Formal Methods in System Design (2021) 59:136–169
https://doi.org/10.1007/s10703-022-00397-5

Automated repair for timed systems

Martin Kölbl[1] · **Stefan Leue**[1] · **Thomas Wies**[2]

Received: 21 May 2021 / Accepted: 13 September 2022 / Published online: 5 October 2022
© The Author(s) 2022

Abstract
We present algorithms and techniques for the repair of timed system models, given as networks of timed automata (NTA). The repair is based on an analysis of timed diagnostic traces (TDTs) that are computed by real-time model checking tools, such as UPPAAL, when they detect the violation of a timed safety property. We present an encoding of TDTs in linear real arithmetic and use the MaxSMT capabilities of the SMT solver Z3 to suggest a minimal number of possible syntactic repairs of the analyzed model. The suggested repairs include modified values for clock bounds in location invariants and transition guards, adding or removing clock resets, etc. We then present an admissibility criterion, called functional equivalence, which ensures that the proposed repair preserves the functional behavior of the considered NTA. We discuss a proof-of-concept tool called TarTar that we have developed, implementing the repair and admissibility analysis, and give insights into its design and architecture. We evaluate the proposed repair technique on faulty mutations generated from a diverse suite of case studies taken from the literature. We show that TarTar can admissibly repair for 69–88% of the seeded errors in the considered system models.

Keywords Timed automata · Automated repair · Admissibility of repair · TarTar tool

1 Introduction

The analysis of system design models using model checking technology is an important step in the system design process. It enables the automated verification of system properties against given design models. The automated nature of model checking facilitates the integration of the verification step into the design process since it requires no further

✉ Martin Kölbl
martin.koelbl@uni-konstanz.de

Stefan Leue
stefan.leue@uni-konstanz.de

Thomas Wies
wies@cs.nyu.edu

[1] University of Konstanz, Konstanz, Germany

[2] New York University, New York, USA

Konstanzer Online-Publikations-System (KOPS)
URL: http://nbn-resolving.de/urn:nbn:de:bsz:352-2-1gelmzvrdev3s7

intervention of the designer once the model has been formulated and the property has been specified.

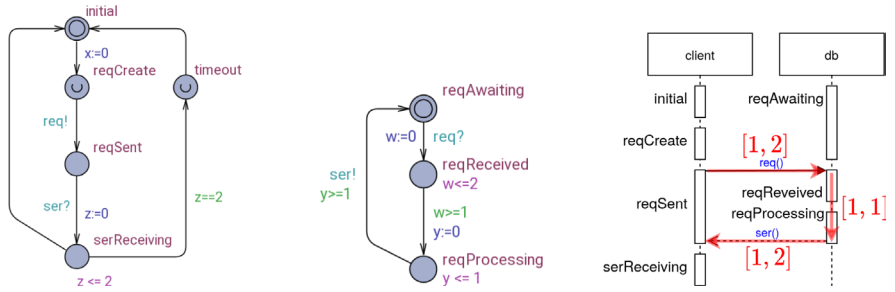
Often it is sufficient to abstract from real time aspects when checking system properties, in particular when the focus is on functional aspects of the system. However, in certain domains, non-functional properties of the system such as response times or the timing of periodic behavior play an important role. In such cases, it is necessary to incorporate real time aspects into the models and the specification, as well as to use specialized real-time model checking tools, such as UPPAAL [10], Kronos [44] or opaal [16] during the verification step.

Next to the automatic nature of model checking, the ability to return counterexamples, in real-time model checking often referred to as timed diagnostic traces (TDT), is a further practical benefit of the use of model checking technology. A TDT describes a timed sequence of steps that lead the design model from the initial state of the system into a state violating a real-time property. However, a TDT alone neither constitutes a causal explanation of the property violation, nor does it provide hints as to how to correct the model.

We present an automated method that computes proposals for possible repairs of a network of timed automata (NTA) that avoid the violation of a timed safety property. Consider the TDT depicted as a time annotated sequence diagram [9] in Fig. 1c. This scenario describes a simple message exchange where the process `client` sends a message `req` to the process `db` which, after some processing steps returns a message `ser` to `client`. Assume a requirement on the system stating that the time from sending `req` to receiving `ser` is not to be more than 4 time units. Further assume that the timing interval annotations on the sequence diagram represent the minimum and maximum time for the message transmission and processing steps that the NTA, from which the diagram has been derived, permits. It is then easy to see that it is possible to execute the system in such a way that this property is violated.

Various changes to the underlying NTA model depicted in Fig. 1 may avoid this property violation. We present analyses that can suggest a whole range of repairs in addition to clock bound variation, such as modifying clock bounds, comparison operators in constraints, clock references, clock resets, and location urgency. Examples of repairs computed for the model in Fig. 1 are:

– Reducing the maximum time it takes to transmit `ser` message to be at most 1 time unit by exchanging $z <= 2$ with $z <= 1$.



(a) Timed Automata `client` (b) Timed Automata `db` (c) TDT `tdt`

Fig. 1 Network of timed automata—running example

- Exchanging the comparison operator in the constraint $w \geq 1$ to $w < 1$ ensures that the time to send a request is below 1 time unit.
- An exchange of clock z in $z \leq 2$ with clock y restricts the time of processing and receiving the response to at most 2 time units.
- To reset the clock y on the previous transition instead ensures that the time for sending and processing the request is below 1 time unit.
- Making the location *serReceiving* urgent reduces the time to receive a response to 0.

Proposing such changes to the model may either serve to correct clerical mistakes made during the editing of the model, or point to necessary changes in the dimensioning of its time resources, thus contributing to improved design space exploration.

The repair method described in this paper relies on an encoding of a TDT as a constraint system in linear real arithmetic. This encoding provides a symbolic abstract semantics for the TDT by constraining the sojourn time of the NTA in the locations visited along the trace. The constraint system is then augmented by auxiliary model variation variables which represent syntactic changes to the NTA model, for instance, the variation of a location invariant condition or a transition guard. We assert that the thus modified constraint system implies the non-reachability of a violation. At the same time, we assert that the model variation variables have a value that implies that no change of the NTA model will occur, for instance, by setting a clock bound variation variable to 0. This renders the resulting constraint system unsatisfiable.

In order to compute a repair, we derive a partial MaxSMT instance by turning the constraints that disable any repair into soft constraints. We solve this MaxSMT instance using the SMT solver Z3 [18]. If the MaxSMT instance admits a solution, the resulting model provides values of the model variation variables. These values indicate a repair of the NTA model which entails that along the sequence of locations represented by the TDT, the property violation will no longer be reachable.

In a next step it is necessary to check whether the computed repair is an admissible repair in the context of the full NTA. This is important since the repair was computed locally with respect to only a single given TDT. Thus, it is necessary to define a notion of *admissibility* that is reasonable and helpful in this setting. To this end, we propose the notion of *functional equivalence* which states that as a result of the computed repair, neither erstwhile existing functional behavior will be eliminated, nor will new functional behavior be added. Functional behavior in this sense is represented by the languages accepted by the untimed automata of the unrepaired and the repaired NTAs. Functional equivalence is then defined as equivalence of the languages accepted by these automata. We propose a zone-based automaton construction for implementing the functional equivalence test that is efficient in practice.

We have implemented our proposed method in a proof-of-concept tool called TarTar. Our evaluation of TarTar is based on several non-trivial NTA models taken from the literature, including the frequently considered pacemaker model [29]. For each model, we automatically generated mutants by injecting syntactic modifications which we then model checked using UPPAAL and repaired using TarTar. The evaluation shows that our technique is able to compute an admissible repair 69–88% of the detected faults.

1.1 Related work

There are relatively few results available on a formal treatment of TDTs. The zone-based approach to real-time model checking, which relies on a constraint-based abstraction of the state space, is proposed in [23]. The use of constraint solving to perform reachability

analysis for NTAs is described in [45]. This approach ultimately lead to the on-the-fly reachability analysis algorithm used in UPPAAL [12]. Henning et al. [17] defines the notion of a time-concrete UPPAAL counterexample. Work documented in [40] describes the computation of concrete delays for symbolic TDTs. The above cited approaches address neither fault analysis nor repair for TDTs. Our use of MaxSMT solvers for computing minimal repairs is inspired by the use MaxSAT solvers for fault localization in C programs, which was first explored in the BugAssist tool [28]. We present an approach that uses a counterexample to prevent bad behavior while the technique in [11] computes repairs to ensure missing expected behavior. The algorithm presented in [19] repairs a TA by adding time constraints, whereas the approach in this paper modifies existing time constraints. Our approach also shares some similarities with syntax-guided synthesis [2, 41], which has also been deployed in the context of program repair [35]. One key difference is how we determine the admissibility of a repair in the overall system, which takes advantage of the semantic restrictions imposed by timed automata. The modification of time constraints by extra variables can also be interpreted as a parametric timed automata (PTA). A PTA is used in [1] to repair a TA that does not behave according to an oracle. Each of the generated repairs can only prevent a finite number of explicit timed traces whereas the repairs generated by our approach can prevent an infinite number of symbolically encoded traces. The algorithm in [22] checks whether a synthesized TA exists for a given set of properties. If this automaton does not exist, mixed integer linear programming is used to modify bounds in the specified properties such that a TA can be synthesized. The modifications of bounds in STL specifications is a different goal than the modification of bounds in the TA that we propose.

1.2 Contributions

We augment in this work the original publications in [31] and [32] on computing syntactic repairs for timed systems. More examples in this work illustrate the approach. We also added proofs for the theorems, extended the definitions given in the preliminaries and harmonized the presentation.

1.3 Structure of the paper

We will introduce the automata and real-time concepts needed in our analysis in Sect. 2. In Sect. 3, we present the logical formalization of TDTs. The repair and admissibility analyses are presented in Sect. 4 and 5, respectively. We present the implementation in the tool TarTar in Sect. 6. In Sect. 7, we report on experimental evaluation and case studies. We provide a conclusion and perspectives for future research in Sect. 8.

2 Preliminaries

The timed automaton model that we use in this paper is adapted from [12]. Given a set of *clocks* C, we denote by $\mathcal{B}(C)$ the set of all *clock constraints* over C, which are conjunctions of *atomic clock constraints* of the form $c \sim n$, where $c \in C$, $\sim \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$. For the remainder of this section, we fix a finite set of clocks C.

Definition 1 (Timed Automaton (TA) [12]) A Timed Automaton T is a tuple $T = (L, l^0, \Sigma, \Theta, I)$ where L is a finite set of locations, $l^0 \in L$ is an initial location, Σ is a finite set of actions, $\Theta \subseteq L \times \mathcal{B}(C) \times \Sigma \times 2^C \times L$ is the transition relation, and $I : L \to \mathcal{B}(C)$ denotes a labeling of locations with clock constraints, referred to as location invariants. For $\theta \in \Theta$ with $\theta = (l, g, a, r, l')$, we refer to g as the *guard* of θ and to r as its *clock resets*.

The operational semantics of T is given by a timed transition system (TTS) in Definition 2. A *clock valuation* u is a function $C \to \mathbb{R}_{\geq 0}$. For a clock constraint b, we write $u \vDash b$ iff b evaluates to true in u. There are two types of transitions. An *action transition* models the execution of an action whose guard is satisfied. These transitions are instantaneous and reset the specified clocks. The passing of time in a location is modeled by *delay transitions*. Both types of transitions guarantee that location invariants are satisfied in the pre and post state.

Definition 2 (Timed Transition System (TTS) [6]) For a Timed Automaton $(L, l^0, \Sigma, \Theta, I)$, a timed transition system is a tuple $H = (S, s_0, \Sigma, \to)$ where $S = \{(l, u) \in L \times \mathcal{R}^C_{\geq 0} \mid u \vDash I(l)\}$, s_0 is the initial state (l^0, u_0) such that u_0 maps all clocks to 0, and a transition $(l, u) \xrightarrow{t} (l', u')$ is in $\to$ iff

- (action transition) $t = (l, g, a, r, l') \in \Theta$, $u \vDash I(l) \wedge g$, $u' \vDash I(l')$ and for all clocks $c \in C$, $u'(c) = 0$ if $c \in r$ and $u'(c) = u(c)$ otherwise; or
- (delay transition) $t \in \mathbb{R}_{\geq 0}$, $l = l'$, $u \vDash I(l)$, $u' \vDash I(l)$ and $u' = u + t$.

An urgent location is a location that has to be left again without taking a delay transition [6]. Urgent locations are syntactic sugar of Uppaal and can be expressed as an additional clock p which is reset with entering the location and a location invariant $p = 0$.

A *run* [12] of a TTS with initial state s_0 is a sequence of states and actions of the form $s_0 \xrightarrow{t_1} \xrightarrow{a_1} s_1 \xrightarrow{t_2} \xrightarrow{a_2} \dots$ with $s_i = (l_i, u_i)$ where every t_i is in $\mathbb{R}_{\geq 0}$ with $(l_i, u_i) \xrightarrow{t_i} (l_i, u_{i+1})$ in $\to$, and every a_i is in Σ with $(l_i, u_{i+1}) \xrightarrow{(l_i, g_i, a_i, r, l_{i+1})} (l_{i+1}, u_{i+1})$ in $\to$. A *timed trace* [12] is a sequence of timed actions $\xi = (t'_1, a_1), (t'_2, a_2), \dots$ that is generated by a run of a TTS associated with a TA, where $t'_i = \sum_{0<j\leq i} t_j$. We capture families of timed traces that perform the same sequence of action transitions but differ in their delay transitions by the notion of a *symbolic timed trace*.

