of) reset values now, can be extended to a constraint system over all reset value pairs by adding the trivially satisfied $v_i - v_j \leq \infty$ for the remaining value pairs, and introducing an artificial reset value $v(0) = 0$ (Fig. 7-(5)) similar to $t(0)$ for clock differences. We can represent this new constraint system as DBM, which we denote as DBM_v (Fig. 7-(6)), as it expresses constraints on reset values v, unlike the usual DBM that expresses constraints on clock values. For the system DBM_v, the following lemma holds:

Lemma 5.9 (Reset values from DBM_v) *Given a reset-df-all sequence S with unknown reset values, each valid reset valuation of DBM_v provides reset values such that S leads to an overapproximation of DBM_{target}.*

Proof Lemma 5.9 holds as DBM_v is composed of all reset value constraints that need to be satisfied to overapproximate DBM_{target}. □

Finally, the DBM_v can be transformed to a graph G_v (Fig. 7-(7), cf. Sect. 3.4).

Valid reset valuations only exist if DBM_v is non-empty, and thus, if G_v has only non-negative weight cycles (cf. Fact 1). Only a subset of cycles needs to be checked; in fact, we can reduce the check as follows:

Lemma 5.10 (Non-emptyness of DBM_v by p_H) *Given a DBM_{target} and a reset-df-all sequence S leading to DBM_{approx}, the DBM_v derived from $DBM_{\text{approx}} \supseteq DBM_{\text{target}}$ is non-empty iff the unique Hamiltonian cycle p_H over non-∞ edges, i.e., the cycle over all vertices in inverse reset order, is an all-positive-prefix path in its graph G_v.*

Proof Fact 8 transfers to G_v if we consider the Hamiltonian path p_H over all vertices $v_i \in V(G_v)$ in *inverse* reset order instead (evident from Fig. 7-(7)), and due to Fact 3, we can then ignore all paths that are neither p_H nor its sub-paths via cycle chords. The remaining cycles traverse only edges with $-DBM_{\text{target}}[i,j]$ and 0 weights; recall that the 0 weights were artificially introduced by $v_i \geq 0$ for edges $e = (v_i, v_0)$. Given a closed DBM_{target} with *minimum edge costs* (cf. Fact 2), we can infer that a *maximum edge costs* property holds

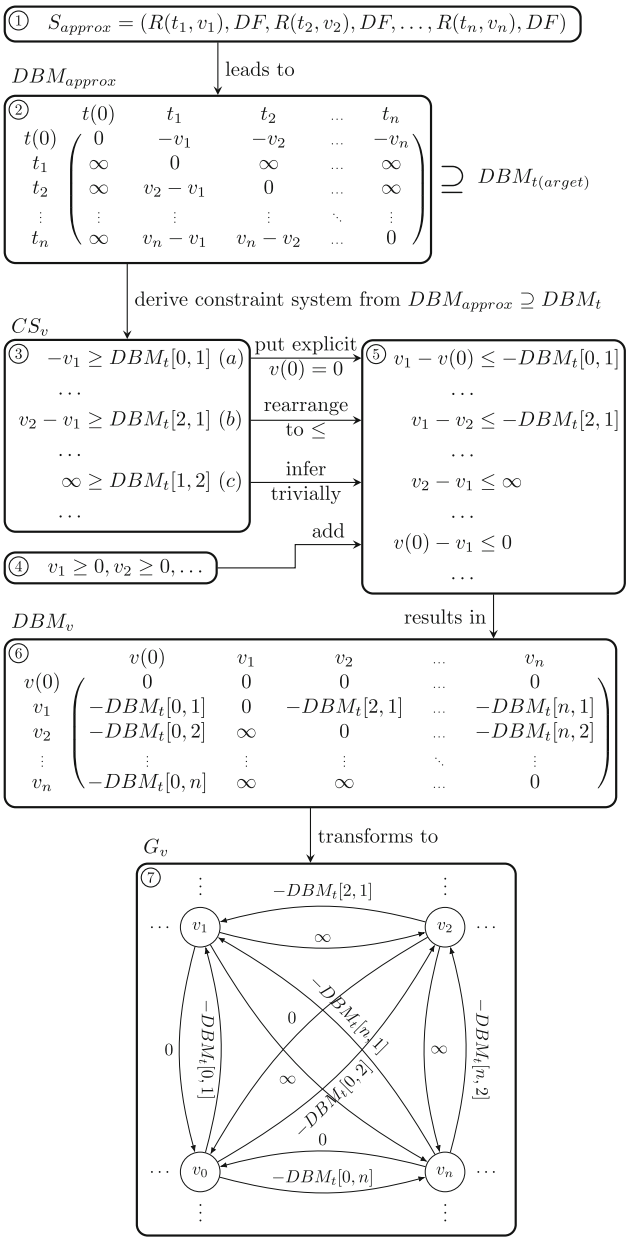


Fig. 7 Derive DBM_v from relation $DBM_{\text{approx}} \supseteq DBM_{\text{target}}$

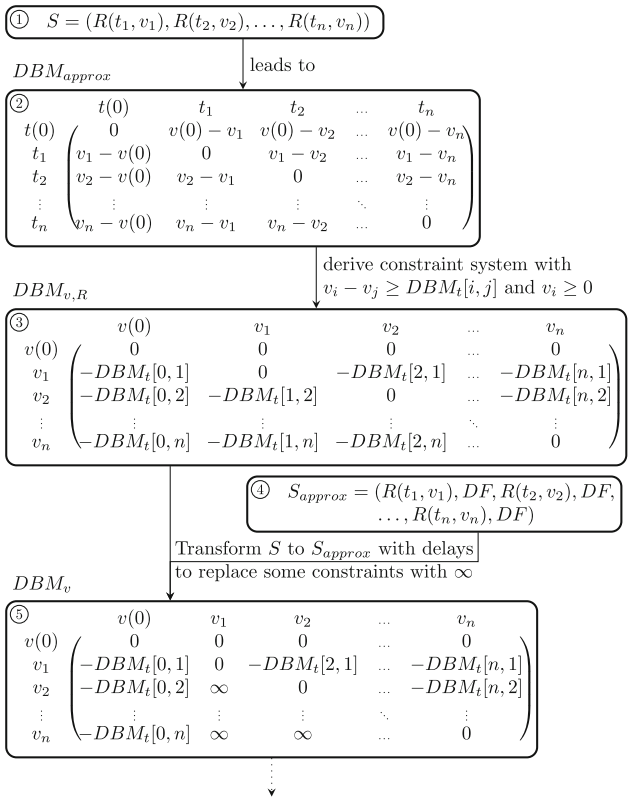


Fig. 8 Derive $DBM_{v,R}$ from relation $DBM_{\text{approx}} \supseteq DBM_{\text{target}}$ for a reduction to DBM_v

for all paths that only traverse edges with $-DBM_{\text{target}}[i, j]$ weights, and in consequence, we can ignore shorter paths derived via cycle chords from paths with known positive costs. Based on the maximum edge costs property, we can ignore all sub-cycles of p_H (via cycle chords) that do not traverse the artificial 0 weights, i.e., we only have to check p_H and its *prefix cycles*. As the final edge of such prefix cycles has 0 weight and thus does not affect the accumulated path cost, it suffices to check the *prefix paths* of p_H instead, and thus, the final set of paths to check are p_H and all its prefix paths, i.e., we check if p_H is an *all-positive-prefix path*. Thus, Lemma 5.10 holds. □

Determine reset order (P2) In the general case, however, the reset-df-all sequence is unknown, so we first need to find a suitable reset order that leads to a non-empty DBM_v before we can derive suitable reset values from that DBM_v as described in **P1**. We denote the sets of all potential DBM_v and their graphs G_v as $\mathcal{DBM}_v$ and $\mathcal{G}_v$, respectively. Recall that $DBM_{\text{approx}} \supseteq DBM_{\text{target}}$ yields constraints $v_i - v_j \geq DBM_{\text{target}}[i, j]$ only for a subset of value pairs (v_i, v_j) depending on the reset order, and if we chose another reset order, we would obtain constraints on a different subset of value pairs. The concept to determine a suitable reset order is shown in Fig. 8. We first assume the reset-all sequence (Fig. 8-(1)), whose resulting clock zone of DBM_{approx} (Fig. 8-(2)) represents a single point, and which thus results in the most restrictive $DBM_{v,R}$ (Fig. 8-(3)) with constraints $DBM_{\text{target}}[i, j] \leq v_i - v_j$ on **all** pairs (v_i, v_j). For this most restrictive constraint set (whose solution set is usually empty unless DBM_{target} was constructed by exactly such reset-all sequence), we determine which constraints have to be removed to transform $DBM_{v,R}$ into a valid and non-empty DBM_v (Fig. 8-(5)), which corresponds to turning the reset-all sequence into a reset-df-all sequence with a suitable reset order (Fig. 8-(4)). The task is now to derive a reset order from the data in $DBM_{v,R}$, which we approach on the level of the corresponding graphs $G_{v,R}$ and G_v. In terms

of Hamiltonian paths (cf. Lemma 5.10), the relation between $G_{v,R}$ and G_v is expressed as follows:

Lemma 5.11 (Relation between $G_{v,R}$ and G_v) *If p_H is an all-positive-prefix Hamiltonian path over edges with non-∞ weights in $G_{v,R}$, then there exists a graph $G_v \in \mathcal{G}_v$, such that p_H is an all-positive-prefix Hamiltonian path over edges with non-∞ weights in G_v.*

Proof Lemma 5.11 holds as we can obtain such a G_v by setting all edges $e = (v_i, v_j), e \notin p_H, v_j \neq v_0$ in $G_{v,R}$ to ∞, which is equivalent to removing all constraints not implied by the reset order that corresponds to the order of vertices in p_H. □

We can use such a path p_H in $G_{v,R}$ to derive a valid graph G_v and thus a suitable reset order:

Proposition 5.3 (Reset order from p_H) *Given a DBM_{target} reached by an (unknown) sequence $S \in \mathcal{OP}(DBM_{\text{init}})^*$ from a zero-initialized DBM_{init}, if a cycle $p_H = ((v_0, v_a), (v_a, v_b), \ldots, (v_k, v_l), (v_l, v_0))$ is an all-positive-prefix Hamiltonian cycle in the graph $G_{v,R}$, the corresponding reset order $Ord = \{(t_l, t_k), \ldots, (t_b, t_a)\}$ leads to a zone DBM_v, and thus, to reset-df-all sequences $S_{\text{approx}} = (DF, R(t_l, x_l), DF, R(t_k, x_k), DF, \ldots, R(t_a, x_a), DF)$ that result in valid overapproximations of DBM_{target} for some reset values x_i.*

Proof Proposition 5.3 directly follows from Lemma 5.11. □

A concrete reset valuation for the reset-df-all sequence can then be picked arbitrarily from the zone DBM_v as described in **P1** (cf. Lemma 5.9).

Algorithm Based on Lemma 5.9, Proposition 5.3, and the steps shown in Fig. 8, we propose the algorithm shown in Fig. 9 for the overapproximation task. In ll.1 − 4, we derive the graph $G_{v,R}$ via $DBM_{v,R}$, which we calculate as $DBM_{v,R} = -DBM_{\text{target}}^T$, and apply the artificial constraints $v_i \geq 0$ for non-negative reset values. As optimization, we already derive a *partial* clock reset order Ord_P at this point, based on the fact that constraints $v_i - v_j \leq -\infty$ (which result from ∞ values in DBM_{target}) are never satisfiable and thus certainly need to be removed (i.e., replaced by $v_i - v_j \leq \infty$) by choosing the reset order in the reset-df-all sequence accordingly (ll.5 − 6). Based on that ordering, we apply the sub-algorithm $AllPosPrefixPath$ (Fig. 10), which determines an all-positive-prefix path in $G_{v,R}$ and its corresponding *total* clock reset order Ord_T (l.8, cf. Proposition 5.3). Afterward, we obtain the graph G_v for our particular reset order by setting all weights of edges in $G_{v,R}$ that correspond to clock pairs in reset order to ∞ (ll.9 − 12), and transform G_v back to a (non-empty) DBM_v (l.13), for which we restore the closed form (l.14). Finally, we derive an arbitrary valuation of reset values from DBM_v (ll.15-16, cf. Lemma 5.9), and based on both the reset order and reset values, derive and return the overapproximation sequence S_{approx} (ll.17 − 18).

Algorithm: `ApproxSeqFromTargetDBM`

Input : *Closed* and *non-empty* DBM_{target}
Output: Overapproximating operation sequence $S_{approx} = (R(t_a, v_a), DF, \ldots, R(t_n, v_n), DF)$

1 $DBM_{v,R} \leftarrow DBM_{target}^T$ // Transposed with negated values
2 $G_{v,R} \leftarrow DBM_{v,R}$ // Derive graph from $DBM_{v,R}$
3 **foreach** $e = (v_i, v_0) \in E(G_{v,R})$ **do** // Add constraints $v_i \geq 0$
4 $\quad w(e) = 0$
5 **foreach** $e = (v_i, v_j) \in E(G_{v,R}), w(e) = -\infty$ **do** // Opt: Remove $-\infty$ edges
6 $\quad Ord_P \leftarrow Ord_P \cup \{(t_i, t_j)\}$ // Reset t_i before t_j
7 $Ord_P \leftarrow Ord_P^+$ // Get transitive closure of Ordering
8 $Ord_T \leftarrow AllPosPrefixPath(G_{v,R}, v_0, Ord_P, 0, \{\})$
9 $G_v \leftarrow G_{v,R}$ // Copy $G_{v,R}$ as base for G_v
10 **foreach** $e = (v_i, v_j) \in E(G_v)$ **do** // Get G_v from reset order
11 $\quad$ **if** $(t_i, t_j) \in Ord_T$ **then**
12 $\quad\quad w(e) = \infty$
13 $DBM_v \leftarrow G_v$ // Derive DBM_v from G_v
14 $DBM_v \leftarrow Close(DBM_v)$ // Restore closed form
15 **foreach** $t_i \in T$ **do** // Derive reset values
16 $\quad Vals \leftarrow v_i \in [-DBM_v(0,i), DBM_v(i,0)]$
17 $S_{approx} \leftarrow GenSeq(Vals, Ord_T) = (R(t_a, v_a), DF, \ldots, R(t_n, v_n), DF)$
18 **return** S_{approx}

Fig. 9 Derive an approximation sequence via DBM

Algorithm: `AllPosPrefixPath`

Input : Graph G, source vertex $v_a \in V(G)$, partial order Ord_P, weight sum s, visited vertices $V_v \subseteq V(G)$
Output: Path p

1 **if** $V_v = V(G)$ **then**
2 $\quad Ord_T \leftarrow Ord_P^+$ // trans. closure now yields total order
3 $\quad$ **return** Ord_T
4 **foreach** $e = (v_a, v_i) \in E(G), v_i \notin V_v$ **do** // Check v for pos. sum
5 $\quad V_{v,n} \leftarrow V_v \cup v_i$
6 $\quad$ **if** $(t_i, t_a) \in Ord_P \wedge (\forall t_b \notin \{t_a, t_i\} : \{(t_i, t_b), (t_b, t_a)\} \not\subseteq Ord_P)$ **then**
7 $\quad\quad Ord_P \leftarrow AllPosPrefixPath(G_v, v_i, Ord_P, s + w(e), V_{v,n})$
8 $\quad\quad$ **return** Ord_P
9 $\quad$ **if** $s + w(e) >= 0$ **then**
10 $\quad\quad Ord_{P,n} \leftarrow Ord_P \cup \{(t_i, t_a)\}$
11 $\quad\quad Ord_{P,n} \leftarrow AllPosPrefixPath(G_v, v_i, Ord_{P,n}, s + w(e), V_{v,n})$
12 $\quad\quad$ **if** $Ord_{P,n} \neq None$ **then return** $Ord_{P,n}$
13 **return** $None$

Fig. 10 Derive reset order from all-positive-prefix path

Example 5.3 (Overapproximation via O(DBM)) Given DBM_{target} introduced in Example 5.1. We determine $DBM_{v,R} = -DBM_{\text{target}}^T$ and its corresponding DBM graph $G_{v,R}$, and search for an all-positive-prefix path in $G_{v,R}$, which is $p_H = (v_0, v_3), (v_3, v_1), (v_1, v_2), (v_2, v_0))$, from which we can derive the overapproximation sequence

$$S_{approx,O(DBM)} = (DF, R(t_2, 0), DF, R(t_1, 0), DF, R(t_3, 0), DF),$$

leading to the overapproximation of DBM_{target}:

$$DBM_{approx,O(DBM)} = \begin{array}{c} \\ t(0) \\ t_1 \\ t_2 \\ t_3 \end{array} \begin{array}{c} \begin{array}{cccc} t(0) & t_1 & t_2 & t_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ \infty & 0 & 0 & \infty \\ \infty & \infty & 0 & \infty \\ \infty & 0 & 0 & 0 \end{pmatrix} \end{array}$$

5.3 Time and size complexity

To conclude the discussion of the O-phase, we analyze the resulting length and generation time of S_{approx}. Given a DBM with clocks T, for the sequence length, we have shown that any given sequence S_{ref} of operations $op \in \mathcal{OP}(DBM)$ can be reduced to a reset-df sequence S_{approx} (Proposition 5.2), where each clock is reset at most once (i.e., $|T|$ resets at most), and as multiple consecutive DF operations can be reduced to one single DF, S_{approx} has $|T|+1$ DF operations at most. Therefore, the number of required operations during the O-phase, considering a minimum bound of 0 operations for the case $DBM_{\text{init}} == DBM_{\text{approx}}$, is:

$$0 \leq |S_{\text{approx}}| \leq 1 + 2 * |T| \tag{5.10}$$

Regarding the generation time, we notice that the reduction in a given S_{ref} depends on its length and the number of clocks. Algorithmically, we start with the last element of S_{ref}, and check for each element, if it is a $Reset$ not encountered yet, or a DF; if neither is true, we remove the element. As each element is compared against the set of already encountered resets, the upper bound of steps required is therefore $|S| * (|T|-1)$. For the second approach, the derivation of S_{approx} from DBM_{target}, the transformation steps to DBM_v and then to G_v, have a complexity of $O(|T|^2)$. Then, for the determination of an all positive prefix path, in the worst case, no $-\infty$ weights are given, and thus, no a priori information of a partial ordering exists. In that case, a possibly full search through the graph is required by $AllPosPrefixPath$ (Fig. 10). There again, in the worst case, only the last searched path yields positive weight sums on all its prefixes, and all other paths turn negative on the last edge, which leads to a complexity of $O((|T|-1)!)$. In the general case though, a path can turn negative earlier, and is discarded then (together with all paths of which that path is a prefix); also, an all positive prefix path can potentially be discovered earlier. Furthermore, the graph size depends only on the number of clocks, which is usually small (between 3 and 10 clocks per model in our experiment model suite). Among all remaining steps, the $Close$ operation on DBM_v has the highest complexity $O(|T|^3)$. In conclusion, the complexity of the $O(SEQ)$ approach grows linearly with the sequence length, and is therefore suitable for shorter sequences, e.g., for the processing of incoming sequence sections in an on-the-fly application, while the $O(DBM)$ approach guarantees upper construction time bounds for DBMs resulting from sequences of arbitrary length.