Definition 3 (Symbolic Timed Trace (STT)) A symbolic timed trace (STT) of T is a sequence of actions $Y = \theta_0, \dots, \theta_{n-1}$. A *realization* of Y is a sequence of delay values $\delta_0, \dots, \delta_n$ such that there exists states $s_0, \dots, s_n, s_{n+1}$ with $s_i \xrightarrow{\delta_i} \xrightarrow{\theta_i} s_{i+1}$ for all $i \in [0, n)$ and $s_n \xrightarrow{\delta_n} s_{n+1}$. We say that a STT is *feasible* if it has at least one realization.

The timed language for a TA T is the set of all its timed traces, which we denote by $\mathcal{L}_T(T)$. The untimed language of T consists of words over T's alphabet Σ so that there exists at least one timed trace of T forming this word. Formally, for a timed trace $\xi = (t_1, a_1), (t_2, a_2) \dots$, the untime operator $\mu(\xi)$ returns an untimed trace $\xi_\mu = a_1 a_2 \dots$. We define the untimed language $\mathcal{L}_\mu(T)$ of the TA T as $\mathcal{L}_\mu(T) = \{\mu(\xi) \mid \xi \in \mathcal{L}_T(T)\}$. We represent a finite untimed language using a *Nondeterministic Finite Automaton* and an infinite untimed language using a *Nondeterministic Büchi Automaton*.

Definition 4 (Nondetermistic Finite Automaton (NFA) [6]) A nondeterministic finite automaton is a tuple $M = (S, \Sigma, \rightarrow, S_0, F)$ where S denotes a finite set of states, Σ denotes an alphabet, $\rightarrow \subseteq S \times \Sigma \times S$ denotes a transition relation, $S_0 \subseteq S$ denotes the set of initial states, and $F \subseteq S$ denotes the set of acceptance states. We write $s \xrightarrow{a} s'$ when $(s, a, s') \in \rightarrow$. An execution of M is a sequence $s_0 \xrightarrow{a_0} s_1 \ldots \xrightarrow{a_{n-1}} s_n$ such that $s_0 \in S_0$ and $s_n \in F$. A run $s_0, s_1, \ldots, s_n$ of M is the projection of an execution $s_0 \xrightarrow{a_0} s_1 \ldots \xrightarrow{a_{n-1}} s_n$ on the state sequence. M accepts a word $w = a_0, a_1, \ldots, a_{n-1}$ when an execution $s_0 \xrightarrow{a_0} s_1 \ldots \xrightarrow{a_{n-1}} s_n$ exists. The language $\mathcal{L}_f(M)$ recognized by M is the set of (finite length) words accepted by M.

Definition 5 (Nondeterministic Büchi Automaton (NBA) [6]) An NFA $B = (S, \Sigma, \rightarrow, S_0, F)$ is called a Büchi automaton in case it is used with an acceptance condition for infinite input sequences. For an infinite sequence r of states, let $\inf(r) = \{s \in S \mid r_i = s \text{ for infinitely many } i\}$. An NBA B accepts an infinite word $w = a_1 a_2 \ldots, w \in \Sigma^\omega$, iff B has an infinite run $r = s_0 s_1 \ldots$ of states on w such that $s_0 \in S_0$, $\inf(r) \cap F \neq \emptyset$ and $\forall i \in \mathbb{N}_0^+, (s_i, a_{i+1}, s_{i+1}) \in \rightarrow$. The language $\mathcal{L}(B) \subseteq \Sigma^\omega$ recognized by an NBA B is the set of infinite words accepted by B. $\text{pref}(\mathcal{L}(B))$ denotes the set of all finite prefixes of words in $\mathcal{L}(B)$.

For a given NFA or NBA M, the *closure* $cl(M)$ denotes the automaton obtained from M by turning every state into an accepting state. We call M closed iff $M = cl(M)$. Notice that an NBA accepts a safety language if and only if it is closed [5].

We use *zones* as given in Definition 6 in order to abstract an infinite-state TTS into a finite-state *Zone Automaton*, c.f. Definition 7.

Definition 6 (Zone [6]) A diagonal constraint is an extended clock constraint of the form $x - y \sim n$ with two clocks x and y. For a finite set C of clocks, let $[\![\varphi]\!]_C = \{u \in \mathbb{R}^C_{\geq 0} \mid u \vDash \varphi\}$ denote the set of clock evaluations u satisfying a conjunction φ of diagonal clock constraints. A subset $z \subseteq \mathbb{R}^C_{\geq 0}$ is called a zone if there exists φ for which $z = [\![\varphi]\!]_C$.

Definition 7 (Zone Automaton (adapted from [6])) Assume a Timed Automaton $T = (L, l^0, \Sigma, \Theta, I)$. We define the zone automaton $[\![T]\!]_Z = (S_Z, s_Z^0, \Sigma_Z, \Theta_Z)$ for T such that $S_Z = \{(l, z) \mid l \in L, z \subseteq \mathbb{R}^C_{\geq 0} \text{ is a zone}\}$, $s_Z^0 = (l^0, \{u_0\})$, $\Sigma_Z = \Sigma \cup \{\delta\}$ and a transition relation $\Theta_Z \subseteq S_Z \times \Sigma_Z \times S_Z$. Let $z^\uparrow = \{u + d \mid u \in z, d \in \mathbb{R}_0^+\}$, and let $z[r]$ denote the clock reset for a clock set r in a zone z such that in the resulting zone every clock in r evaluates to 0. We write $l \overset{a}{\rightsquigarrow} l'$ for a transition $(l, a, l') \in \Theta_Z$. The transition relation Θ_Z is the smallest relation that satisfies the following rules:

1. If $(l, z) \in S_Z$, then $(l, z) \overset{\delta}{\rightsquigarrow} (l, z^\uparrow \cap [\![I(l)]\!]_C)$.
2. If $l \xrightarrow{\varphi, a, r} l' \in \Theta$, then $(l, z) \overset{a}{\rightsquigarrow} (l', (z \cap [\![\varphi]\!]_C)[r] \cap [\![I(l')]\!]_C)$.

A trace $a_0 \ldots a_n$ is a trace of Θ_Z iff for $0 \leq i \leq n$, $l_i \overset{a_i}{\rightsquigarrow} l_{i+1}$ exists and $l_0 = l^0$. We call a trace of Θ_Z a symbolic trace.

$\mathcal{L}_T([\![T]\!]_Z)$ denotes the set of timed traces with time delays that satisfy the zones of at least one symbolic trace of $[\![T]\!]_Z$.

2.1 Property specification

We focus on the analysis of timed safety properties, which we characterize by an invariant formula that has to hold for all reachable states of a TA. These properties state, for instance, that there are certain locations in which the value of a clock variable is not above, equal to or below a certain (integer) bound. Formally, let $T = (L, l^0, C, \Sigma, \Theta, I)$ be a TA. A *timed safety property* Π is a Boolean combination of atomic clock constraints and *location predicates* $@l$ where $l \in L$. A location predicate $@l$ holds in a state (l', u) of T iff $l' = l$. We say that an STT Y witnesses a violation of Π in T if there exists a realization of Y whose induced final state does not satisfy Π. We refer to such an STT as a *timed diagnostic trace* (TDT) of T for Π.

T satisfies Π iff all its reachable states satisfy Π. This problem can be decided using model checking tools such as Kronos [44] and UPPAAL [10]. UPPAAL in particular computes a finite abstraction of the state space of an NTA using zones for the graph construction. Reachability analysis is then performed by an on-the-fly search of the zone graph. If the property is violated, the tool generates a feasible TDT that witnesses the violation. The objective of our work is to analyze TDTs and to propose repairs for the property violation that they represent. We use TDTs generated by the UPPAAL tool in our implementation, but we maintain that our results can be adapted to any other tool producing TDTs.

We further note that UPPAAL takes a *network of timed automata* (NTA) as input, which is a CCS [38] style parallel composition of timed automata $T_1 \mid \ldots \mid T_n$, yielding a TA. Since our analysis and repair techniques focus on timing-related errors rather than synchronization errors, we use TAs rather than NTAs in our formalization. However, our repair analysis can be directly applied to the TA obtained by the parallel composition and the implementation in TarTar works directly on NTAs.

Example 1 The running example that we use throughout the paper consists of an NTA containing two timed automata, depicted in Fig. 1. As alluded to in the introduction, the TAs `client` and `db` synchronize via the exchange of messages modeled by the pairs of send and receive actions `req!` and `req?` as well as `ser!` and `ser?`, respectively. The transmission time of the `req` message is controlled by the clock variable `w` and can range between 1 and 2 time units. This is achieved by the location invariant `w <= 2` on the `reqReceived` location in `db` together with the transition guard `w >= 1` on the transition from `reqReceived` to `reqProcessing`. A similar mechanism using clock variable `z` is used to constrain the timing of the transfer of message `ser` to be within 1 and 2 time units. The processing time in `db` is constrained to exactly 1 time unit by the location invariant `y <= 1` and the transition guard `y >= 1`. In `client`, a transition to location `timeout` can be triggered when the guard `z==2` is satisfied in location `serReceiving`. The clock variable `x`, which is not reset until the next `req` message is sent, is recording the time that has elapsed since sending `req` and is used in location `serReceiving` in order to verify if more than 4 time units have passed since `req` was sent. Every transition in the system is labeled with an action τ. The timed safety property that we will consider for our example is $\Pi = \neg @\texttt{client.serReceiving} \vee (x \leq 4)$. For the violation of this property, UPPAAL produces the TDT $S = \theta_0 \ldots \theta_3$ where

$\theta_0 = ((\texttt{initial}, \texttt{reqAwaiting}), \emptyset, \tau, \{x\}, (\texttt{reqCreate}, \texttt{reqAwaiting}))$

$\theta_1 = ((\texttt{reqCreate}, \texttt{reqAwaiting}), \emptyset, \tau, \{w\}, (\texttt{reqSent}, \texttt{reqReceived}))$

$\theta_2 = ((\texttt{reqSent}, \texttt{reqReceived}), \{w \geq 1\}, \tau, \{y\}, (\texttt{reqSent}, \texttt{reqProcessing}))$

$\theta_3 = ((\texttt{reqSent}, \texttt{reqProcessing}), \{y \geq 1\}, \tau, \{z\}, (\texttt{serReceiving}, \texttt{reqAwaiting})).$

3 Logical encoding of timed diagnostic traces

We present a logical formalization of the concept of Timed Diagnostic Traces (TDTs) generated by a real-time model checking tool. Practical model checking tools generate these traces in a textual format. In the case of UPPAAL, which we use for generating the traces used in the analysis in this paper, symbolic TDTs are stored in an XML formatted file. The formalization that we develop will enable a logic encoding and automated analysis of TDTs.

3.1 Timed diagnostic trace (TDT)

A TDT is a linear data structure that represents steps from an initial state of the system leading into a property violating state of an NTA. Given an NTA N and property Π this means that the final state in the TDT is violating Π. Figure 2 represents an excerpt of a TDT obtained from the UPPAAL tool and illustrates the violation of the property $\Pi = \neg @\texttt{client.serReceiving} \vee (x < 4)$ by the NTA in Fig. 1.

The TDT contains information regarding the locations, location invariants, transition guards and reset operations affected by the execution of transitions. It also contains a representation of the difference bound matrices (DBMs) that are attached to every location in the NTA. The TDTs that we consider are symbolic in the sense that they do not give concrete time values at which certain locations are reached, but rather constraints on the clock values that need to be met for the property violating state to be reachable. For instance, consider the DBM table entry < clockboundclock1 = εsys.xε clock2="sys.t(0)" bound="5" comp = ε <= ε/ > in Fig. 2 which represents the constraint on x in the state serReceiving as $x \leq 5$. In other words, no concrete value of x when entering the violating state is given.

The objective of our analysis is to identify possible syntactic changes to the NTA model in order to propose repairs for the violation of timed reachability properties. The symbolic representation of the valid clock assignments per location in the TDT given by DBMs, as is done for instance by UPPAAL, is not useful for this purpose. This is due to the fact that DBMs perform operations and optimizations on the constraints when they construct zones [12]. As a result, bound values in the UPPAAL code can not necessarily be directly associated with bound entries in the DBM. It is therefore necessary to define our own logic representation of the TDT. This representation is based on the following observations:

- The TDT represents an equivalence class of runs of the NTA in the sense that all runs in this class traverse the same sequence of action transitions. This yieds a sequence $l_0 ... l_n$ of NTA locations that are reached during the execution of the TDT.
- For every location l_j, there is a corresponding sojourn time in this location that we denote using a positive real valued delay variable δ_j.

```
<trace initial_node="State1" trace_options="symbolic">
<system>
<clock name="t(0)" id="sys.t(0)"/>
<clock name="x" id="sys.x"/>
...
<process id="db" name="db">
<location id="db.reqAwaiting" name="reqAwaiting">...</location>
<location id="db.reqProcessing" name="reqProcessing"><![CDATA[1
    && y <= 1]]>
</location>
...
<edge id="db.Edge3" from="db.Processing" to="db.reqAwaiting">
<guard>y >= 1</guard>
<sync>ser!</sync>
<update>z:=0</update></edge>
...
<edge id="client.Edge3" from="client.reqSent"
    to="client.serReceiving">
<guard>1</guard>
<sync>ser?</sync>
<update>1</update></edge>
...
<node id="State4" location_vector="LocVec4" .../>
<node id="State5" location_vector="LocVec5" .../>
...
<location_vector id="LocVec4" locations="client.reqSent
    db.reqProcessing"/>
<location_vector id="LocVec5" locations="client.serReceiving
    db.reqAwaiting"/>
...
<dbm_instance id="DBM5">
<clockbound clock1="sys.x" clock2="sys.t(0)" bound="5" comp="<="/>
</dbm_instance>
...
<transition from="State4" to="State5" edges="client.Edge3 db.Edge3 "/>
</trace>
```