6 Constraint phase (C-phase)

At this point, we have derived an overapproximating sequence S_{approx}, which, when applied to DBM_{init}, results in DBM_{approx}, overapproximating DBM_{target}. In the second phase, the C-phase, we now determine a sequence S_{constr}, which transforms DBM_{approx} to the final DBM_{target}:

$$S_{\text{constr}}(DBM_{\text{approx}}) = DBM_{\text{target}} \tag{6.1}$$

All entries of DBM_{approx} fulfill the superzone property of Eq. (5.2) compared to DBM_{target} at this point. Therefore, the main task is now to decrease each entry of DBM_{approx} so that it becomes equal to those of DBM_{target}. While OP does not contain an operation to increase individual DBM entries for the O-phase, it provides such an operation for decreasing individual entries: $Constraint(t_a, t_b, v)$. Therefore, the goal of the C-phase is to obtain a sequence of constraint operations:

$$\begin{aligned} S_{\text{constr}} = {} & C(t_{a1}, t_{b1}, DBM_{\text{target}}\,[a1, b1]) \circ \\ & C(t_{a2}, t_{b2}, DBM_{\text{target}}\,[a2, b2]) \circ \ldots \end{aligned} \tag{6.2}$$

We describe three approaches to derive such a constraining sequence S_{constr} with different trade-offs between generation time and sequence length: A trivial approach using the full constraint system (FCS) in Sect. 6.1, a reduced approach using a minimal constraint system (MCS) (introduced by Larsen et al. [23]) in Sect. 6.2 as well as a minimal approach using a newly introduced relative constraint system (RCS) in Sect. 6.3. Among these approaches, the former already solves the problem of constraining DBM_{approx} to DBM_{target} in general, and the latter two optimize the resulting sequence lengths.

6.1 C(FCS): full constraint system approach

The first approach uses the FCS, i.e., a system of all constraints contained in a DBM, so we perform an individual constraint operation $C(t_i, t_j, DBM_{\text{target}}[i, j])$ on DBM_{approx} for each entry of DBM_{target}. This is the simplest form, and in fact, for its construction we only turn every entry of DBM_{target} into a $Constraint$ operation:

$$\begin{aligned} S_{\text{constr}} = {} & C(t(0), t(0), DBM_{\text{target}}\,[0, 0]) \circ \ldots \circ \\ & C(t(0), t_n, DBM_{\text{target}}\,[0, n]) \circ \ldots \circ \\ & C(t_n, t_n, DBM_{\text{target}}\,[n, n]) \end{aligned} \tag{6.3}$$

While the simplicity of generation is a clear advantage of this approach, the downsides are manifold, even if we remove the constraints on diagonal DBM entries, representing the differences $t_i - t_i$ for clocks $t_i \in T(DBM)$, which are constantly 0 and thus do not need an explicit constraint operation. Certain constraints are already implied by others, and thus, setting a constraint may—via a $Close$ operation—already set multiple DBM entries to their targeted values; applying their $Constraint$ operations from S_{constr} is then redundant. Furthermore, several entries may already equal those of DBM_{target} after the O-phase, so that a constraint for these entries is not required either. Still, this approach allows us to determine a worst case upper bound of required $Constraint$ operations during the C-phase as $(|T|+1)^2$, i.e., every entry of DBM_{target}, which has $T+1$ clocks (the model clocks plus the reference clock $t(0)$).

Example 6.1 (Full constraint system) Given the reference sequence S_{ref} and target DBM DBM_{target} introduced in Example 5.1. Applying the C(FCS) approach to DBM_{target} results in the constraining sequence

$$
\begin{aligned}
&S_{constr,C(FCS)} = \\
&\quad (C(t(0), t_1, 0), C(t(0), t_2, -3), C(t(0), t_3, 0),\\
&\quad C(t_1, t(0), 0), C(t_1, t_2, -3), C(t_1, t_3, 0),\\
&\quad C(t_3, t(0), 0), C(t_3, t_1, 0), C(t_3, t_2, -3))
\end{aligned}
$$

6.2 C(MCS): minimal constraint system approach

The second sequence generation strategy uses the data of an MCS, i.e., a minimal system of constraints that implies all remaining constraints. The idea is to perform a constraint operation on DBM_{approx} for each entry of the MCS of DBM_{target}. An algorithm for the reduction in an FCS to an MCS was introduced by Larsen et al. [23].

The algorithm performs the following steps: For a graph G representing a given DBM, it determines vertex equivalence classes E_i regarding *zero-equivalence*, which means that a cycle with a weight of 0 (*zero-cycle*) containing those vertices exists. In each equivalence class E_i, it determines a cycle over all its vertices (in index ordering), where each weight between two vertices is the shortest path weight between them. The edges of those cycles are added to the MCS. Furthermore, between equivalence classes E_i and E_j, it takes the vertex with smallest index from each of the two classes, and includes two edges between them with shortest path weight.

As only a single cycle instead of the complete graph is considered inside each equivalence class, and only two edges are needed between each pair of equivalence classes, it is obvious that the total number of constraints in an MCS can be much smaller compared to the FCS. On average, the MCS contains a number of constraints reduced by 70–86% [23].

The trade-off lies in the time complexity, as the complexity of determining the equivalence classes is $O(n^2)$, getting the edges between equivalence classes is $O(n)$, and finding the cycles within E_i is $O(n)$, leading to an overall complexity of $O(n^2)$ (or $O(n^3)$, if the DBM is not in closed form). Additionally, when applied in our approach, the MCS may still include constraints whose corresponding DBM entries already had the targeted values.

Example 6.2 (Minimal constraint system) Given the reference sequence S_{ref} and target DBM DBM_{target} introduced in Example 5.1. Applying the C(MCS) approach to DBM_{target} results in the constraining sequence

$$
\begin{aligned}
S_{constr,C(MCS)} = (&C(t(0), t_2, -3), C(t(0), t_1, 0),\\
&C(t_1, t_3, 0), C(t_3, t(0), 0), Cl)
\end{aligned}
$$

6.3 C(RCS): relative constraint system approach

To minimize the required amount of constraints, we introduce the RCS, a system which contains only those constraints that are not already correctly set (i.e., entries $DBM_{\text{approx}}[i, j] \neq DBM_{\text{target}}[i, j]$) or implied by other constraints in the DBM. This approach can be based on the concepts of either FCS or MCS. In the following, we will refer to edges of constraints that are already correctly set as *fixed edges*, and all remaining ones as *non-fixed edges*.

Using the FCS, the procedure is straight-forward: While transforming each entry of DBM_{target} into a $Constraint$ operation, we check if the corresponding DBM_{approx} entry has the same value as the DBM_{target} entry. In that case, the $Constraint$ operation is not added to the final sequence of constraints.

Deriving an RCS with the properties of an MCS requires additional steps. The overall idea is to obtain a minimal set of constraints not already included in DBM_{approx}, which, together with the constraints included in DBM_{approx}, implies all remaining constraints of DBM_{target}. Recall that for the construction of an MCS [23], we determine a) two opposing edges between each pair of equivalence classes E_i and E_j and b) a cycle over all vertices within each equivalence class E_i. In steps where vertices are selected, the original MCS construction algorithm uses the index order of vertices to eliminate non-determinism, which could occur as usually multiple minimal sets of constraints exist for a given constraint system. Our RCS approach introduces an additional step which picks fixed edges first, and only then—if still multiple ambiguous choices exist—relies on the vertex order for the remaining edges like the original MCS algorithm. That way, we refine the selection of vertices in favor of constraints that are already correctly set, and thus, reduce the amount of constraints required additionally.

Algorithm: `Construct(M)RCS`

Input : Graph G, fixed edges E_f
Output: Minimal relative constraint system S

```
1  F ← {}      // Selected fixed edges (⇒ not required in RCS)
2  S ← {}      // Remaining non-fixed edges (⇒ the RCS)
3  E ← GetZeroEqClasses(G)          // as in ref. algorithm
4  P ← AllUniquePairs(E)
5  foreach (Eq_i, Eq_j) ∈ P do      // edges between eq. classes
6      if ∃e = (t_a, t_b) ∈ E_f : t_a ∈ Eq_i ∧ t_b ∈ Eq_j then
7          E_{s,min} ← {e = (t_a, t_b) ∈ E_f | t_a = GetMinSrcIndex(E_f)}
8          e ← {e = (t_a, t_b) ∈ E_{s,min} | t_b = GetMinTgtIndex(E_{s,min})}
9          F ← F ∪ {e}
10     else
11         S ← S ∪ {(minE_i, minE_j)}
12     if ∃e = (t_b, t_a) ∈ E_f : t_a ∈ Eq_i ∧ t_b ∈ Eq_j then
13         E_{s,min} ← {e = (t_b, t_a) ∈ E_f | t_b = GetMinSrcIndex(E_f)}
14         e ← {e = (t_b, t_a) ∈ E_{s,min} | t_a = GetMinTgtIndex(E_{s,min})}
15         F ← F ∪ {e}
16     else
17         S ← S ∪ {(minE_j, minE_i)}
18 foreach Eq_i ∈ E do               // cycles within eq. classes
19     C ← CycleWithMostFixedEdges(Eq_i)   // ⇒ trav. salesman
20     S ← S ∪ (C \ E_f)
21     F ← F ∪ (C ∩ E_f)
22 return S
```

Fig. 11 Construct minimal relative constraint system

Figure 11 shows the adapted procedure, where l.11, l.17 and ll.19 − 20 (with $E_f = \emptyset$, i.e., no fixed edges) match the original MCS algorithm, and ll.7 − 9, ll.13 − 15 and ll.19 − 21 (with $E_f \neq \emptyset$) implement our adaptions for the case that fixed edges are involved. For the determination of edge pairs in a), we will first search for edges of already correctly set DBM entries (l.7 − 9), and only revert to edges of index ordered vertices (l.11)—as in the original algorithm—if none can be found. For the equivalence classes E_i and E_j (with $t_a \in E_i$, $t_b \in E_j$, and $minE_i$ the vertex with smallest index among vertices in E_i), we proceed as follows: Among fixed edges (t_a, t_b), we choose the one with the smallest index a (l.7), and if multiple such edges exist, the one with smallest index b (l.8) among them. If no such fixed edge exists, we choose the edge $(minE_i, minE_j)$ (l.11) as defined in the reference algorithm. Similarly, we determine the second, opposing edge (t_b, t_a) (ll.13 − 15), but with swapped indices, i.e., first the smallest b (l.13) and then the smallest a (l.14), and $(minE_j, minE_i)$ (l.17) otherwise.

For (b), in each equivalence class E_i, we want to determine a cycle over all its vertices, which additionally contains the maximal possible number of fixed edges. This problem is NP-complete and can be transformed to the traveling salesman problem by assigning weight 0 to fixed edges and weight 1 to non-fixed edges. Then, the task is to find the lowest-cost Hamiltonian cycle in that graph, i.e., one that contains as few non-fixed edges that impose additional constraints, and as many fixed edges as possible. Fortunately, the problem complexity grows non-polynomially only in the number of clocks in the DBM, which is rather low in the general case (e.g., between 3 and 10 clocks in our experiment suite, cf. Sect. 9). Therefore, it is viable to determine a solution by both brute force and heuristic methods.

Proof The correctness argument for Fig. 11 is as follows: In the original MCS algorithm, the choice of both the concrete edge pairs between equivalence classes and the edges on a Hamiltonian cycle within each equivalence class could be arbitrary, and each solution would be a correct minimal constraint system (cf. [23] for justification). While the MCS algorithm uses the index order of vertices to choose one of these correct solutions deterministically, our approach just adds another objective (i.e., favoring of fixed edges) to that choice in ll.7 − 8, ll.13 − 14, and l.19. That way, the resulting constraint system is still necessarily correct, and as we pick fixed edges first whenever possible, the resulting RCS is also minimal with respect to DBM_{approx}. □

Example 6.3 (Relative constraint system) Given the reference sequence S_{ref} and target DBM DBM_{target} introduced in Example 5.1, and the resulting overapproximation DBM DBM_{approx} of Example 5.2. Applying the C(MCS) approach to DBM_{target} starting from DBM_{approx} results in the constraining sequence

$$S_{constr,RCS} = (C(t(0), t_2, -3), Cl)$$

6.4 Time and size complexity

Again, we analyze the resulting length and generation time of S_{constr} and further consider the application times of sequences based on FCS, MCS, and RCS. Given a DBM with clocks T, in the worst case (FCS), one C operation is required for every entry of DBM_{approx}, which has $|T|+1$ columns and $|T|+1$ rows. Considering 0 operations as the lower bound for the case $DBM_{\text{approx}} == DBM_{\text{target}}$, this leads to the following estimate of required operations:

$$0 \leq |S_{\text{constr}}| \leq (|T|+1)^2 \tag{6.4}$$

The actual maximal number of required operations lies below that bound, as the main diagonal entries are always 0, leading to $|S_{\text{constr}}| \leq |T| * (|T|+1)$. As certain DBM entries may already be implied by other entries (MCS), or are already set correctly in DBM_{approx} (RCS), we can further reduce the bound by $\sim 80\%$ or $\sim 90\%$, respectively (see Sect. 9).

Combining the sequence length bounds of the O-phase and C-phase, we obtain the total size of our sequence S—the major theoretical result of this work:

Theorem 1 *(CSC sequence complexity bound) Any valid and closed DBM can be reached by a clock state construction sequence S of*

$$0 \leq (|S| = |S_{\text{approx}}| + |S_{\text{constr}}|)$$
$$\leq 1 + 2 * |T| + |T| * (|T| + 1)$$

operations.

Regarding generation times, deriving the FCS has a time complexity of $O(|T|^2)$, as we directly transform each DBM entry into a $Constraint$ operation. For the MCS, the complexity is $O(|T|^3)$, which follows from the complexity of deriving the shortest-path reduction G^R (cf. [23]). Finally, the RCS approach has a complexity of $O(|T|^2)$ for the edges between equivalence classes, and $O((|T| - 1)!)$ to obtain a Hamiltonian cycle with most fixed edges within an equivalence class. However, as for the factorial-time step of the O-phase, the actual costs depend only on the number of clocks, which is comparatively small in the common case.

Finally, we consider the complexity of applying the derived constraint sequence for the C-phase. On the one hand, the FCS approach has the greatest sequence length, but does not require a final $Close$ operation, as all constraints are explicitly contained in the sequence and do not need to be inferred. On the other hand, the MCS and RCS approaches result in shorter sequences, but require a closing step with $O(|T|^3)$ complexity which infers the left out constraints. In the end, the most suitable approach in practice depends on the concrete aim, i.e., whether we require a comparatively short sequence, a shorter execution time, or faster generation times; this question cannot be answered universally though.

7 State construction for Uppaal extended timed automata

At this point, we are given the targeted system state, which—as a reminder—contains the information of the variable values, active locations, and clock state that we want to set, and a clock state construction sequence derived in Sect. 5 and Sect. 6. In this section, we explain how this information extends a concrete (network of) Uppaal automata to restore the desired starting state from which the model simulation will then continue. Note that this section focuses on the (more complex) case of state construction in ETAs (see Sect. 3.1 for the definition of ETAs); however, for a simple TA without synchronization, urgency, and variables, one would proceed similarly, but add the construction sequences directly into the given TA, and implement urgency via explicit invariants $t_u \leq 0$ on a helper clock t_u that was reset before. For state construction in ETAs, we transform the clock state construction sequence and variable data into an additional automaton whose execution restores the clock and variable state before the original system is re-entered at the targeted starting locations. An example of the adapted form of the introduction model (Fig. 1) is shown in Fig. 12, and we will explain the changes in the following.

To integrate the derived DBM operation sequence leading to DBM_{target} into an Uppaal automaton system, we need to translate the operations into a sequence of artificially added locations and edges, each labeled with invariants, or guards and resets, respectively. Transferring the symbolic semantics (cf. Definition 2), i.e., the execution of transitions and validity checks of active locations, to corresponding DBM operations, we obtain the following types of operation sequences:

Initialization: $S_{\text{init}} = (R(t_1, 0), \ldots, R(t_n, 0))$
Initially, all clocks are reset to 0.

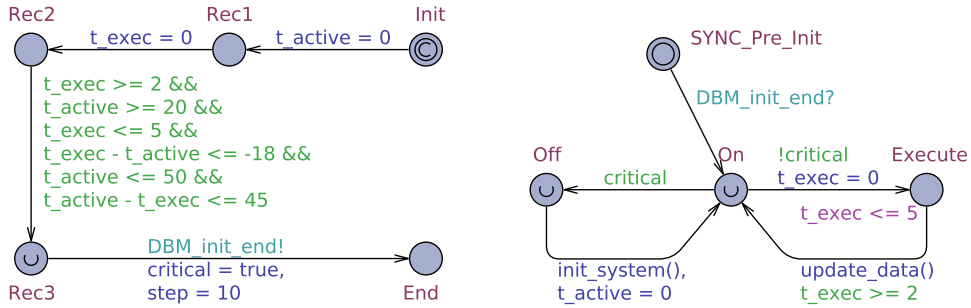
Location: $S_{\text{loc}} \in DF? \otimes \mathcal{C}(DBM)^* \otimes Cl^1$
In a location l, a DF is applied if no urgent or committed next transition is enabled, and all invariants C_l (represented as constraint operations) are applied to the DBM. Afterward, a Cl operation is performed to restore DBM closedness.