Fig. 2 Excerpt from the XML representation of a symbolic TDT generated by UPPAAL

- The value of every clock variable c in location l_j, which we denote by c_j, is constrained by the value of c_j when entering the location plus the delay δ_j and any location invariant constraint that refers to c in location l_j. Notice that we need to be able to refer to the values of clocks in individual location, which is why we choose a Static Single Assignment form encoding of the clock variables.
- For any clock c in the successor location l_{j+1}, the value c_{j+1} is determined by the value c_j plus the sojourn time δ_j, if the clock is not being reset during the transition from l_j to l_{j+1}, or 0 otherwise.
- The sojourn time δ_j in a location l_j is constrained by the clock invariant conditions of the NTA component TAs that perform the computation step triggering the transition into location l_{j+1}. This means that the clock value c_j plus δ_j are bounded by any clock constraint that refers to a clock variable c. For instance, in the TDT in Fig. 2, the location LocVec4 contains the TA location `reqProcessing` with the invariant y <= 1. Hence, the sojourn time in LocVec4 is constrained by $y_4 + \delta_4 \leq 1$.
- The transition from location l_j to location l_{j+1} is guarded by clock constraints on some of the clocks c, if at least one is given, and otherwise by *true*. This means

that the time when entering location l_j as well as the time elapsing while in location l_j are constrained by these guards. We represent this by adding constraints of the form $c_j + \delta_j \sim \beta$ with $\beta \in \mathbb{N}$ to the symbolic constraint system. Notice that only the transition guards of the component TAs that performed the step leading to the step in the TDT need to be taken into account. In case the step performs a synchronization, the conjunctions of the transition guards for all clock variables on the sending and the receiving transitions need to be considered. To illustrate this point consider the < transitionfrom = εState4εto = εState5ε edges = εclient.Edge3db.Edge3ε/ > entry in the TDT in Fig. 2 which specifies that both the client and db component TAs are executing a step. The synchronization is indicated by the < sync > ser! < /sync > and < sync > ser? < /sync > entries which refer to edges in the transition graph of the TAs engaged in this synchronous communication step. The < guard > 1 < /guard > and < guard > y >= 1 < /guard > entries in these edges imply that this transition needs to be guarded by the condition $true \wedge y \geq 1$, where the *true* corresponds to the transition guard in the client TA and $y \geq 1$ is the guard in the db TA.
- When a location l_j is labeled as urgent, l_j has to be left immediately and we constrain the delay δ_j in this location to have value 0.

3.2 TDT formalization

Our analysis relies on a logical encoding of TDTs in the theory of quantifier-free linear real arithmetic. For the remainder of this paper, we fix a TA $T = (L, l^0, C, \Sigma, \Theta, I)$ with a safety property Π and assume that $Y = \theta_0, \ldots, \theta_{n-1}$ is an STT of T. We use the following notation for our logical encoding where $j \in [0, n+1]$ is a position in a realization of Y and $c \in C$ is a clock:

- l_j denotes the location of the pre state of θ_j for $j < n$ and the location of the post state of θ_{j-1} for $j = n$.
- c_j denotes the value of clock variable c when reaching the state at position j.
- δ_j denotes the delay of the delay transition leaving the state at position $j \leq n$.
- $reset_j$ denotes the set of clock variables that are being reset by action θ_j for $j < n$.
- $ibounds(c, l)$ denotes the set of pairs $(\beta, \sim)$ such that the atomic clock constraint $c \sim \beta$ appears in the location invariant $I(l)$.
- $gbounds(c, \theta)$ denotes the set of pairs $(\beta, \sim)$ such that the atomic clock constraint $c \sim \beta$ appears in the guard of action θ.
- *urgent* denotes the set of indices of those locations l_j, $1 \leq j \leq n$, which are marked as *urgent* in the TA model from which Y has been derived.

To illustrate the use of *ibounds*, assume location l to be labeled with invariants $x > 2 \wedge x \leq 4 \wedge y \leq 1$, then $ibounds(x, l) = \{(2, >), (4, \leq)\}$. The usage of *gbounds* is accordingly.

Definition 8 The *timed diagnostic trace constraint system* (TDTCS) associated with STT Y is the conjunction $\mathcal{T}$ of the following constraints:

$$\mathcal{C}_0 \equiv \bigwedge_{c \in C} c_0 = 0 \qquad \text{(clock initialization)}$$

$$\mathcal{A} \equiv \bigwedge_{j \in [0,n]} \delta_j \geq 0 \qquad \text{(time advancement)}$$

$$\mathcal{R} \equiv \bigwedge_{c \in reset_j,} c_{j+1} = 0 \qquad \text{(clock resets)}$$

$$\mathcal{D} \equiv \bigwedge_{c \notin reset_j} c_{j+1} = c_j + \delta_j \qquad \text{(sojourn time)}$$

$$\mathcal{I} \equiv \bigwedge_{(\beta,\sim) \in ibounds(c,l_j)} c_j \sim \beta \wedge c_j + \delta_j \sim \beta \qquad \text{(location invariants)}$$

$$\mathcal{G} \equiv \bigwedge_{(\beta,\sim) \in gbounds(c,\theta_j)} c_j + \delta_j \sim \beta \qquad \text{(transition guards)}$$

$$\mathcal{U} \equiv \bigwedge_{j \in urgent} \delta_j = 0 \qquad \text{(urgent location)}$$

$$\mathcal{L} \equiv @l_n \wedge \bigwedge_{l \neq l_n} \neg @l \qquad \text{(location predicates)}$$

Let further $\boldsymbol{\Phi} \equiv \Pi[\mathbf{c}_{n+1}/\mathbf{c}]$ where $\Pi[\mathbf{c}_{n+1}/\mathbf{c}]$ is obtained from Π by substituting every occurrence of a clock $c \in C$ by c_{n+1}. For instance, for the property Π used in Example 1 we have $\boldsymbol{\Phi} = \neg @client.serReceiving \vee x_5 < 4$. Then, the Π-extended TDTCS associated with Y is defined as $\mathcal{T}^{\Pi} = \mathcal{T} \wedge \neg \boldsymbol{\Phi}$.

A satisfying assignment of a TDTCS $\mathcal{T}$ is a variable assignment to every clock c_j and delay variable δ_j in $\mathcal{T}$ that satisfies every constraint in $\mathcal{T}$. A satisfying assignment of $\mathcal{T}$ induces a realization $\delta_0 ... \delta_n$ of Y. $\mathcal{T}$ is a correct formalization of Y when any assignment of $\mathcal{T}$ contains a realization of Y and any realization r in Y is part of an assignment of $\mathcal{T}$.

Theorem 1 *When a TDTCS $\mathcal{T}$ is created for an STT Y by Definition 8, then $\delta_0^c, \ldots, \delta_n^c$ is a realization of Y iff there exists a satisfying variable assignment ι for $\mathcal{T}$ such that for all $j \in [0, n], \iota(\delta_j) = \delta_j^c$.*

Proof Assume an STT $Y = \theta_0, ..., \theta_{n-1}$ of a TA, a TDTCS $\mathcal{T}$ of Y and an arbitrary delay sequence $\delta = \delta_0, \ldots, \delta_n$. We need to prove that δ either satisfies Y and $\mathcal{T}$ or none of both. We show by induction over the delays that after each delay δ_j the clocks values are equivalent in Y and $\mathcal{T}$, and satisfy the same clock constraints.

Base Case: In the initial state, no delay has yet occurred. The value of every clock in Y is 0, which is also ensured in $\mathcal{T}$ by $\mathcal{C}_0$.

Induction Step: It holds that the clock values in Y and $\mathcal{T}$ are equivalent for $\delta_0 ... \delta_{j-1}$. We assume that after a time delay δ_j a transition θ_j transits from a location l_j to a location l_{j+1}. A constraint in Y or $\mathcal{T}$ is not satisfied for δ_j in one of the following cases:

- When δ_j is negative, it violates the semantics of Y but also violates the constraints in $\mathcal{A}$ of $\mathcal{T}$.
- After δ_j, a guard $c_i \sim \beta$ of a clock c_i is not enabled. The clock value of c_i was equivalent by assumption in Y and $\mathcal{T}$ before δ_j, and so is equivalent after δ_j. Since the guard

is by construction of $\mathcal{T}$ encoded in Y and $\mathcal{T}$ and the clock value of c_i is equivalent, the guard is not enabled in Y and $\mathcal{T}$.
- After δ_j, a clock c_i does not satisfy a location invariant $c_i \sim \beta$ in l_j. Since the invariant is by construction encoded in Y and $\mathcal{T}$ and the clock value of c_i is equivalent as shown above, $c_i + \delta_j$ is violated in Y and $\mathcal{T}$.
- A clock c_i does not satisfy a location invariant $c_i \sim \beta$ in l_{j+1}. By construction of $\mathcal{T}$, the invariant is encoded in Y and $\mathcal{T}$. The clock value of c_i is either $c_i + \delta_j$ when c_i is reset or 0. By the construction of $\mathcal{T}$, c_i is reset in $\mathcal{T}$ exactly when it is reset by θ_j in Y. Thus, $c_i \sim \beta$ in l_{j+1} is not satisfied in Y and $\mathcal{T}$.
- δ_j is not 0 even l_j is an urgent location. By construction of $\mathcal{T}$, l_j is in $\mathcal{U}$ and so only $\delta_j = 0$ is an assignment of $\mathcal{T}$. Thus, Y and $\mathcal{T}$ are not satisfied.
- With any other value of δ_j, the time constraints in Y and $\mathcal{T}$ are satisfied.

As we see, the clock values in Y and $\mathcal{T}$ are equivalent after each delay, and also equivalently satisfy the clock constraints in Y and $\mathcal{T}$. In consequence, δ satisfies Y and $\mathcal{T}$, or none of both. □

Theorem 1 ensures that an assignment of $\mathcal{T}$ and the contained realization of Y satisfy the same time constraints and location predicates. Consequently, they also equivalently violate or satisfy the time constraints and location predicates of which an invariant property Π exists. This allows us to infer Corollary 1 because the conjunction of $\mathcal{T}$ and Π is a Π -extended TDTCS $\mathcal{T}^{\Pi}$. A modification of $\mathcal{T}^{\Pi}$ that turns the formula unsatisfiable is also a potential repair in the underlying TA since it prevents the property violation. We will use this insight in the next section to compute a repair.

Corollary 1 *An STT Y witnesses a violation of Π in a TA iff $\mathcal{T}^{\Pi}$ is satisfiable.*

Proof We know by Theorem 1 that a realization r of an STT exists iff a satisfying assignment ι of the STT exists. r and ι satisfy equivalent location predicates and clock constraints and, thus, r and ι equivalently hold or violate Π. □

4 Repair computation

We propose a repair technique that analyzes the responsibility of syntactic elements occurring in a single TDT for causing the violation of a specification Π. We first present the formalization of different syntactic modifications of a TDT that can potentially remedy causes for property violations and then present the algorithm to compute a set of such syntactic modifications that are possible repairs. The question of admissibility of the computed repairs will be addressed in Sect. 5. Throughout this section, we assume that Y is a TDT for T and Π.

4.1 Formal modification of syntactic elements

We introduce *variation variables* v that represent correction values of syntactic elements of the TA for which Y is produced and which characterize the proposed repair. For instance, to modify clock bounds for a transition in a TA, a fresh variation variable is added to every clock bound occurring in location invariants and transition guards involved in this transition. The values of the variation variables are computed so that

none of the realizations of Y in the modified automaton leads to a violation of Π. This is done by defining a new constraint system that captures the conditions on the variable v under which the violation of Π will not occur in the corresponding trace of the modified automaton. Using this constraint system, we then define a MaxSMT problem so that its solution minimizes the number of changes to T that are needed to achieve the repair.

4.1.1 Clock bound variation

Recall that the clock bounds occurring in location invariants and in transition guards are represented by the $\mathcal{I}$ and $\mathcal{G}$ sets defined for the TDT Y. We then introduce bound variation variables v_i^{bv} describing the possible static variation in the TA code for a clock bound β with an index i in the set ibounds ∪ gbounds, and modify the TDT constraint system accordingly. A variation of the bounds only affects the location invariant constraints $\mathcal{I}$ and the transition guard constraints $\mathcal{G}$. We thus define an appropriate invariant variation constraint $\mathcal{I}^{bv}$ and guard variation constraint $\mathcal{G}^{bv}$ that capture the clock bound modifications:

$$\mathcal{I}^{bv} \equiv \bigwedge_{(c\sim\beta)\in ibounds} c \sim (\beta + v_i^{bv}) \wedge c + \delta_j \sim (\beta + v_i^{bv})$$

$$\mathcal{G}^{bv} \equiv \bigwedge_{(c\sim\beta)\in gbounds} c + \delta_j \sim (\beta + v_i^{bv})$$

We also need constraints ensuring that the modified clock bounds remain positive:

$$\mathcal{Z}^{bv} \equiv \bigwedge_{(c\sim\beta)\in ibounds\ \cup\ gbounds} \beta + v_i^{bv} \geq 0$$

Putting all of this together, we obtain the *bound variation TDT constraint system*

$$\mathcal{T}^{bv} \equiv \mathcal{C}_0 \wedge \mathcal{A} \wedge \mathcal{R} \wedge \mathcal{D} \wedge \mathcal{I}^{bv} \wedge \mathcal{G}^{bv} \wedge \mathcal{Z}^{bv} \wedge \mathcal{U} \wedge \mathcal{L}$$

which captures all realizations of Y in TAs T^{bv} that are obtained from T by modifying the clock bounds by some syntactically consistent variations v_i^{bv}.

As an example, consider the bound variation for the guard $y \geq 1$ of transition Θ_3 in Example 1. The modified guard constraint, a conjunct in $\mathcal{G}^{bv}$, is $y_3 + \delta_3 \geq 1 + v_i^{bv}$. The corresponding non-negativity constraint in $\mathcal{Z}^{bv}$ is $1 + v_i^{bv} \geq 0$.

4.1.2 Operator variation

This type of variation is motivated by the assumption that a wrong comparison operator in a location invariant or transition guard may cause a property violation. We assume for the repair encoding that the operators $\sim$ are indexed according to their order in the sequence $\langle <, \leq, =, \geq, > \rangle$. The possible repairs are encoded by a fresh variation variable v_i^{ov} where the value of v_i^{ov} is the index of the corresponding comparison operator. For instance, the use of the $<$ comparison operator as a repair is indicated by setting $v_i^{ov} = 1$.

We define operator variation constraints $\mathcal{I}^{ov}$ and $\mathcal{G}^{ov}$ with the help of an n-ary exclusive or operation $\bigoplus_{i=0\ldots n} f_i$ which is satisfied iff exactly one of the formulas f_i is true:

$$\mathcal{I}^{ov} \equiv \bigwedge_{(c\sim\beta)\in ibounds} \bigoplus_{0\le k\le 5} c \sim_k \beta \wedge c + \delta_j \sim_k \beta \wedge v_i^{ov} = k.$$

$$\mathcal{G}^{ov} \equiv \bigwedge_{(c\sim\beta)\in gbounds} \bigoplus_{0\le k\le 5} c + \delta_j \sim_k \beta \wedge v_i^{ov} = k.$$

Now, we construct an operator variation TDT constraint system $\mathcal{T}^{ov} \equiv \mathcal{C}_0 \wedge \mathcal{A} \wedge \mathcal{R} \wedge \mathcal{D} \wedge \mathcal{I}^{ov} \wedge \mathcal{G}^{ov} \wedge \mathcal{U} \wedge \mathcal{L}$.

As an example for the operator variation encoding, consider the guard $w >= 1$ of transition θ_2 in Fig. 1b. The $\mathcal{G}^{ov}$ contains the constraint $w_2 >= 1 \wedge (v_i^{ov} = 0) \bigoplus ... \bigoplus w_2 > 1 \wedge (v_i^{ov} = 5)$.

4.1.3 Clock reference variation

This variation aims to repair property violations resulting from errors that can be traced back to the unintended use of a wrong clock variable. We uniquely identify every clock in the repair encoding by an index k. We introduce a fresh variation variable v_i^{cv} for every constraint with a clock c. Its assigned value k indicates that in the constraint, the clock c_k is to be used instead of c. For example, if $y \le 2$ is a repaired constraint, where clock y has index $k = 1$, then $v_i^{cv} = 1$.

We define the appropriate clock variation constraints $\mathcal{I}^{cv}$ and $\mathcal{G}^{cv}$:

$$\mathcal{I}^{cv} \equiv \bigwedge_{(c\sim\beta)\in ibounds} \bigoplus_{0\le k\le |C|} (c_k \sim \beta) \wedge (c_k + \delta_j \sim \beta) \wedge (v_i^{cv} = k)$$

$$\mathcal{G}^{cv} \equiv \bigwedge_{(c\sim\beta)\in gbounds} \bigoplus_{0\le k\le |C|} (c_k + \delta_j \sim \beta) \wedge (v_i^{cv} = k)$$

From this we obtain the clock reference variation TDT constraint system $\mathcal{T}^{cv} \equiv \mathcal{C}_0 \wedge \mathcal{A} \wedge \mathcal{R} \wedge \mathcal{D} \wedge \mathcal{I}^{cv} \wedge \mathcal{G}^{cv} \wedge \mathcal{U} \wedge \mathcal{L}$.

An example for the clock reference repair encoding is given by the guard $y \ge 1$ of transition θ_3 in Fig. 1b. $\mathcal{G}^{cv}$ contains the constraint $(y_3 + \delta_3 \ge 1) \wedge (v_i^{cv} = 0) \bigoplus ... \bigoplus (z_3 + \delta_3 \ge 1) \wedge (v_i^{cv} = 4)$.

4.1.4 Reset clock variation

This variation aims to repair a property violation by adding or removing clock resets. We introduce a variation variable $v_{c,\theta}^{rv}$ for the transition θ leaving location l_j and every clock c in the TDT. The reset status in the extended constraint system is inverted when $v_{c,\theta}^{rv} \neq 0$: if c was not reset before, it will now be reset, and vice versa. This is encoded by the clock reset variation constraints $\mathcal{R}^{rv}$ and $\mathcal{D}^{rv}$:

$$\mathcal{R}^{rv} \equiv \bigwedge_{c\in reset(\theta_j)} c_{j+1} = \begin{cases} 0, & \text{if } v_{c,\theta}^{rv} = 0 \\ c + \delta_j, & \text{otherwise} \end{cases}.$$

$$\mathcal{D}^{rv} \equiv \bigwedge_{c\notin reset(\theta_j)} c_{j+1} = \begin{cases} c + \delta_j, & \text{if } v_{c,\theta}^{rv} = 0 \\ 0, & \text{otherwise} \end{cases}.$$

As a result we obtain the reset clock variation TDT constraint system $\mathcal{T}^{v} \equiv \mathcal{C}_0 \wedge \mathcal{A} \wedge \mathcal{R}^{rv} \wedge \mathcal{D}^{rv} \wedge \mathcal{I} \wedge \mathcal{G} \wedge \mathcal{U} \wedge \mathcal{L}$.

To illustrate the clock reset repair encoding, consider the clock reset $y := 0$ on transition θ_2 in Fig. 1b. $\mathcal{R}^{rv}$ contains the constraint $((y_3 = 0) \wedge (v^{rv}_{y,\theta_2} = 0)) \vee ((y_3 = y_2 + \delta_2) \wedge (v^{rv}_{y,\theta_2} \neq 0))$.

4.1.5 Urgent location variation

Here we aim to repair cases where a faulty usage of urgent locations causes a property violation. Urgency of a location is modeled in the TDT constraint system by setting the location delay δ_j to 0. We define a fresh variation variable v^{uv}_i for a location l_j. For $v^{uv}_i \neq 0$, the urgency for a location l_j is inverted. We encode this idea using the following urgency variation constraint $\mathcal{U}^{uv}$:

$$\mathcal{U}^{uv} \equiv \bigwedge_{l_j \in \text{urgent}} (v^{uv}_i = 0 \implies \delta_j = 0) \ \wedge \bigwedge_{l_j \notin \text{urgent}} (v^{uv}_i \neq 0 \implies \delta_j = 0).$$

We construct the urgent location variation TDT constraint system $\mathcal{T}^{uv} \equiv \mathcal{C}_0 \wedge \mathcal{A} \wedge \mathcal{R} \wedge \mathcal{D} \wedge \mathcal{I} \wedge \mathcal{G} \wedge \mathcal{U}^{uv} \wedge \mathcal{L}$.

As an example for the urgent location repair encoding, consider the location `serReceiving` reached by θ_3 in Fig. 1a. $\mathcal{U}^{uv}$ contains the constraint $(v^{uv}_i \neq 0) \rightarrow (\delta_4 = 0)$.

4.2 Repair by variation analysis

The objective of the variation analysis is to provide hints to the system designer regarding which minimal syntactic changes to the considered model might prevent the violation of property $\boldsymbol{\Pi}$. Minimality here is considered with respect to the number of syntactic modifications that need to be performed in the considered TA model.

For the analysis, we first choose one of the variation TDTCS described before and denote it with $\mathcal{T}^*$. We implement an analysis by using $\mathcal{T}^*$ to derive an instance of the partial MaxSMT problem whose solutions yield candidate repairs for the timed automaton T. The partial MaxSMT problem takes as input a finite set of assertion formulas belonging to a fixed first-order theory. These assertions are partitioned into *hard* and *soft* assertions. The hard assertions $\mathcal{F}^*_H$ are assumed to hold and the goal is to find a maximal subset $\mathcal{F}' \subseteq \mathcal{F}^*_S$ of the soft assertions such that $\mathcal{F}' \cup \mathcal{F}^*_H$ is satisfiable in the given theory.

For the purpose of our analysis, the hard assertions are formed by the conjunction

$$\mathcal{F}^*_H \equiv (\exists \delta_j, c_j.\, \mathcal{T}^*) \wedge (\forall \delta_j, c_j.\, \mathcal{T}^* \Rightarrow \boldsymbol{\Phi}).$$

Note that the free variables of $\mathcal{F}^*_H$ are exactly the variation variables v^*_i. Given a satisfying assignment ι for $\mathcal{F}^*_H$, let T_ι be the timed automaton obtained from T by modifying a clock bound according to the variation value $\iota(v^*_i)$ and let Y_ι be the TDT corresponding to Y in T_ι. Then $\mathcal{F}^*_H$ guarantees that

1. Y_ι is feasible, and
2. Y_ι has no realization that witnesses a violation of $\boldsymbol{\Pi}$ in T_ι.

We refer to such an assignment ι as a *local clock repair* for T and Y. To obtain a minimal local clock repair, we use the soft assertions given by the conjunction

$$\mathcal{F}_S^* \equiv \bigwedge_{v_i^*} v_i^* = 0.$$

Clearly, $\mathcal{F}_H^* \wedge \mathcal{F}_S^*$ is unsatisfiable because $\mathcal{T}^* \wedge \mathcal{F}_S^*$ is equisatisfiable with $\mathcal{T}$, and $\mathcal{T} \wedge \neg\Phi$ is satisfiable by assumption. However, if there exists at least one local clock repair for T and Y, then $\mathcal{F}_H^*$ alone is satisfiable. In this case, the MaxSMT instance $\mathcal{F}_H^* \cup \mathcal{F}_S^*$ has at least one solution. Every satisfying assignment of such a solution corresponds to a local repair that minimizes the number of syntactic modifications that need to be performed on T, and hence on the considered TA. Note that hard and soft assertions remain within a decidable logic. Using an SMT solver such as Z3, we enumerate all the optimal solutions for the partial MaxSMT instance and obtain a minimal local clock bound repair from each of them.

Example 2 We have applied the bound variation repair analysis to the TDT from Example 1. The following repairs exist:

1. $v_{z,l_5}^{bv} = -1$. This corresponds to a variation of the location invariant regarding clock `z` in location 5 of the TDT, corresponding to location `client.serReceiving`, to read $z \leq 1$ instead of $z \leq 2$. This indicates that the violation of the bound on the total duration of the transaction, as indicated by a return to the `serReceiving` location and a value greater than 4 for clock `x`, can be avoided by ensuring that the time taken for transmitting the `ser` message to the `client` is constrained to take exactly 1 time unit.
2. A further computed repair is $v_{w,l_2}^{bv} = -1$. Interpreting this variation in the context of Example 1 means that location `db.reqReceived` will be left when the clock `w` has value 1. In other words, the transmission of the message `req` to the `db` takes exactly one time unit, not between 1 and 2 time units as in the unrepaired model.
3. Another computed repair is $v_{y,l_3}^{bv} = -1$ and $v_{y,\theta_3}^{bv} = -1$, which reduce the time delay in location `db.reqProcessing` by 1 time unit.
4. No further minimal repair exists.

Examples of repairs according to the other analyses for the TDT in Fig. 1c include the following. The operator repair analysis returns, among others, a repair $v_{w,\theta_2}^{ov} = 1$, which suggests to exchange the operator in the transition guard $w \geq 1$ by $<$ and to reduce the time for receiving the message `req` to be less than one time unit. The reference clock analysis returns a repair $v_{z,l_4}^{cv} = y$. The repair exchanges the clock z in the clock constraint $z \leq 2$ by y, which implies that the model has to receive the message `ser` in less than one time unit since the guard $y \geq 1$ is satisfied when entering location `serReceiving`. The reset clock analysis returns a repair $v_{x,\theta_2}^{rv} = \mathit{true}$, which indicates to reset clock x during the transition leading into location `reqProcessing`. The urgent location analysis returns a repair $v_{\mathit{serReceiving}}^{uv} = \mathit{true}$ which suggests to turn `serReceiving` into an urgent location. This repair reduces the time for receiving message `ser` to zero.

4.2.1 Complexity considerations

The tool Z3 [46] computes a repair for partial MaxSMT instances without quantifiers. The repair analysis first executes quantifier elimination on $\mathcal{F}_H^*$, which has a double exponential worst-case complexity in the number of quantifier variables [15]. The number of quantified variables in $\mathcal{F}_H^*$ rises linearly with the length of the TDT. Afterwards, every repair analysis

solves a MaxSMT problem for quantifier-free linear real arithmetic constraints. The complexity of the repair computation is determined by the number of time constraints in the TDT which increases with the length of the TDT. The complexity of quantifier-free linear real arithmetic is polynomial [30, 33]. The MaxSMT problem can be reduced in polynomial time to a MaxSAT problem, and a MaxSAT problem is an optimization problem in the complexity class NP [34]. We conclude that the quantifier elimination is the most complex computation in the analysis, which means that the overall complexity is in the worst case double exponential in the length of the TDT.

5 Admissibility of repair

The synthesized repairs that lead to a TA T_ι change the original TA T in fundamental ways, both syntactically and semantically. This brings up the question whether the synthesized repairs are admissible. In fact, one of the key questions is what notion of admissibility is meaningful in this context.

5.1 Admissibility criteria

From a *syntactic* point of view, the repair obtained from a satisfying assignment ι of the MaxSMT instance ensures that T_ι is a syntactically valid TA model, for instance, by placing non-negativity constraints on repaired clock bounds. In case repairs alter right hand sides of clock constraints to rational numbers, this can easily be transformed to integers by normalizing all clock constraints in the TA.