Edge: $S_{\text{edge}} \in \mathcal{C}(DBM)^* \otimes Cl^1 \otimes \mathcal{R}(DBM)^* \otimes \mathcal{C}(DBM)^*$ On an edge e_{ab} from l_a to l_b, the atomic guard constraints in $g(e_{ab})$ (again represented as C operations) are applied to the clock state DBM, followed by a Cl operation to turn the DBM into closed form again. Then, all clock resets in $r(e_{ab})$ (represented as R operations) are applied to the DBM, and finally, the invariants $I(l_b)$ of the target location are checked.

Recall that our construction sequence S consists of a subsequence of (alternating) R and DF operations (O-phase), followed by a sequence of C operations (C-phase), which, when based on the MCS or RCS approach, require a final Cl operation to adapt the remaining DBM entries:

$$S \in (\mathcal{R}(DBM)^1 \otimes DF?)^* \otimes \mathcal{C}(DBM)^* \otimes Cl^1 \tag{7.1}$$

The individual elements of this sequence need to be mapped to components of the NTA, i.e., locations and edges, based on the operation sequences applied during simulation identified above. For the subsequence $S_{\text{approx}} \in (\mathcal{R}(DBM)^1 \otimes DF?)^*$, we can use edges with corresponding resets but without guards to new (non-urgent and non-committed) locations without invariants, which results in the sequences $S_{\text{edge}} \in \mathcal{R}(DBM)^*$ and $S_{\text{loc}} = (DF)$. That way, we can express each subsequence $\mathcal{R}(DBM)^* \otimes DF^1$ by an edge–location pair; a sequence of those pairs will cover the complete O-phase. In Fig. 7a, these are the locations `Init`, `Rec1`, and `Rec2`, and their corresponding edges. To enforce all constraints of the C-phase, i.e., the subsequence $S_{\text{constr}} \in (\mathcal{C}(DBM)^* \otimes Cl^1)$, we can use an edge without resets to an unconstrained location, which results in $S_{\text{edge}} \in (\mathcal{C}(DBM)^* \otimes Cl^1)$. Note for all



(a) State construction template **(b)** Adapted model template

Fig. 12 Introduction model adapted for initial state construction

these sequences that in *Uppaal*, a partial $Close(t_a, t_b)$ operation is applied after each $Constraint(t_a, t_b, v)$ instead of a single *Close* over all clocks after a sequence of *Constraint* operations.

The construction of the variable state, which is introduced by the ETA formalism, can be performed via a single transition. We can set the individual variable values, in contrast with the DBM state, directly via variable assignments (cf. Definition 2). In Fig. 7a, we added these assignments to a new edge from `Rec3` to `End` for a separation of concerns; however, a separate edge is not needed for these assignments, as we could add them to the last edge of the clock state construction sequence (i.e., the edge with the constraining guards of the C-phase).

Furthermore, we need to reach the original active locations in the adapted model after the variable and DBM state construction. Normally, we would simply define those locations as *initial* in Uppaal, so that they become active on model initialization. To traverse our artificially added construction sequence though, the first location of that sequence needs to be defined as *initial*, and the final edge of the construction section should lead to the targeted initial location of the original model section. Once the construction sequence is fully traversed (and only then), the original model sections of all automata are re-entered (which is achieved in ETAs by `broadcast` synchronization via `DBM_init_end`).

8 Implementation

We provide a Python implementation of the introduced OC approach, as well as the required interface and interpreter code needed to apply the approaches to Uppaal models. The project implementation is open source [34] [35] under MIT license, and mainly consists of:

- An Uppaal model and Uppaal C code parser (via EBNF grammars building up on the BNF grammar provided with Uppaal),
- An Uppaal model simulator, which allows tracking the simulated operation sequences (= S_{ref}) [34],
- The DBM data structure and operations,
- The state constructor implementations for the trivial approach, Rinast approach (port from Java [20]), and the variants of our OC approach,
- The experiments performed in this paper, including a random sequence generator as alternative input data source.

Compared to the concepts introduced in Sect. 5 and 6, the concrete implementation differs in the following aspects:

- The O(DBM) implementation uses DBM_v directly instead of its graph representation G_v; the result is identical.
- The C(RCS) implementation checks only up to a fixed amount of cycle permutations; while this limitation may not preserve minimality of the resulting sequences, the boundedness is not affected, and it allows a time-efficient application to models with higher amounts of clocks as well.
- Where suitable for sequence-based algorithms (i.e., trivial, Rinast, O(SEQ)), on-the-fly versions are implemented, which process only the newly tracked operations instead of the complete S_{ref} on each call.

After installation, executing `run` in the experiments CLI will run all clock state construction experiments and store the corresponding analysis data. The experiments are available online [36], and the experiment setup and results are explained in Sect. 9.

8.1 Random operation sequence generation

The experiments in Sect. 9 use operation sequences that are either obtained from simulation of TA models, or generated (semi-)randomly based on the subsequences for location states and transitions described in Sect. 7. In general, the sequence generator generates operation sequences via a simulating approach, which keeps track of the current DBM

to which each generated operation is applied; based on that current DBM, the next invariant, guard, or reset values are selected from intervals that keep the DBM non-empty, and thus, keep the sequences feasible. The generation process starts with specifying if non-zero resets should be allowed, and if the sequence should start with an initialization sequence (i.e., a sequence of resets of all clocks to 0). Then, the generator randomly chooses whether a DF is added, followed by a sequence of single-clock invariants on a random subset of clocks with (valid) values (i.e., constraints $C(t_i, t(0), v_i)$, with $t_i \in T_{\text{sub}} \subseteq T(DBM), v_i \in [a, b], a = -DBM[0, i], b = DBM[i, 0]$) that are random within bounds, and a Cl operation; this part reflects the sequence imposed by the initially active locations. Afterward, until the targeted sequence length is reached, we repeatedly add an operation sequence reflecting a transition, followed by another sequence for the newly active location state. Each transition sequence is composed of a sequence of single-clock guards on a random subset of clocks with random (valid) values (i.e., constraints $C(t(0), t_i, -v_i)$, with $t_i \in T_{\text{sub}} \subseteq T(DBM), v_i \in [a, b], a = -DBM[0, i], b = DBM[i, 0]$), followed by a Cl and a sequence of resets of a random subset of clocks to values that are either random (within bounds) if non-zero resets are allowed, or 0 otherwise. The generation of sequences of the newly active location state is identical to the process described for the initial location state. Note that we restrict infinite intervals (i.e., constraint intervals $[a, \infty)$ or the reset value interval $[0, \infty)$) to finite ones during value choice (i.e., $[a, a + c_1]$ and $[0, c_2]$, respectively, for some constants $c_1, c_2 \in \mathbb{N}_0$).

9 Empirical evaluation

Using the described implementation, we evaluate the presented approaches for the O-phase and C-phase. All experiments were executed on an Ubuntu 18.04 LTS system with AMD Ryzen 7 2700X eight-core CPU and 16GB RAM. Overall, we perform three types of experiments, by which we compare:

1. The overall state construction sequence lengths of the `trivial`, `Rinast`, and our `OC` approach, which confirms that in contrast with the former two, our approach generates bounded sequences over time in all tested cases.
2. The constraint sequence lengths of the C-phase approaches $C(FCS)$, $C(MCS)$, and $C(RCS)$, showing that the $C(MCS)$ and $C(RCS)$ approaches reduce the sequence lengths compared to the $C(FCS)$ approach.
3. The construction times of the O-phase and C-phase approaches.

We apply these experiments to two types of data:

Model name	Locations	Edges	Instances	Clocks
2doors	16	22	4	4
bridge	20	21	5	5
fischer	24	30	6	6
fischer-symmetry	40	50	10	10
train-gate	33	41	7	6
train-gate-orig	31	42	6	4
csmacd2	10	22	3	3
tdma	100	217	6	6

Fig. 13 Uppaal model details

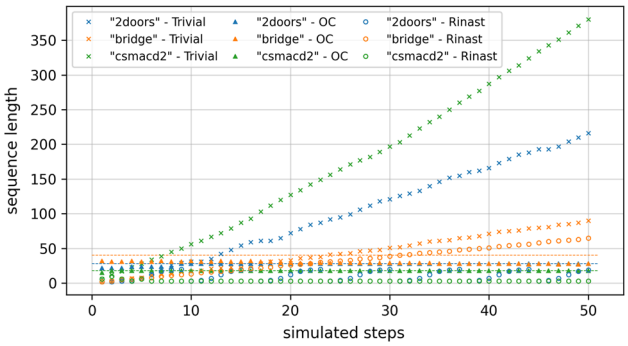


Fig. 14 Sequence lengths after n steps for the trivial (cross), Rinast (circle) and OC (triangle) approach

1. A test suite composed of 6 Uppaal demo models and 2 case study models, from which we derive DBM operation sequences during ongoing execution.
2. 1000 randomly generated operation sequences (per experiment), reflecting the general sequence structure of execution traces in a TA (see Sect. 7 and Sect. 8.1), as a stress test for validation of the model experiment insights.

The main parameters of the experiments are the model size, the number of clocks, and the observed sequence length.