From a *semantic* perspective, the impact of the repairs is more profound. Since the repairs affect time bounds in location invariants and transition guards, as well as clock resets, the behavior of T_ι may be fundamentally different from the behavior of T.

- The computed repair for one property Π may render another property Π' violated. To check admissibility of the synthesized repair with respect to the set of all properties $\widehat{\Pi}$ in the system specification, a full re-checking of $\widehat{\Pi}$ is necessary.
- A repair may have introduced zenoness and timelock [8] into T_ι. As discussed in [8], there exist both an over-approximating static test for zenoness as well as a model checking based precise test for timelocks that can be used to verify whether the repair is admissible in this regard.
- Due to changes in the possible assignment of time values to clocks, reachable locations in the TA T may become unreachable in T_ι, and vice versa. On the one hand, this means that some functionalities of the system may no longer be provided since part of the actions in T will no longer be executable in T_ι, and vice versa. Further, a reduction in the set of reachable locations in T_ι compared to T may mean that certain locations with property violations in T are no longer reachable in T_ι, which implies that certain property violations are masked by a repair instead of being fixed. On the other hand, the repair leading to locations becoming reachable in T_ι that were unreachable in T may have the effect that previously unobserved property violations become visible and that T_ι possesses functionality that T does not have, which may or may not be desirable.

It should be pointed out that we assess admissibility of a repair leading to T_ι with respect to a given TA model T, and not with respect to a correct TA model T^* satisfying Π. Before we define an admissibility test based on functional equivalence, we introduce some necessary foundation.

5.2 Functional equivalence

While various variants of semantic admissibility may be considered, we are focusing on a notion of admissibility that ensures that a repair does not unduly change the functional behavior of the modeled system while adhering to the timing constraints of the repaired system. We refer to this as *functional equivalence*. The functional capabilities of a timed system manifest themselves in the sets of action or transition traces that the system can execute. For TAs T and T_ι this means that we need to consider the languages over the action or transition alphabets that these TAs define. Considering the timed languages of T and T_ι, we can state that $\mathcal{L}_T(T) \neq \mathcal{L}_T(T_\iota)$ since the repair forces at least one timed trace to be purged from $\mathcal{L}_T(T)$. This means that equivalence of the timed languages cannot be an admissibility criterion ensuring functional equivalence.

At the other end of the spectrum we may relate the de-timed languages of T and T_ι. The *de-time* operator $\alpha(T)$ is defined such that it omits all timing constraints and resets from any TA T. Requiring $\mathcal{L}(\alpha(T)) = \mathcal{L}(\alpha(T_\iota))$ is tempting since it states that when eliminating all timing related features from T and from the repaired T_ι, the resulting action languages will be identical. However, this admissibility criterion would be flawed, since the repair in T_ι may imply that unreachable locations in T will be reachable in T_ι, and vice versa. This may have an impact on the untimed languages, and even though $\mathcal{L}(\alpha(T)) = \mathcal{L}(\alpha(T_\iota))$, it may be that $\mathcal{L}_\mu(T) \neq \mathcal{L}_\mu(T_\iota)$. To illustrate this point, consider the running example in Fig. 1a and assume the invariant in location `client.serReceiving` to be modified from $z \leq 2$ to $z \leq 1$ in the repaired TA T_ι. Applying the de-time operator to T_ι implies that the location `client.timeout`, which is unreachable in T_ι, becomes reachable in the de-timed model. Since `client.timeout` is reachable in T, the TAs T and T_ι are not functionally equivalent, even though their de-timed languages are identical. Notice that for the untimed languages it holds that $\mathcal{L}_\mu(T) \neq \mathcal{L}_\mu(T_\iota)$ since no timed trace in $\mathcal{L}_T(T_\iota)$ reaches location `timeout`, even though such a timed trace exists in $\mathcal{L}_T(T)$. In particular, $\mathcal{L}_\mu(T)$ contains the untimed trace $\Theta_0\Theta_1\Theta_2\Theta_3\Theta_4$, where Θ_4 is the transition towards the location `client.timeout`, which is missing in $\mathcal{L}_\mu(T_\iota)$. As a consequence, we resort to considering the untimed languages of T and T_ι when assessing functional equivalence and require $\mathcal{L}_\mu(T) = \mathcal{L}_\mu(T_\iota)$. It is easy to see that $\mathcal{L}_\mu(T) = \mathcal{L}_\mu(T_\iota) \Rightarrow \mathcal{L}(\alpha(T)) = \mathcal{L}(\alpha(T_\iota))$. In conclusion, the equivalence of the untimed languages ensures functional equivalence.

5.3 Admissibility test

Designing an algorithmic admissibility test for functional equivalence is challenging due to the computational complexity of determining the equivalence of the untimed languages $\mathcal{L}_\mu(T)$ and $\mathcal{L}_\mu(T_\iota)$. While language equivalence is decidable for languages defined by NBAs, it is undecidable for timed languages [4]. For untimed languages, however, this problem is again decidable [4]. The algorithmic implementation of the test for functional equivalence that we propose proceeds in two steps.

- First, the untimed languages $\mathcal{L}_\mu(T)$ and $\mathcal{L}_\mu(T_\iota)$ are constructed. This requires an untime transformation of T and T_ι yielding NBAs representing $\mathcal{L}_\mu(T)$ and $\mathcal{L}_\mu(T_\iota)$. While the standard untime transformation for TAs [4] relies on a region construction, we propose a transformation that relies on a zone construction [23]. This will provide a more succinct representation of the resulting untimed languages and, hence, a more efficient equivalence test.
- Second, it needs to be determined whether $\mathcal{L}_\mu(T) = \mathcal{L}_\mu(T_\iota)$. As we shall see, the obtained NBAs are closed. Hence, we can reduce the equivalence problem for these ω -regular languages to checking equivalence of the regular languages obtained by taking the finite prefixes of the traces in $\mathcal{L}_\mu(T)$ and $\mathcal{L}_\mu(T_\iota)$. This allows us to interpret the NBAs obtained in the first step as NFAs, for which the language equivalence check is a standard construction [24].

5.4 Automata for untimed languages

The construction of an automaton representing an untimed language, here referred to as an *untime construction*, has so far been proposed based on a region abstraction [4]. The region abstraction is known to be relatively inefficient since the number of regions is, among other things, exponential in the number of clocks [8]. We therefore propose an untime construction based on the construction of a zone automaton [23], which in the worst case is of the same complexity as the region automaton, but is more succinct in practice [12].

Definition 9 (Untimed Nondeterministic Büchi Automaton) Assume a TA T and the corresponding zone automaton $[\![T]\!]_Z = (S_Z, s_Z^0, \Sigma_Z, \Theta_Z)$. We define the *untimed Nondeterministic Büchi automaton* as the closed NBA $B_T = (S, \Sigma, \rightarrow, S_0, F)$ obtained from $[\![T]\!]_Z$ such that $S = S_Z$, $\Sigma = \Sigma_Z \setminus \{\delta\}$, $S_0 = \{s_Z^0\}$ and $F = S$. For every transition in Θ_Z with a label $a \in \Sigma$, we add a transition to $\rightarrow$ created by the rule $\frac{(l,z) \overset{\delta}{\rightsquigarrow} (l,z^\uparrow) \overset{a}{\rightsquigarrow} (l',z')}{(l,z) \overset{a}{\longrightarrow} (l',z')}$ with $z^\uparrow = \{v + \delta \in I(l) \mid v \in z, \delta \in \mathbb{R}_{\geq 0}\}$. In addition, we add self-transitions $(l,z) \overset{\tau}{\longrightarrow} (l,z)$ to every state $(l,z) \in S_B$.

The following observations justify this definition:

- A timed trace of T may remain forever in the same location after a finite number of action transitions. In order to enable B_T to accept this trace, we add a self-transition labeled with τ to $\rightarrow$ for each state $s \in S$ in B_T, and later define s as accepting. These τ -self-transitions extend every finite timed trace t leading to a state in S_τ to an infinite trace $t.\tau^\omega$.
- The construction of the acceptance set F is more intricate. Convergent traces are often excluded from consideration in real-time model checking [8]. As a consequence, in the untime construction proposed in [4], only a subset of the states in S may be included in F. A repair may render a subgraph of the location graph of T that is only reachable by divergent traces, into a subgraph in T_ι that is only reachable by convergent traces. However, excluding convergent traces is only meaningful when considering unbounded liveness properties, but not when analyzing timed safety properties, which in effect are safety properties. As argued in [12], unbounded liveness properties appear to be less important than timed safety properties in timed systems.

This is due to the observation that divergent traces reflect unrealistic behavior in the limit, but finite prefixes of infinite divergent traces, which only need to be considered for timed safety properties, correspond to realistic behavior. This observation is also reflected in the way in which, e.g., UPPAAL treats reachability by convergent traces. In conclusion, this argument justifies our choice to define the zone automaton in the untime construction as a closed BA, i.e., $F = S$.

We will now prove Theorem 2 and Theorem 3 to show that our admissibility test is correct. Theorem 2 states that the zone based untimed NBA construction actually preserves the untimed languages. In particular, we show that for a given TA T, $\mathcal{L}(B_T) = \mathcal{L}_\mu(T)$. For the proof, we use *Untimed Bisimulation* in order to replace the concrete time delay in T by the empty word ϵ, and to establish the desired language equality.

Definition 10 (Untimed Bisimulation (adapted from [12])) For a TTS $H = (S, s_0, \Sigma, \rightarrow)$, let $\rightarrow_\epsilon$ be the relation obtained from $\rightarrow$ by replacing all delay transitions (s_1, δ, s_2) by (s_1, ϵ, s_2). Then, a TTS $H = (S, s_0, \Sigma, \rightarrow)$ is said to *untimed simulate* another TTS $H' = (S', s'_0, \Sigma, \rightarrow')$ if there exists a relation $R \subseteq S \times S'$ such that (1) $(s_0, s'_0) \in R$ and (2) for all $(s_1, s'_1) \in R$, $s_2 \in S$, and $a \in \Sigma \cup \{\epsilon\}$ it holds that if $s_1 \xrightarrow{a} s_2 \in \rightarrow_\epsilon$, then there exists s'_2 with $s'_1 \xrightarrow{a} s'_2 \in \rightarrow'_\epsilon$ and $(s_2, s'_2) \in R$. H and H' are *untimed bisimilar* if in addition H' untimed simulates H.

The proof of Theorem 2 is based on the insight from Lemma 1 that two untimed bisimilar TTSs have equivalent untimed languages.

Lemma 1 *Given two TTS H_1 and H_2, if H_1 untimed simulates H_2, then $\mathcal{L}_\mu(H_1) \subseteq \mathcal{L}_\mu(H_2)$.*

Proof Assume that TTS $H_1 = (S, s_0, \Sigma, \rightarrow)$ untimed simulates a TTS $H_2 = (S', s'_0, \Sigma, \rightarrow')$ with a simulation relation R. For any word v in $\mathcal{L}_\mu(H_1)$, we know that a timed trace $\xi = (t_1, a_1) \ldots$ exists in $\mathcal{L}(H_1)$. We show that v is also in $\mathcal{L}_\mu(H_2)$ by inductively constructing a timed trace $\xi' = (t'_1, a'_1) \ldots$ in $\mathcal{L}(H_2)$ with $\mu(\xi) = \mu(\xi')$. The induction iterates over the timed actions (t_i, a_i) in ξ.

Base Case: Initially, no timed action is taken in ξ. The relation (s_0, s'_0) is in R. For the induction step, we assume helper variables $t_0 = 0$ and $t'_0 = 0$.

Induction Step: For a timed action (t_{i+1}, a_{i+1}) in ξ and (s_i, s'_i) in R, we now construct a timed action (t'_{i+1}, a'_{i+1}) in ξ'.

For (t_{i+1}, a_{i+1}), a state s_* exists in H_1 such that a delay transition (s_i, t, s_*) with $t = t_{i+1} - t_i$ and an action transition (s_*, a_{i+1}, s_{i+1}) are in $\rightarrow$. The untimed bisimulation replaces delay transitions by ϵ, and defines $\rightarrow_\epsilon$ and $\rightarrow'_\epsilon$. Since (s_i, s'_i) is in R and $(s_i, \epsilon, s_*) \in \rightarrow_\epsilon$, a state s'_* has to exist with $(s'_i, \epsilon, s'_*) \in \rightarrow'_\epsilon$ and $(s_*, s'_*) \in R$. (s'_i, ϵ, s'_*) exists only since a delay transition (s'_i, t', s'_*) exists in $\rightarrow'$. We construct $t'_{i+1} = t'_i + t'$. Because (s_*, a_{i+1}, s_{i+1}) is in $\rightarrow_\epsilon$ and $(s_*, s'_*) \in R$, a state s'_{i+1} in H_2 has to exist such that $(s_{i+1}, s'_{i+1}) \in R$ and $(s'_*, a_{i+1}, s'_{i+1}) \in \rightarrow'_\epsilon$. We conclude $a'_{i+1} = a_{i+1}$. Hence, we constructed the timed action (t'_{i+1}, a'_{i+1}) of ξ' and $(s_{i+1}, s'_{i+1}) \in R$ holds.

In conclusion, we can construct for every timed word ξ in $\mathcal{L}(H_1)$, a timed word ξ' in $\mathcal{L}(H_2)$, such that $\mu(\xi) = \mu(\xi')$. □

Additionally, the proof of Theorem 2 relies on the assumption that the timed language of the zone automaton $[\![T]\!]_Z$ is identical to the timed language of T, which we state in the following lemma.

Lemma 2 (Zone Language Equivalence) *Let $[\![T]\!]_Z$ be a zone automaton derived from a TA T, then $\mathcal{L}_T([\![T]\!]_Z) = \mathcal{L}_T(T)$.*

Proof The proof of this lemma can be derived from the proofs of Theorem 1 and Theorem 2 given in the full version of [45], which together prove that the reachable states in the transition system of T are also reachable in $[\![T]\!]_Z$, and vice versa. The proofs rely on an induction over the length of the traces of T and $[\![T]\!]_Z$ and imply an equivalence of the sets of traces of T and $[\![T]\!]_Z$. This implies $\mathcal{L}_T([\![T]\!]_Z) = \mathcal{L}_T(T)$. □

Theorem 2 (Correctness of Untimed NBA Construction) *For an untimed NBA B_T derived from a TA T according to Definition 9 it holds that $\mathcal{L}(B_T) = \mathcal{L}_\mu(T)$.*

Proof Assume a zone automaton $[\![T]\!]_Z = (S_Z, s_Z^0, \Sigma_Z, \Theta_Z)$ for a TA T and let $B_T = (S_B, \Sigma_B, \rightarrow, S_B^0, F)$ be the associated untimed NBA obtained according to Definition 9 with $[\![T]\!]_Z$.

Lemma 2 permits us to prove Theorem 2 by showing $\mathcal{L}(B_T) = \mathcal{L}_\mu([\![T]\!]_Z)$. We prove the conjecture by proving the stricter condition that the zone automaton $[\![T]\!]_Z$ and the untimed NBA B_T are untimed bisimilar. We may then conclude language equivalence by Lemma 1 and bisimilarity. It is appropriate to compare an untimed NBA B_T to a zone automaton $[\![T]\!]_Z$ by untimed bisimulation since B_T is an untimed automaton, in particular, an untimed zone automaton.

We now show that $R = \{((l, z_Z), (l, z_B)) | z_Z^\uparrow = z_B^\uparrow \wedge (l, z_Z) \in S_Z \wedge (l, z_B) \in S_B\}$ is an untimed bisimulation relation. R is an untimed bisimulation relation for the alphabet $\Sigma = (\Sigma_B \setminus \{\tau\}) \cup (\Sigma_Z \setminus \{\delta\})$, where the delay transitions δ and τ are removed by the construction of the untimed bisimulation.

We show for an arbitrary $((l, z_Z), (l, z_B)) \in R$ that $((l', z_Z'), (l', z_Z')) \in R$ for all actions $a \in \Sigma$. $z_B \subseteq z_Z$ holds by construction of B_T since transitions in B_T reach a state after taking an action transition. Hence, there are two cases to be considered in order to determine whether $z_Z^\uparrow = z_B^\uparrow$:

1. If $z_Z = z_B$, then the untime construction creates a transition $(l, z_B) \overset{a}{\rightsquigarrow} (l', z_Z') \in \rightarrow$ iff $(l, z_Z) \xrightarrow{\delta} (l, z_Z^\uparrow) \xrightarrow{a} (l', z_Z') \in \Theta_Z$ exists.
2. If $z_Z = z_B^\uparrow$, then the untime construction creates a transition $(l, z_B) \overset{a}{\rightsquigarrow} (l', z_Z') \in \rightarrow$ iff $(l, z_Z) \xrightarrow{a} (l', z_Z') \in \Theta_Z$ exists.

In both cases it holds that $((l', z_Z'), (l', z_Z')) \in R$. Thus, R is a bisimulation relation. The tuple $((l_0, z_0), (l_0, z_0))$ of the initial states of $[\![T]\!]_Z$ and B_T is in R, since $S_B^0 = \{s_Z^0\}$ holds by the definition of the untime construction, and trivially $z_0^\uparrow = z_0^\uparrow$. This implies that B_T and $[\![T]\!]_Z$ are untimed bisimilar and $\mathcal{L}(B_T) = \mathcal{L}_\mu([\![T]\!]_Z)$ holds. □

5.5 Equivalence check for untimed languages

Given that the zone automaton construction delivers closed NBAs we can reduce the admissibility test $\mathcal{L}_\mu(T) = \mathcal{L}_\mu(T_\iota)$ defined over infinite languages to an equivalence test over the finite prefixes of these languages, represented by interpreting the zone automata as NFAs. The following theorem justifies this reduction.

Theorem 3 (**Language Equivalence of Closed NBAs**) *Given closed NBAs B and B', if $\mathcal{L}_\mathrm{f}(B) = \mathcal{L}_\mathrm{f}(B')$ then $\mathcal{L}(B) = \mathcal{L}(B')$.*

Proof For a closed NBA B, it is easy to see that $\mathcal{L}_\mathrm{f}(B) = \mathrm{pref}(\mathcal{L}(B))$. Assume $\mathcal{L}_\mathrm{f}(B) = \mathcal{L}_\mathrm{f}(B')$ and $\mathcal{L}(B) \neq \mathcal{L}(B')$. This means that, w.l.o.g., $(\exists v)(v \in \mathcal{L}(B) \wedge v \notin \mathcal{L}(B'))$. This implies that there is no accepting run on some v in B'. Since B' is closed, the only way of not accepting v is that B' blocks when reading a finite prefix $\hat{v} = v_1, \ldots, v_n$ of v. Hence, $\hat{v} \in \mathcal{L}_\mathrm{f}(B) \wedge \hat{v} \notin \mathcal{L}_\mathrm{f}(B')$, which contradicts $\mathcal{L}_\mathrm{f}(B) = \mathcal{L}_\mathrm{f}(B')$. Therefore, $\mathcal{L}(B) = \mathcal{L}(B')$. □

5.6 Complexity of the admissibility test

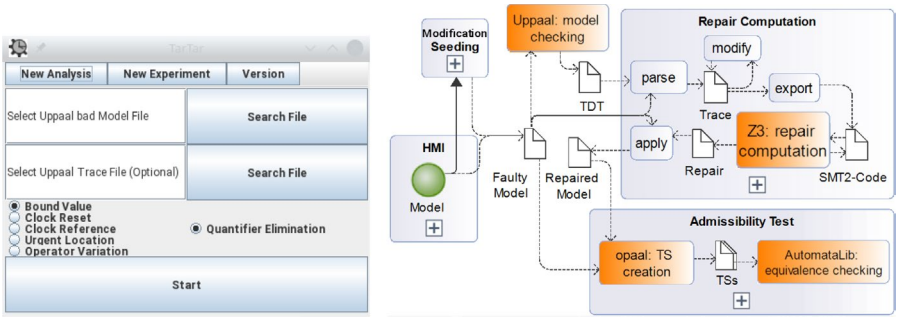
In order to test admissibility, we generate the state space of the original TA and the TA to which we apply the repair. The computation of the state space uses real-time model checking which is in PSPACE [3]. Afterwards, we interpret the two state spaces as NFAs and perform a language equivalence test. Checking language equivalence of two NFAs is decidable in NP [21]. The overall complexity of the admissibility test is thus in PSPACE.

5.7 Discussion

One may want to adapt the admissibility test so that it only considers divergent traces, e.g., in cases where only unbounded liveness properties need to be preserved by a repair. This can be accomplished as follows. First, an overapproximating non-zenoness test [8] can be applied to T and T_ι. If it shows non-zenoness, then one knows that the respective TA does not include convergent traces. If this test fails, a more expensive test needs to be developed. It requires a construction of the untimed NBA using the approach from [4], and subsequently a language equivalence test of the untimed languages accepted by the untimed NBAs using, for instance, the automata-theoretic constructions proposed in [13]. Checking language equivalence of two NBAs is in the complexity of PSPACE [14]. Creating the untimed NBA, which is the input to the equivalence check, is also in PSPACE. So the overall complexity of this analysis remains in PSPACE.

6 Tool implementation

We implemented the repair computations and the admissibility test in the tool TarTar. The software architecture of TarTar is depicted in Fig. 3b. The orange rectangles in the figure represent external tools that TarTar calls in the course of the repair analysis. Uppaal is a state-of-the-art and closed-source model checking tool, which TarTar uses to compute a TDT for a given model and property. The SMT solver Z3 [18] is used to solve the



(a) TarTar GUI (b) TarTar Architecture

Fig. 3 TarTar Tool

generated partial MaxSMT problems. In order to check the admissibility of a repair, TarTar uses opaal and the AutomataLib component of LearnLib [25] since they conveniently provide functionality used in the implementation of the admissibility test.

6.1 Data flow architecture

TarTar consists of many computation steps. For example, a TDT is parsed internally and stored as a *Trace*. This *Trace* is then modified and exported as SMT-LIB2 [7] code. We define a computation step of TarTar as the computation transforming input into result artifacts. This focus on artifacts ensures a highly cohesive architecture and clear interfaces between any two computation steps. Computation steps with identical objectives are grouped into a project. This results in four projects depicted by blue rectangles in Fig. 3b.

- *HMI* denotes the user interfaces of TarTar. The user inputs a timed model. TarTar then calls the project *Repair Computation* using a faulty timed model as a parameter. In case that the model is correct, TarTar calls the project *Modification Seeding*.
- *Modification Seeding* seeds modifications into a correct model and then repairs the resulting faulty models by computing repairs using *Repair Computation*. We use this analysis in Sect. 7 in order to benchmark the *Repair Computation* analyses.
- *Repair Computation* computes candidate repairs for a faulty timed model, applies these repairs to the model and finally automatically calls the *Admissibility Test*.
- *Admissibility Test* checks for every repaired model whether the computed repair is admissible.

6.2 Control flow architecture

TarTar computes iteratively a set of repairs for a given faulty Uppaal model and a given property Π using the following steps:

0. *Counterexample Creation*. TarTar calls Uppaal to verify the model against Π. In case Π is violated, it stores a shortest symbolic TDT witnessing the violation in XML format.

1. *Diagnostic Trace Creation*. TarTar parses the model and the TDT into a data structure *Trace*. To add potential repairs, TarTar copies the trace and replaces the constraints that will potentially be subject to a repair by their modified variants. The modified trace is then translated to a logic constraint system $\mathcal{T}^*$, represented in SMT-LIB2 code.
2. *Repair Computation*. Z3 [18] then solves a MaxSMT problem on the modified trace constraint system, computing a repair in which the number of unmodified constraints on the variation variables of the form $v^* = 0$ is maximized. Since Z3 can solve a MaxSMT problem only for quantifier-free linear real arithmetic, TarTar first runs a quantifier elimination on the constraint system $\forall \delta_j, c_j.\, \mathcal{T}^* \Rightarrow \Phi$ of $\mathcal{F}^*_H$.
3. It then solves the MaxSMT problem with soft constraints requiring $v^* = 0$ for all variation variables. In case no solution is found, TarTar terminates. Otherwise, TarTar applies the repair to the faulty model and returns a repaired model.
4. *Admissibility Test*. TarTar checks the admissibility of a repair and compares the untimed languages of the faulty and repaired models. TarTar calls the model checker opaal in order to compute the timed transition systems (TTS) of the original and the repaired Uppaal model. We modified the opaal model checker in such a way that it returns the TTS for a model. TarTar then checks whether the two TTS have equivalent untimed languages, in which case the repair is admissible. This check is implemented using the library AutomataLib. In case the two TTS are not equivalent, the admissibility test returns a trace as a witness for the difference.
5. *Iteration*. TarTar enumerates all repairs, i.e., every combination of the same kind of constraint modifications that corrects the TDT. The repairs are iteratively enumerated starting with the ones that require the smallest number of modifications to the model. After a repair is computed, the combination of modified variables that has been found is prevented from being reconsidered for future repairs by setting these modification variables to 0 using hard asserts. TarTar then proceeds with attempting to compute further, previously unconsidered repairs.

6.3 Component architecture

We implemented TarTar with the general infrastructure depicted in Fig. 4. The interface *Job* provides a general abstraction for an algorithm and specifies the necessary input and result values of the algorithm by the class *Description*. TarTar contains a *Job* for the projects *Modification Seeding*, *Repair Computations* and *Admissibility Test*. The class *Session* executes a *Job* and derivations of *Session* provide the different interfaces to the user. With this infrastructure, the analysis implementation in TarTar is independent from the

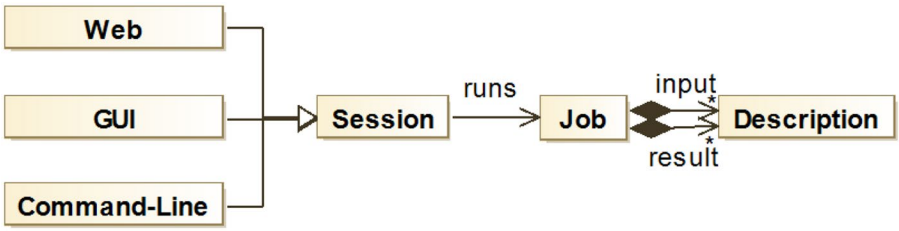


Fig. 4 TarTar component architecture

implementation of the user interfaces, thus reducing coupling and improving modifiability of the code.

6.4 Implementation details

We implemented the different projects that constitute TarTar in Java and use the build-management tool maven [37] to manage the dependencies between the projects. TarTar interacts differently with the external tools that are needed for different purposes. It calls Uppaal via the command-line interface in order to generate a TDT, calls Z3 via its API to compute a repair. For the admissibility test, it calls opaal using a command-line script and the AutomataLib as an included Java library. For the implementation of the TarTar analyses the following two details are essential.