Our experiment model suite consists of 8 models in total, among which the 6 models `2doors`, `bridge`, `fischer`, `fischer-symmetry`, `train-gate`, and `train- gate-orig` are the complete set of cyclic, deadlock-free models of the demo model suite of standard Uppaal (i.e., without extensions such as *Uppaal SMC*), and `csmacd2` [19] and `tdma` [26] were developed in case studies. Figure 13 gives an overview of their characteristics. The suite contains models with different amounts of locations (10–100), edges (21–217), and clocks (3–10). For each model, we execute 1000 simulation runs over 100 transitions, and calculate the minimum, average, and maximum of sequence lengths and construction times over all runs at each individual simulation step.

Applying the approaches to the test suite gives the results shown in Figs. 14 and 15, as well as the data table as shown in Fig. 20 provided in appendix 1, which shows the construction sequence lengths after 1, 10, 50, and 100 executed transitions during model simulation for all Uppaal models. For 3

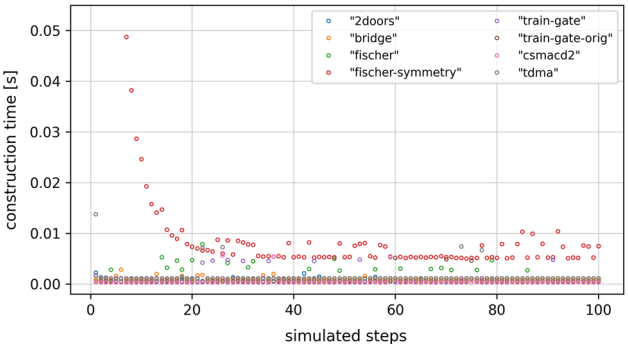


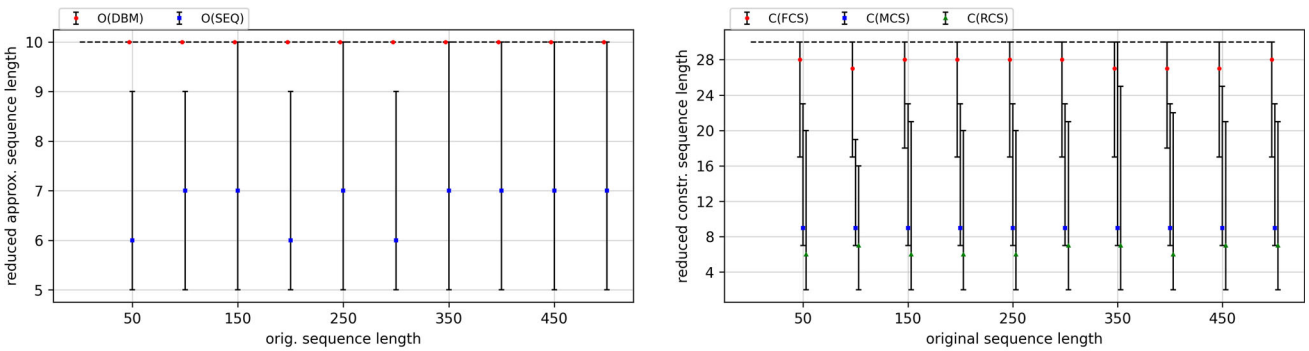
Fig. 15 Average construction times at each step

selected models of the test suite (`2doors`, `bridge`, and `csmacd2`), the graphs in Fig. 14 show the state construction sequence lengths of the trivial, Rinast, and OC approach for a simulation up to 50 steps, together with the calculated bounds of the OC approach. Finally, Fig. 15 shows the total construction sequence generation and application time required for all 8 Uppaal models on each step during simulation.

Applying the approaches to random DBM operation sequences based on a range of different numbers of clocks and sequence lengths gives the results shown in Figs. 16, 17, 18, and 19. For a fixed number of 5 clocks, Figs. 16 and 18 show the ranges of reduced lengths and construction times, respectively, for different O-phase and C-phase approaches applied to different lengths of random input sequences (50–500 operations). Figures 17 and 19 show the lengths of state construction sequences derived by the OC approach variants and construction times, respectively, from fixed-length random sequences (100 operations) based on different numbers of clocks (1–10 clocks).

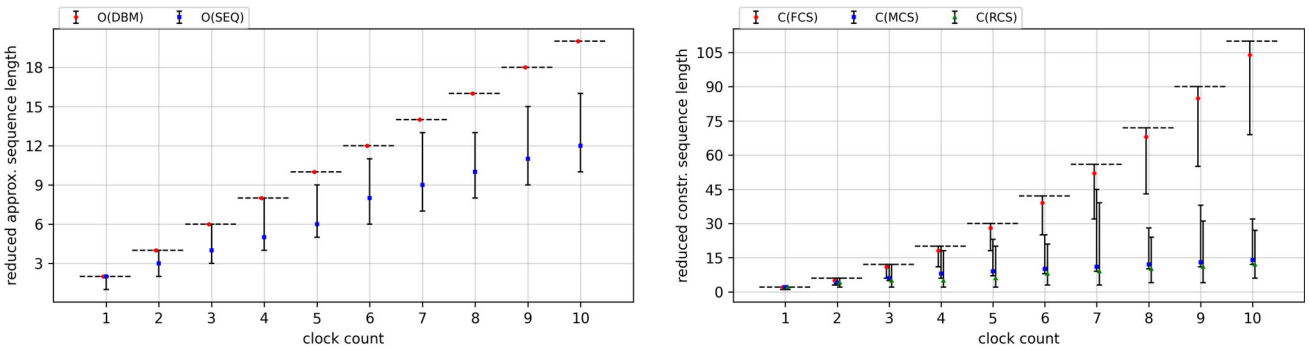
9.1 Evaluation

The model-based experiments show that, as expected for all models, the sequences of the trivial approach are not bounded, and grow linearly over time (cf. Figure 20). The approach by Rinast produces bounded sequences for all models except for `bridge`, which has a global clock that is never reset, and thus, never re-visits any reached state during simulation. Our approach generates bounded sequences



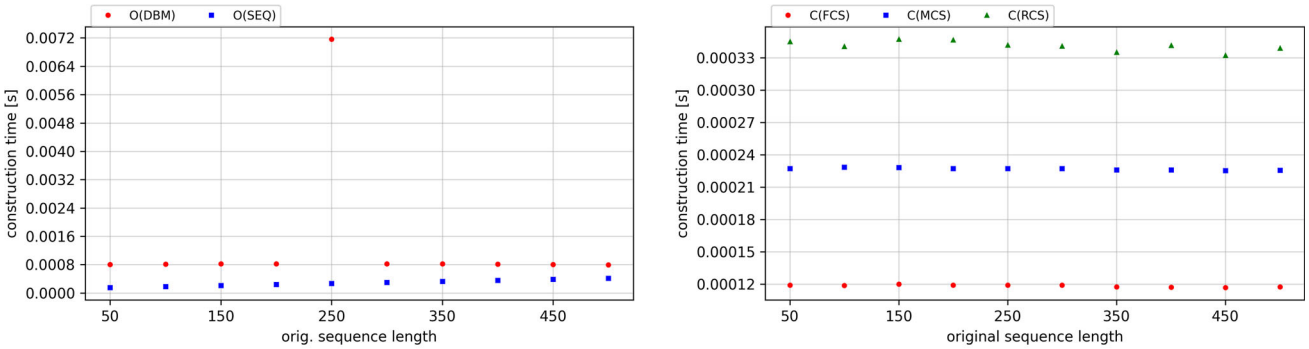
(a) O-phase **(b)** C-phase

Fig. 16 State construction sequence lengths for fixed clock count and variable input sequence length (the intervals represent the minimum, average, and maximum values over 1000 runs)



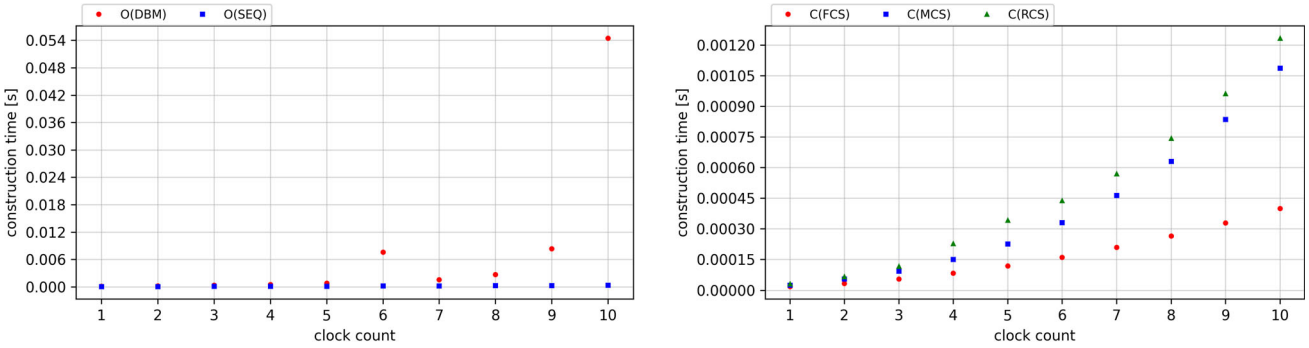
(a) O-phase **(b)** C-phase

Fig. 17 State construction sequence lengths for fixed input sequence length and variable clock count (the intervals represent the minimum, average, and maximum values over 1000 runs)



(a) O-phase **(b)** C-phase

Fig. 18 State construction times for fixed clock count and variable input sequence length



(a) O-phase **(b)** C-phase

Fig. 19 State construction times for fixed input sequence length and variable clock count

in all cases, with lengths that are generally shorter than the other two approaches in the long run (e.g., 26 operations for O(DBM) + C(FCS) for `bridge`, compared to 212 operations of `trivial`).