- We modify constraints in an Uppaal model in order to apply a repair or to seed a fault. Since neither clock constraints nor transitions possess explicit unique identifiers in an Uppaal model, it is not obvious which constraint to change. We therefore uniquely identify a constraint by traversing the constraints in the sequence stored in the Uppaal model file and use the constraint index in this sequence as its identifier.
- The complexity of the algorithms for solving quantifier elimination and the MaxSMT problem rise with the number of variables in the SMT model. We therefore reduce the number of variables by exploiting implied equality constraints. For example, a variable c_j is created for every clock c in every step j of the TDT. We eliminate c_j explicitly before quantifier elimination by replacing it with the term $\sum_{i \in r \ldots j} d_i$, where d_i is the time delay at step i in the trace and r is the last step before j where c was reset.

7 Case studies and experimental evaluation

We evaluate the repair analyses by applying TarTar to the database model in Example 1 and to more complex case studies of different sizes.

7.1 Database model repairs

We applied TarTar to the database model in Fig. 1. TarTar finds two admissible clock bound repairs. A replacement of $w \leq 2$ by $w \leq 1$ and a replacement of $y \leq 1$ by $y \leq 0$ and $y \geq 1$ by $y \geq 0$ repairs the database model. With operator variation repair analysis, TarTar finds two admissible repairs that replace the operator in the clock constraint $w >= 1$ by $<$ or $<=$, respectively. With clock reference repair analysis, TarTar finds 13 admissible clock reference modification repairs, each involving two modifications. Nine repairs replace y in the constraints $y \leq 1$ and $y \geq 1$ by a selection from the set of clocks z, x and w. Four repairs replace y in the constraint $y \leq 1$ by w or x, and w in the constraint $w \geq 1$ by y or z. Applying the reset repair analysis, TarTar finds four admissible repairs. One repair removes the reset of clock y, another removes the reset of clock z, and two repairs add a reset of clock x either on the transitions towards the location *reqProcessing* or the transition towards the location *serReceiving*. Applying the urgency location repair analysis, TarTar finds only two

inadmissible and no admissible repairs, one setting the location *reqAwaiting* and the other the location *serReceiving* to urgent.

The computed repairs are reasonable and we expected them. We remain to evaluate whether TarTar can analyze more complex models.

7.2 Evaluation strategy

In order to perform an experimental evaluation of the repair analyses both qualitatively and quantitatively, we need a set of timed automata models that violate given properties. Such models are not publicly available in significant numbers and to the best of our knowledge, no benchmark suite for property violating timed automata models exists. As a consequence, the evaluation of our analyses is based on ideas taken from mutation testing [26]. Mutation testing evaluates a test set by systematically modifying the program code to be tested and computing the ratio of modifications that are detected by the test set. In adopting this strategy, we seed modifications in existing models and check whether those can be successfully repaired by our automated repair analyses. Thereby, we evaluate the quality of our repair technique.

7.2.1 Modification seeding

modifies a given TA model syntactically in one of the following ways:

- Replacing the comparison operator in a clock constraint by $\{<, \leq, =, \geq, >\}$,
- Swapping a clock name referenced in a clock constraint or invariant with some other clock name occurring in the original model,
- Modifying the set of clocks that are reset in any given transition,
- switching for any location whether it is urgent or not,
- Or modifying the bound of a single clock constraint by an amount selected from the set $\{-10, -1, +1, +0.1M, +M\}$, where M is the maximal clock bound occurring in a given model. Our observation was that making either a small modification that is close to the bound value or a modification in the order of the maximal bound value M often introduces a property violation in the considered models.

The result is a modified TA that contains a single modification. TarTar automatically checks for every modified TA whether it violates a given property, in which case a TDT is generated.

7.2.2 Experimental procedure

TarTar applies modification seeding to several models used for experimentation and generates a set of TDTs by performing model checking for a given property on each of the modified models. For every computed TDT, TarTar performs the above defined repair analyses one after another, and measures the ratio of TDTs for which the analyses compute at least one repair.

Table 1 Experimental results according to model

Model	db rep.	csma	Elevator	Viking	Bando	Pacemaker	SBR	FDDI
Source		[27]	[42]	[39]	[43]	[29]	[36]	[43]
#Location	8	10	9	24	77	104	21	16
#Transition	9	22	12	25	144	223	23	20
#Seed	110	191	88	310	1,955	1,187	353	314
#TDT	13	10	5	9	40	12	50	36
Time_{UP}	0.016	0.012	0.011	0.015	0.111	0.022	0.027	0.014
Length	4	2	1	18	279	9	84	11
#Repair	229	70	7	9	4,061	62	751	166
#Admissible	138	26	5	7	209	19	660	105
#Solved	9	8	4	5	21	10	31	34
Time_{QE}	89.346	0.049	0.049	86.539	31.555	0.663	117.057	29.859
#Timeout	2	0	0	21	46	20	86	51
$\text{Time}_{\text{Repair}}$	0.911	0.023	0.020	1.436	4.922	0.325	2.686	3.074
$\text{Mem}_{\text{Repair}}$	14.53	0.58	0.53	20.07	20.86	2.59	37.16	9.70
#Variable	30	16	6	120	1,156	116	765	116
#Constraint	91	72	28	180	8,144	988	1,211	272
Time_{Adm}	2.080	1.825	1.665	1.952	19.57	1.994	138.004	2.241
Mem_{Adm}	45	75	17	543	1,251	206	211	128

Table 2 Experimental results according to type of repair

Repair	Bound mod.	Operator var.	Clock ref.	Reset clock	Urgent loc.
#Repair	533	3,929	693	45	155
#Admissible	364	96	625	37	47
#Solved	85 (48%)	51 (29%)	35 (20%)	13 (7%)	37 (21%)
Time_{QE}	15.209	117.057	33.282	89.346	0.107
#Timeout	8	44	61	113	0
$\text{Time}_{\text{Repair}}$	4.922	2.686	3.074	0.911	0.135
$\text{Mem}_{\text{Repair}}$	20.86	37.16	14.13	14.53	3.16
#Variable	1,156	996	1,120	996	1,120
#Constraint	2,498	8,144	5,355	2,836	2,502
Time_{Adm}	138.004	59.117	116.944	2.051	58.551
Mem_{Adm}	525	543	206	45	1,251

7.3 Experiments

We have applied this evaluation strategy to eight UPPAAL models (see Table 1). Not all of the models that we considered have been published with a property that can be violated by mutating a clock constraint. For those models, we manually identify a suitable timed safety property specifying an invariant condition which can then be violated by some of the proposed mutations. In particular, we add a property to the Bando [43] model which ensures that, for as long as the sender is active, its clock never exceeds the value of 28, 116 time

Table 3 Subset of experimental results where type of repair corresponds to type of seeded modification

Seed/repair	Bound mod.	Operator var.	Clock ref.	Reset clock	Urgent loc.
#Seed	1,385	1,385	578	891	269
#TDT	60	72	27	8	8
Time_{UP}	0.111	0.083	0.093	0.027	0.021
Length	279	248	279	84	18
#Repair	314	3,508	383	6	11
#Admissible	226	68	380	6	2
#Solved	53 (88%)	38 (52%)	9 (33%)	1 (12%)	1 (12%)
Time_{QE}	15.209	31.858	25.392	43.922	0.027
#Timeout	3	8	10	7	0
$\text{Time}_{\text{Repair}}$	4.922	1.436	2.525	0.558	0.049
$\text{Mem}_{\text{Repair}}$	20.86	12.12	13.01	10.88	0.71
#Variable	1,156	996	1,120	25	90
#Constraint	2,498	8,144	5,355	57	279
Time_{Adm}	18.045	52.749	116.944	2.051	19.570
Mem_{Adm}	525	543	205	33	1,251

units. In the FDDI token ring protocol [43], the property that we use checks whether the first member of the ring never remains in any given state for more than 140 time units. The Viking model is taken from the set of test models of opaal [39]. For this model we use a property that checks whether one of the Viking processes can only enter a safe state during the first 60 time units. Note that all of these properties are satisfied by the original models prior to modification seeding.

We applied modification seeding to the models given in Table 1 and analyzed the obtained TDTs using the above described repair analyses implemented in TarTar. All analyses were performed on a computer with an i7-6700K CPU (4.00GHz), 60GB of RAM and a 64 bit Linux operating system. We summarize the results of this experiment per considered model (Table 1) and per type of considered repair (Table 2). In Table 3, we give insight into a subset of the repair analysis results for the cases where the repair analysis was of the same type as the seeded modification.

In Table 1, we give in the first three rows the source of the model as well as its number of locations and transitions. In every model, we modify the time constraints as described above, which results in a number *#Seed* of modifications and the same number of modified models. Uppaal analyzes each modified model and finds *#TDT* property-violating models. $Time_{UP}$ is the maximal time that Uppaal needs to create a TDT for the property violating models, and the longest TDT has a length of *Length*. Overall, TarTar computed a number *#Repair* of repairs for the TDTS, of which *#Admissible* are admissible. We say a TDT is solved when at least one admissible repair is computed by a repair analysis. *#Solved* states the number of solved TDTs. The computation effort for a repair analysis is given by the time $Time_{QE}$ for successful quantifier elimination, the number of timeouts *#Timeout* that occurred during quantifier elimination after 10 minutes, the average time $Time_{Repair}$ to compute a repair, and the memory consumption Mem_{Repair} for the overall analysis. The constraint system that Z3 solves has the count *#Variable* of variables and *#Constraint* of constraints. The effort for the admissibility test is given in time $Time_{Adm}$ and memory Mem_{Adm}. All times are given in seconds and memory consumption in MB. Notice that

we omit the lines pertaining to the modification seeding and TDT computation in Table 2 since they are irrelevant there.

We illustrate the evaluation procedure using an instance of the pacemaker case study, which is a real world model of realistic size. One of the seeded bound modifications increases a location invariant of this model which controls the minimal heart period to remain within a range from 400 to 1, 600 time units. The modification allows the pacemaker to delay an induced ventricular beat for too long so that this violates the property that the time between two ventricular beats of a heart is never longer than the maximal heart period of 1, 000. TarTar finds three repairs. The first repair reduces the maximal time delay between two articular heartbeats such that the pacemaker cannot and no longer needs to trigger the articular heartbeat. The second repair reduces the maximal time delay between two ventricular heartbeats such that the pacemaker cannot trigger the first ventricular heartbeat. Both repairs changes the heart behavior and removes functional behavior of the pacemaker and, hence, are classified as inadmissible. In the model context, this appears to be reasonable since the repairs restrict the environment of the pacemaker, and not the pacemaker itself. The third repair is admissible and reduces the bound modified during the seeding of bound modifications by 1199.5. The minimal heart period is then also below or equal to the maximal heart period of 1000.

Overall, TarTar seeded 4508 modifications. This resulted in 175 TDTs in total. 60 TDTs were due to bound modification, 72 were due to operator variation, 27 were due to changing the clock reference, 8 were due to complementing the reset of clocks and 8 were due to the switching of urgent locations (see Table 3). TarTar found in total 5355 repairs, out of which 1,169 were admissible. It found at least one admissible repair for 122 (69%) of the 175 TDTs. The maximal number of modified constraints in an admissible repair computed for a single TDT using all types of repair analysis was 25. A total of 4222 repairs were of the same type as the seeded modification. Out of these same type repairs, 682 were admissible. At least one admissible same type repair exists for 102 (58%) of the 175 TDTs.

7.4 Result interpretation

7.4.1 Pacemaker instance results

Our repair strategy minimizes the number of repairs but does not optimize the computed value. For instance, in the pacemaker model the computed repair of 1199.5 time units would be a correct and admissible repair even if the value was reduced by 600 time units, which would be the minimal possible repair value.

7.4.2 Modification seeding results

Few of the seeded modifications resulted in a property violation. TarTar seeded 4,508 faults, which led to 175 TDTs, thus only 3.9% of these modifications result in a TDT. This supports the hypothesis that, in practice, often times only few time constraints impact the satisfaction of a checked property.

7.4.3 Repair results where the seeded modification is of arbitrary type

TarTar computes at least one admissible repair by bound modification for 85 (48%), by operator variation for 51 (29%), by clock reference for 35 (20%), by clock reset for 13 (7%) and by urgent location for 37 (21%) of the 175 TDTs. Every analysis on its own computes less admissible repairs than the combination of all repair analyses, which solves 122 (69%) of the 175 TDTs. The largest number of constraints that an admissible repair modified was 25, which is less than anticipated. This low number of modified constraints allow us to infer that only few constraints have an impact on whether a property is violated or not.

TarTar does not find an admissible repair for every TDT. For instance, TarTar finds an admissible repair for 53 TDT of the 60 TDTs created by bound modification, but only 3 analyses result in a timeout (Table 3). Hence, there exist 4 TDTs for which no admissible repair was computed even though they did not result in a timeout. There are two reasons for this phenomenon. Notice that a TDT is caused by modification seeding in which a constraint c in a model is modified. First, the seeding of a modification in the model may enable or disable transitions and, as a consequence, change the functional behavior of the unmodified model. A repair computed by TarTar may not precisely undo the modification, and therefore it cannot be guaranteed that the repaired model is functionally consistent with the unmodified model. Second, even if the modified model is functionally consistent with the unmodified model, the computed repair may not undo the modification and at the same time be inadmissible.

7.4.4 Computational effort

A comparison of T_{QE} and T_R in every table confirms that the computational effort for the repair computation is largely determined by the quantifier elimination step, as we also concluded from the complexity analysis for the repair analysis. Only the urgent location repair consumes more computation time during the repair computation than during the quantifier elimination step. We expect that in light of the observed 226 timeouts, a more efficient quantifier elimination procedure would lead to a significantly higher number of repairs that could be computed without encountering a timeout. Furthermore, the number of timeouts, and thus the computation time needed for the repair, seems to correlate with the length of the analyzed TDT. With 86 the *SBR* model includes the largest number of timeouts and the third longest TDT with a length of 84 steps. The *bando* model has the third most timeouts (46) and the longest TDT. Obviously, the longer the TDT, the larger the resulting constraint system, leading to increased computational effort. With 1, 156 variables and 8, 144 constraints, the *bando* model yields the largest constraint system. The *SBR* model has the second largest constraint system with 765 variables and 1, 211 constraints. The model *FDDI* has a shorter TDT with a length of 11 transitions and a much smaller constraint system with 116 variables and 272 constraints. In order to provide some statistical evidence for the impact of the TDT length and the intrinsic model complexity on the computational effort we analyzed the statistical relationship between the length of the TDT and the computation time for a repair ($T_r = T_{QE} + T_R$) as well as the relationship between *#Variable* and T_r, both by estimating Kendall's tau [20] for every TDT. Kendall's tau is a measure for the ordinal association between two measured quantities. A correlation computed by Kendall's tau estimation approach is considered significant if the probability p that there is actually no correlation in a larger data set is below a certain threshold. We use the commonly applied significance threshold of 0.05. The length of a TDT is significantly related ($\tau_1 = 0.521$, $p < .001$) to T_r. Also *#Variable* is significantly related ($\tau_2 = 0.496$,

$p < .001$) to T_r. From this we conclude that the complexity of a repair depends not only on the TDT length, but also on the intrinsic complexity of the model as expressed by the number of variables occurring in the model.

We also observe that the admissibility test requires more computational resources than the repair computation. The maximum amount of memory used for the admissibility test was 1, 251MB in contrast to 37.16MB for the repair computation. This is in line with our expectation since the admissibility test involves searches of the state space of the full NTA, while the repair analyses only consider a single TDT.

Modifying states from urgent to non-urgent during modification seeding resulted in only 8 TDTs. This low number is due to the observation that the considered models contain only very few urgent states. Modifying non-urgent states to urgent ones, however, did not lead to a single property violation resulting in a TDT. The rationale is that urgency ensures to leave a state immediately without a delay which leads to a restriction rather than a relaxation regarding the time budget spent along an execution trace. As a consequence, making a state urgent does not cause a property violation in many models since the type of the checked properties is typically time bounded reachability, and a restricted time budget does not make it more likely that the property is violated.

7.4.5 Results of repairs that have the same type as the seeded modification

Recall that we seed different types of modifications that correspond to the different types of repair analyses that we propose. For every type of a repair analysis, we now compare the probability to compute an admissible repair when the seeded modificaton is of the same type as the repair (see Table 3) with the case when the seeded modification is of arbitrary type (see Table 2). Bound modification analysis repairs with a higher probability (88% to 48%) a TDT with a seeded bound modification than a modification of an arbitrary type. Also, operator modification analysis (52% to 29%), clock reference analysis (33% to 20%) and clock reset analysis (12% to 7%) compute a repair with a higher probability for the same type of seeded fault as a modification of an arbitrary type. Only the urgent location analysis computes a repair with a lower probability (12% to 21%) for a TDT with a seeded urgent location modification than for a modification of arbitrary type. In summary, with the exception of the urgent repair analysis, an admissible repair is computed with a higher probability if the repair analysis is of the same type as the seeded modification.

We finally observe that the repair computations of a seeded modification of an arbitrary type (Table 2) and repair computations of the same type as the seeded modification (Table 3) require similar computational effort in terms of time and memory consumption. We conclude that the computational effort of a repair analysis is independent of the type of a seeded modification that it repairs.

8 Conclusion

We have presented an approach to derive minimal repairs for timed reachability properties of NTA models from TDTs returned as counterexamples during model checking. The objective of this repair synthesis is to facilitate fault localization and debugging of such models during the design process. Our approach includes a

formalization of TDTs using linear real arithmetic, a repair strategy based on MaxSMT solving, the definition of an admissibility criterion and test for the computed repairs, the development of an analysis and repair tool, and the application of the proposed method to a number of case studies of realistic complexity. To the best of our knowledge, this is the first rigorous treatment of counterexamples in real-time model checking. We have nonetheless observed that our analysis computes a significant number of admissible repairs within realistic computation time bounds and memory consumption.

In future work we plan to explore the interplay between different repairs that are computed for a repaired system that still violates a property, and develop refined strategies to select promising repairs from a repair set. A further generalization of the analysis is to not only compute clock constraint modifications for faulty models but also to compute possible relaxations of clock constraints for correct models in order to support design space exploration.

Acknowledgements We wish to thank Nikolaj Bjorner and Zvonimir Pavlinovic for advice on the use of Z3. We are grateful to Sarah Stoll for helping us with the statistical evaluation of the experimental results. This work was in part supported by the National Science Foundation (NSF) under grant CCF-1350574.

Funding Open Access funding enabled and organized by Projekt DEAL. This work was in part supported by the National Science Foundation (NSF) under Grant CCF-1350574.

Availability of data and material The source code and part of the UPPAAL models are available from the github site https://github.com/sen-uni-kn/tartar. Further sources of UPPAAL models are referenced in the manuscript.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Consent to participate All authors consent to participate in the research described in this manuscript.

Consent for publication All authors and the institutions that they are affiliated with consent to the publication of the results contained in this manuscript.

Code availability See above. The tool TarTar was successful in the artefact evaluation of the CAV 2020 conference, c.f. [32].

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. André É, Arcaini P, Gargantini A, Radavelli M (2019) Repairing timed automata clock guards through abstraction and testing. In: Beyer D, Keller C (eds) *Tests and Proofs - 13th international conference, TAP@FM 2019, Porto, Portugal, October 9-11, 2019, Proceedings*, volume 11823 of *Lecture notes in computer science*, pp 129–146. Springer
2. Alur R, Bodík R, Dallal E, Fisman D, Garg P, Juniwal G, Kress-Gazit H, Madhusudan P, Martin Milo MK, Mukund R, Shambwaditya S, Seshia SA, Singh R, Solar-Lezama A, Torlak E, Udupa A (2015)

Syntax-guided synthesis. In: *Dependable software systems engineering*, volume 40 of *NATO Science for peace and security series, D: information and communication security*, pp 1–25. IOS Press
3. Alur R, Courcoubetis C, Dill DL (1993) Model-checking in dense real-time. Inf Comput 104(1):2–34
4. Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235
5. Alpern B, Schneider FB (1987) Recognizing safety and liveness. Distrib Comput 2(3):117–126
6. Bouyer P, Fahrenberg U, Larsen KG, Markey N, Ouaknine J, Worrell J (2018) Model checking real-time systems. Handbook of model checking. Springer, Berlin, pp 1001–1046
7. Barrett C, Fontaine P, Tinelli C (2017) *SMT-lib*, http://smtlib.cs.uiowa.edu/language.shtml
8. Baier C, Katoen J-P (2008) Principles of model checking. MIT Press, Cambridge
9. Ben-Abdallah H, Leue S (1997) Timing constraints in message sequence chart specifications. In: *FORTE*, volume 107 of *IFIP conference proceedings*, pp 91–106. Chapman and Hall
10. Bengtsson J, Larsen KG, Larsson F, Pettersson P, Yi W (1995) Uppaal - a tool suite for automatic verification of real-time systems. In: *Hybrid systems*, volume 1066 of *Lecture notes in computer science*, pp 232–243. Springer, Berlin
11. Bendík J, Sencan A, Gol EA, Cerná I (2021) Timed automata relaxation for reachability. In: Groote JF, Larsen KG (eds) *Tools and algorithms for the construction and analysis of systems - 27th international conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part I*, volume 12651 of *Lecture notes in computer science*, pp 291–310. Springer
12. Bengtsson J, Yi W (2003) Timed automata: semantics, algorithms and tools. In: *Lectures on concurrency and petri nets*, volume 3098 of *Lecture notes in computer science*, pp 87–124. Springer
13. Clarke EM, Draghicescu IA, Kurshan RP (1993) A unified approach for showing language inclusion and equivalence between various types of omega-automata. Inf Process Lett 46(6):301–308
14. Czerbo DB (1992) Handbook of theoretical computer science: van leeuwen J (ed), vol. A: algorithms and complexity, vol. B: formal methods and semantics (Elsevier, Amsterdam, 1990), p 2296, hardcover, dfl. 555.00. *Artif Intell Medicine* 4(4):309
15. Davenport JH, Heintz J (1988) Real quantifier elimination is doubly exponential. J Symb Comput 5(1/2):29–35
16. Dalsgaard AE, Hansen RR, Jørgensen KY, Larsen KG, Olesen MC, Olsen P, Srba J (2011) opaal: A lattice model checker. In: *NASA formal methods*, volume 6617 of *Lecture notes in computer science*, pp 487–493. Springer
17. Dierks H, Kupferschmid S, Larsen KG (2007) Automatic abstraction refinement for timed automata. In *FORMATS*, volume 4763 of *Lecture notes in computer science*, pp 114–129. Springer
18. de Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pp 337–340. Springer
19. Ergurtuna M, Yalcinkaya B, Gol EA (2022) An automated system repair framework with signal temporal logic. Acta Inf 59(2):183–209
20. Field A (2013) *Discovering statistics using IBM SPSS statistics: and sex and drugs and rock 'n' roll, 4th Edition*. Sage
21. Garey MR, Johnson DS (1979) *Computers and intractability: a guide to the theory of NP-Completeness*. W. H. Freeman
22. Ghosh S, Sadigh D, Nuzzo P, Raman V, Donzé A, Sangiovanni-Vincentelli AL, Sastry SS, Seshia SA (2016) Diagnosis and repair for synthesis from signal temporal logic specifications. In: Abate A, Fainekos G (eds) *Proceedings of the 19th international conference on hybrid systems: computation and control, HSCC 2016, Vienna, Austria, April 12–14, 2016*, pp 31–40. ACM
23. Henzinger TA, Nicollin X, Sifakis J, Yovine S (1994) Symbolic model checking for real-time systems. Inf Comput 111(2):193–244
24. Hopcroft JE, Ullman JD (2000) Introduction to automata theory, languages and computation, 2nd edn. Addison-Wesley, Boston
25. Isberner M, Howar F, Steffen B (2015) The open-source learnlib - A framework for active automata learning. In: *CAV (1)*, volume 9206 of *Lecture notes in computer science*, pp 487–495. Springer
26. Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. IEEE Trans Softw Eng 37(5):649–678
27. Jensen HE, Larsen KG, Skou A (1996) Modelling and analysis of a collision avoidance protocol using spin and uppaal. In: *The Spin verification system*, volume 32 of *DIMACS series in discrete mathematics and theoretical computer science*, pp 33–50. DIMACS/AMS
28. Jose M, Majumdar R (2011) Bug-assist: assisting fault localization in ANSI-C programs. In: *CAV*, volume 6806 of *Lecture notes in computer science*, pp 504–509. Springer,

29. Jiang Z, Pajic M, Moarref S, Alur R, Mangharam R (2012) Modeling and verification of a dual chamber implantable pacemaker. In: *TACAS*, volume 7214 of *Lecture notes in computer science*, pp 188–203. Springer
30. Karmarkar N (1984) A new polynomial-time algorithm for linear programming. In: Proceedings of the sixteenth annual ACM symposium on Theory of computing, pp 302-311
31. Kölbl M, Leue S, Wies T (2019) Clock bound repair for timed systems. In: *CAV (1)*, volume 11561 of *Lecture notes in computer science*, pp 79–96. Springer
32. Kölbl M, Leue S, Wies T (2020) Tartar: a timed automata repair tool. *CoRR*, arXiv:2002.02760. Also available from URL https://www.sen.uni-konstanz.de/publications
33. Kroening D, Strichman O (2016) *Decision procedures - an algorithmic point of view, Second Edition*. Texts in theoretical computer science. An EATCS Series. Springer
34. Korte B, Vygen J (2012) Combinatorial optimization: theory and algorithms. Algorithms and combinatorics. Springer, Berlin Heidelberg
35. Le X-BD, Chu D-H, Lo D, Le GC, Visser W (2017) S3: syntax- and semantic-guided repair synthesis via programming by examples. In: *Proceedings of the 2017 11th joint meeting on foundations of software engineering, ESEC/FSE 2017*, pp 593–604. ACM
36. Liu S (2018) Analysing timed traces using SMT solving. Master's thesis, University of Konstanz
37. Apache Software Foundation (2019) *Maven*, https://maven.apache.org/
38. Milner R(1980) *A calculus of communicating systems*, volume 92 of *Lecture notes in computer science*. Springer
39. opaal (2011). opaal test folder. http://opaal-modelchecker.com/opaal-ltsmin/. Accessed: 2018-11-08
40. Polsen DB, van Vliet J (2010) Concrete delays for symbolic traces. Master's thesis, Department of Computer Science, Aalborg University. Available from https://projekter.aau.dk/projekter/files/32183338/report.pdf
41. Reynolds A, Kuncak V, Tinelli C, Barrett C, Deters M (2017) Refutation-based synthesis in smt. *Formal methods in system design*, Feb
42. Brito T (2015) Uppaal elevator example. https://github.com/tfbrito/UPPAAL. Accessed: 2019-01-20
43. Uppaal (2017) Uppaal benchmarks. http://www.it.uu.se/research/group/darts/uppaal/benchmarks/#benchmarks. Accessed: 2019-01-20
44. Yovine S (1997) KRONOS: a verification tool for real-time systems. STTT 1(1–2):123–133
45. Yi W, Pettersson P, Daniels M (1994) Automatic verification of real-time communicating systems by constraint-solving. In: *FORTE*, volume 6 of *IFIP conference proceedings*, pp 243–258. Chapman and Hall. Full version of the paper is available from http://www.it.uu.se/research/group/darts/papers/texts/wpd-forte94-full.pdf
46. Microsoft Research (2019) *The Z3 Theorem Prover*. https://github.com/Z3Prover/z3

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.