Figure 14 underlines these results. It shows that at an (early) point during simulation (for the three models between 8 and 28 steps), the sequence lengths of the trivial approach outgrow the lengths of our OC approach. For `csmacd2`, we observe that the Rinast approach performs best, as *Reset* operations overwrite the clock values on almost every edge in the model. Furthermore, we see that the actual sequence lengths of our approach lie well below the theoretically calculated bounds (dashed lines) for the most part. These properties also hold for the remaining models.

As last experiment applied to the test suite, Fig. 15 shows that the construction times lie in a real-time feasible range; during the experiments, below 50 milliseconds were required for most constructions. Only for very few outliers, especially in the first steps of `fischer-symmetry` (outside of the plotted range), the construction time exceeds 1 second, with averages still below 1 second. In the regular case, it is possible to restore a model state 2–20 times within a second.

The experiments on generated DBM operation sequences validate the results of the test suite in a systematic manner, and yield the following results: The construction sequence lengths lie in constant ranges for growing input sequence lengths (Fig. 16). With increasing numbers of clocks (Fig. 17), the construction sequence lengths grow linearly for the O-phase and quadratically for the C-phase, both as expected from Theorem 1. Furthermore, we see that the MCS and RCS sequence lengths always lie below the FCS sequence length, and the C(RCS) approach leads to sequences that are $\sim$ 25% shorter compared to the C(MCS) approach. In terms of construction times for growing input sequences (Fig. 18), the random sequence experiments confirm that for both O-phase and C-phase the required times stay constant, except for O(SEQ); the latter is expected as O(SEQ) gets the complete sequence as a whole in this experiment, in contrast with the step-wise data increments of the model experiments, so that in the worst case, the full input sequence needs to be searched. From Fig. 19, we see that increasing the number of clocks results in constant times for O(DBM) and non-polynomial times for O(SEQ) (cf. Sect. 5.3) in the O-phase, and in the C-phase, the times grow quadratically for C(FCS), cubically for C(MCS), and non-polynomial for C(RCS) (cf.

Sect. 6.4). C(RCS), however, switches to cubic growth similar to C(MCS) once the number of clocks leads to permutation counts that exceed the predefined limit (cf. Sect. 8).

Overall, we can conclude that the new approach allows generating length-bounded sequences throughout the complete model simulation, and does so within real-time capable time frames.

10 Conclusion and future work

In this article, we introduced and implemented multiple approaches to derive bounded-length operation sequences to restore given DBM states in timed automata. We found out that the complexity of the individual approach variants either only depends on the (fixed) amount of clocks in a system (i.e., for O(DBM), C(FCS), C(MCS), and C(RCS)) or allows the formulation of alternative versions (i.e., for O(SEQ)) which process operations on the fly, making them suitable for use in online model checking contexts. The experiments revealed that early during simulation (between 8 and 28 steps for the test model suite), the sequence lengths of the OC approaches become (and remain) shorter than the lengths of the trivial approach and—except for one model—of the Rinast approach as well.

In future versions, the sequence lengths could be further shortened by an extension of our approach that handles short sections in which not all clocks have been reset already, or by a hybrid approach that selects the minimum sequence of the trivial and our OC approach. Furthermore, more efficient algorithms may be used for the graph-based search problems of O(DBM) and C(RCS), so that the approach becomes applicable for systems with high amounts of clocks. The insights on overapproximating and constraining sequences may be used for model checking routines, e.g., for time efficient falsification of safety properties, and deserve further investigation.

Acknowledgements We thank the anonymous reviewers who spotted a serious flaw in our argumentation on the removal of operations and who provided numerous helpful hints that improved the readability and accessibility of the article.

Funding Open Access funding enabled and organized by Projekt DEAL.

Availability of data and material The models `2doors`, `bridge`, `train-gate`, `train-gate- orig`, `fischer`, and `fischer-symmetry` are part of the demo model suite of Uppaal available at https://www.uppaal.org/. The models `csmacd2` [19] and `tdma` [26] were developed in case studies. All other experimental data were programmatically generated by the authors via the experiments linked under *Code availability*.

Code availability The project implementation is open source under MIT license. The base Uppyyl simulator is available at https://github.com/S-Lehmann/uppyyl-simulator, the Uppyyl state constructor is available at https://github.com/S-Lehmann/uppyyl-state-constructor, and the examples and experiments covered in this article are altogether available at https://github.com/S-Lehmann/uppyyl-state-constructor-experiments.

Declarations

Conflicts of interest Not applicable.

Model-based experiments: detailed data table

See Fig. 20.

Fig. 20 Sequence lengths after n steps (minimum, average, and maximum lengths)

		(min/avg/max) seq. length after n steps			
Model	**Approach**	**1**	**10**	**50**	**100**
2doors	Trivial	(2,2,2)	(34,37,40)	(228,233,238)	(473,478,483)
	Rinast	(2,2,2)	(24,27,29)	(44,83,154)	(20,78,118)
	O(DBM)+C(FCS)	(22,22,22)	(28,28,28)	(28,28,28)	(28,28,28)
	O(DBM)+C(MCS)	(15,15,15)	(15,20,23)	(19,21,23)	(16,19,23)
	O(DBM)+C(RCS)	(11,11,11)	(11,17,19)	(16,18,20)	(13,16,21)
	O(SEQ)+C(FCS)	(16,16,16)	(26,26,27)	(26,26,27)	(25,26,27)
	O(SEQ)+C(MCS)	(9,9,9)	(13,19,22)	(17,19,22)	(13,17,22)
	O(SEQ)+C(RCS)	(3,3,3)	(10,14,17)	(13,16,20)	(8,13,19)
bridge	Trivial	(2,2,2)	(20,21,22)	(97,104,110)	(198,206,212)
	Rinast	(2,2,2)	(14,15,16)	(67,69,71)	(136,138,141)
	O(DBM)+C(FCS)	(32,32,32)	(26,27,28)	(25,26,26)	(25,26,26)
	O(DBM)+C(MCS)	(18,18,18)	(20,21,23)	(23,24,25)	(23,23,25)
	O(DBM)+C(RCS)	(13,13,13)	(17,18,21)	(22,23,25)	(21,22,25)
	O(SEQ)+C(FCS)	(24,24,24)	(23,23,23)	(24,24,24)	(24,24,24)
	O(SEQ)+C(MCS)	(10,10,10)	(15,17,20)	(21,22,24)	(21,22,24)
	O(SEQ)+C(RCS)	(3,3,3)	(11,12,17)	(19,20,24)	(18,20,24)
fischer	Trivial	(5,5,5)	(43,51,65)	(208,239,262)	(421,465,524)
	Rinast	(5,5,5)	(15,34,47)	(111,133,170)	(211,254,289)
	O(DBM)+C(FCS)	(44,44,44)	(37,41,48)	(35,39,44)	(35,38,40)
	O(DBM)+C(MCS)	(22,22,22)	(27,39,48)	(35,39,44)	(35,38,40)
	O(DBM)+C(RCS)	(15,15,15)	(21,28,34)	(30,34,35)	(30,34,37)
	O(SEQ)+C(FCS)	(35,35,35)	(32,39,47)	(35,39,44)	(35,38,40)
	O(SEQ)+C(MCS)	(13,13,13)	(22,37,47)	(35,39,44)	(35,38,40)
	O(SEQ)+C(RCS)	(5,5,5)	(15,25,33)	(30,34,34)	(30,33,34)
fischer-symmetry	Trivial	(5,5,5)	(45,48,52)	(200,246,271)	(432,507,564)
	Rinast	(5,5,5)	(16,26,34)	(97,139,168)	(216,292,339)
	O(DBM)+C(FCS)	(112,112,112)	(91,95,99)	(80,87,97)	(82,90,103)
	O(DBM)+C(MCS)	(34,34,34)	(41,47,54)	(75,86,97)	(82,90,103)
	O(DBM)+C(RCS)	(31,31,31)	(35,40,43)	(67,74,77)	(68,75,78)
	O(SEQ)+C(FCS)	(95,95,95)	(82,84,86)	(79,86,97)	(82,90,103)
	O(SEQ)+C(MCS)	(17,17,17)	(28,36,45)	(72,85,97)	(82,90,103)
	O(SEQ)+C(RCS)	(14,14,14)	(22,28,31)	(63,73,76)	(68,75,76)
train-gate	Trivial	(5,5,5)	(44,46,47)	(213,214,216)	(423,424,426)
	Rinast	(5,5,5)	(22,26,29)	(133,136,138)	(273,276,278)
	O(DBM)+C(FCS)	(44,44,44)	(44,47,54)	(54,54,54)	(54,54,54)
	O(DBM)+C(MCS)	(22,22,22)	(33,43,48)	(45,50,54)	(45,50,54)
	O(DBM)+C(RCS)	(15,15,15)	(28,31,34)	(40,47,53)	(38,47,52)
	O(SEQ)+C(FCS)	(35,35,35)	(40,44,53)	(53,54,54)	(53,54,54)
	O(SEQ)+C(MCS)	(13,13,13)	(29,41,47)	(44,50,54)	(44,50,54)
	O(SEQ)+C(RCS)	(5,5,5)	(22,28,32)	(38,46,53)	(36,46,52)
train-gate-orig	Trivial	(1,1,1)	(28,34,37)	(115,118,127)	(206,216,227)
	Rinast	(1,1,1)	(14,16,18)	(59,63,69)	(62,117,129)
	O(DBM)+C(FCS)	(24,24,24)	(22,25,28)	(28,28,28)	(28,28,28)
	O(DBM)+C(MCS)	(14,14,14)	(17,22,24)	(23,24,28)	(23,24,28)
	O(DBM)+C(RCS)	(10,10,10)	(14,16,17)	(20,22,25)	(18,22,28)
	O(SEQ)+C(FCS)	(17,17,17)	(18,24,27)	(27,27,28)	(27,27,28)
	O(SEQ)+C(MCS)	(7,7,7)	(13,21,23)	(22,24,28)	(22,24,28)
	O(SEQ)+C(RCS)	(2,2,2)	(8,14,16)	(18,20,25)	(16,21,28)
csmacd2	Trivial	(6,6,6)	(61,79,89)	(430,449,459)	(894,911,922)
	Rinast	(6,6,6)	(3,14,29)	(3,9,15)	(3,7,14)
	O(DBM)+C(FCS)	(15,15,15)	(15,18,18)	(18,18,18)	(18,18,18)
	O(DBM)+C(MCS)	(13,13,13)	(12,14,15)	(12,14,15)	(12,13,15)
	O(DBM)+C(RCS)	(9,9,9)	(9,11,13)	(9,10,11)	(9,10,11)
	O(SEQ)+C(FCS)	(13,13,13)	(14,17,17)	(17,17,17)	(17,17,17)
	O(SEQ)+C(MCS)	(11,11,11)	(11,13,14)	(11,13,14)	(11,12,14)
	O(SEQ)+C(RCS)	(6,6,6)	(8,9,11)	(8,9,9)	(8,8,9)
tdma	Trivial	(17,17,17)	(95,95,95)	(427,431,435)	(836,847,857)
	Rinast	(7,7,8)	(18,22,32)	(57,65,89)	(155,195,249)
	O(DBM)+C(FCS)	(54,54,54)	(54,54,54)	(54,54,54)	(54,54,54)
	O(DBM)+C(MCS)	(24,24,24)	(20,20,20)	(20,23,36)	(20,24,37)
	O(DBM)+C(RCS)	(17,17,17)	(20,20,20)	(19,21,30)	(20,22,34)
	O(SEQ)+C(FCS)	(45,45,45)	(51,51,51)	(50,52,53)	(53,54,54)
	O(SEQ)+C(MCS)	(15,15,15)	(17,17,17)	(16,21,35)	(20,24,37)
	O(SEQ)+C(RCS)	(7,7,7)	(17,17,17)	(16,20,29)	(20,23,35)

References

1. Abdelli, A.: Improving the construction of the DBM over approximation of the state space of real-time preemptive systems. Acta Cybern. **20**, 347–384 (2012)
2. Aho, A.V., Garey, M.R., Ullman, J.D.: The transitive reduction of a directed graph. SIAM J. Comput. **1**(2), 131–137 (1972)
3. André, É., Arcaini, P., Gargantini, A., Radavelli, M.: Repairing timed automata clock guards through abstraction and testing. In: Tests and Proofs, pp. 129–146 (2019)
4. Audemard, G., Cimatti, A., Kornilowicz, A., Sebastiani, R.: Bounded model checking for timed systems. In: Formal Techniques for Networked and Distributed Sytems—FORTE 2002, pp. 243–259 (2002)
5. Behrmann, G., Bengtsson, J., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Uppaal implementation secrets. In: Formal Techniques in Real-Time and Fault-Tolerant Systems, pp. 3–22 (2002)
6. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 312–326 (2004)
7. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal 4.0 (2006)
8. Bengtsson, J.: Clocks, dbms and states in timed systems. Ph.D. thesis, Uppsala University (2002)
9. Bücker, H.M., Petera, M., Vehreschild, A.: Code optimization techniques in source transformations for interpreted languages. In: Advances in Automatic Differentiation, pp. 223–233 (2008)
10. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods Syst. Des. **19**, 7–34 (2001)
11. David, A.: Uppaal DBM library programmer's reference (2006)
12. Dubois, D., Fargier, H., Prade, H.: Possibility theory in constraint satisfaction problems: Handling priority, preference and uncertainty. Appl. Intell. **6**, 287–309 (1996)
13. Ehlers, R., Fass, D., Gerke, M., Peter, H.: Fully symbolic timed model checking using constraint matrix diagrams. In: 2010 31st IEEE Real-Time Systems Symposium, pp. 360–371 (2010)
14. Evangelista, S., Pradat-Peyre, J.F.: Memory efficient state space storage in explicit software model checking. In: Model Checking Software, pp. 43–57 (2005)
15. Fages, F., Rizk, A.: From model-checking to temporal logic constraint solving. In: Principles and Practice of Constraint Programming—CP 2009, pp. 319–334 (2009)
16. Hertzberg, J., Güsgen, H.W., Voβ, A., Fidelak, M., Voβ, H.: Relaxing constraint networks to resolve inconsistencies. In: Künstliche Intelligenz, pp. 61–65 (1988)
17. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.D.: Software rejuvenation: analysis, module and applications. In: Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers, pp. 381–390 (1995)
18. Jagtap, P., Abdi, F., Rungger, M., Zamani, M., Caccamo, M.: Software fault tolerance for cyber-physical systems via full system restart. ACM Trans. Cyber-Phys. Syst. **4**(4), 1–20 (2020)
19. Jensen, H., Larsen, K., Skou, A.: Modelling and analysis of a collision avoidance protocol using SPIN and UPPAAL. BRICS Rep. Ser. **3**(24), 1–20 (1996)
20. Jonas Rinast: OMC framework. https://www.tuhh.de/sts/research/model-checking-abstract-interpretation/online-model-checking.html
21. Kaplan, S.F., Smaragdakis, Y., Wilson, P.R.: Trace reduction for virtual memory simulations. Tech. rep. (1998)
22. Kong, F., Xu, M., Weimer, J., Sokolsky, O., Lee, I.: Cyber-physical system checkpointing and recovery. In: 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS), pp. 22–31 (2018)
23. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: compact data structure and state-space reduction. In: Proceedings Real-Time Systems Symposium, pp. 14–24 (1997)
24. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Clock difference diagrams. Nordic J. Comput. **6**(3), 271–298 (1999)
25. Liu, Y., Leangsuksun, C., Song, H., Scott, S.L.: Reliability-aware checkpoint/restart scheme: a performability trade-off. In: 2005 IEEE International Conference on Cluster Computing, pp. 1–8 (2005)
26. Lonn, H., Pettersson, P.: Formal verification of a TDMA protocol start-up mechanism. In: Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems, pp. 235–242 (1997)
27. Makowsky, J.A., Ravve, E.V.: Incremental model checking for decomposable structures. In: Mathematical Foundations of Computer Science 1995, pp. 540–551 (1995)
28. Mohan, C.: A cost-effective method for providing improved data availability during DBMS restart recovery after a failure. In: Proceedings of the 19th International Conference on Very Large Data Bases, VLDB '93, pp. 368–379 (1993)
29. Pettersson, P.: Modelling and verification of real-time systems using timed automata: Theory and practice. Ph.D. thesis, Department of Computer Systems, Uppsala University (1999)
30. Rinast, J.: An online model-checking framework for timed automata. Ph.D. thesis, Hamburg University of Technology (2015)
31. Rinast, J., Schupp, S., Gollmann, D.: State space reconstruction in UPPAAL: an algorithm and its proof. Int. J. Adv. Syst. Meas. **7**(1–2), 91–102 (2014)
32. Salah, R.B., Bozga, M., Maler, O.: On interleaving in timed automata. In: CONCUR 2006—Concurrency Theory, pp. 465–476 (2006)
33. Salehi, M., Khavari Tavana, M., Rehman, S., Shafique, M., Ejlali, A., Henkel, J.: Two-state checkpointing for energy-efficient fault tolerance in hard real-time systems. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **24**(7), 2426–2437 (2016)
34. Sascha Lehmann: Uppyyl simulator. https://github.com/S-Lehmann/uppyyl-simulator
35. Sascha Lehmann: Uppyyl state constructor. https://github.com/S-Lehmann/uppyyl-state-constructor
36. Sascha Lehmann: Uppyyl state constructor experiments. https://github.com/S-Lehmann/uppyyl-state-constructor-experiments
37. Sorea, M.: Bounded model checking for timed automata. Electron. Notes Theor. Comput. Sci. **68**(5), 116–134 (2003)
38. Sorin, D.J., Martin, M.M.K., Hill, M.D., Wood, D.A.: Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In: Proceedings 29th Annual International Symposium on Computer Architecture, pp. 123–134 (2002)
39. Zhao, Y., Rammig, F.: Online model checking for dependable real-time systems. In: 2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, pp. 154–161 (2012)
40. Ziv, A., Bruck, J.: An on-line algorithm for checkpoint placement. IEEE Trans. Comput. **46**(9), 976–985 (1997)