Towards Practical Verification of Reachability Checking for Timed Automata

Simon Wimmer[0000−0001−5998−4655] and
Joshua von Mutius[0000−0002−8268−0075]

Fakultät für Informatik, Technische Universität München
wimmers@in.tum.de joshua.von-mutius@tum.de

Abstract. Prior research has shown how to construct a mechanically verified model checker for timed automata, a popular formalism for modeling real-time systems. We extend this work to improve its value for practical timed automata model checking in two ways.
First, we shift the focus from verified model checking to certifying unreachability. This allows us to benefit from better approximation operations for symbolic states, and reduces execution time by exploring fewer states and by exploiting parallelism. Moreover, this gives us the ability to audit results of unverified model checkers that implement a range of further optimizations, including certificate compression.
Second, we provide an improved modeling language that includes the popular modeling features of broadcast channels as well as urgent and committed locations. The resulting tool is evaluated on a set of standard benchmarks to demonstrate its practicality, using a new unverified model checker implementation in Standard ML to construct the certificates.

Keywords: Timed automata · Interactive Theorem Proving · Isabelle/HOL.

Timed automata [1] are a widely used formalism for modeling real-time systems, which is employed in a class of successful model checkers such as Uppaal [4]. These tools can be understood as trust-multipliers: we trust their correctness to deduce trust in the safety of systems checked by these tools. As a consequence, one wants to ensure as rigorously as possible that the computation results of timed automata model checkers are correct.

Previous work [27] has addressed this problem by constructing a model checker for timed automata that is fully verified using Isabelle/HOL [22]. This tool is intended to be a reference implementation that can be used to scrutinize the correctness of other model checkers. As such, it is mainly able to check small and medium-sized benchmark examples, but the performance gap w.r.t. more practical model checkers prevents it from checking realistic benchmark models within reasonable time and space bounds. Moreover, its modeling language is a rather ad-hoc translation of Uppaal's modeling language and lacks commonly used modeling features such as broadcast transitions and committed locations.

We address these issues by providing a new modeling language, and shifting the focus from full verified model checking to only certifying that the result

produced by an unverified model checker is correct. We only study reachability (for now): it is the most important property that is checked with timed automata model checkers, and some model checkers only support reachability. It is crucial to ensure that a bad state is certainly not reachable if the model checker claims so, thus we want to certify *unreachability*. Certifying that a state is indeed reachable would amount to extracting a timed trace and certifying that the trace is compatible with the model. While implementing this in a verified manner would be comparatively easy, we consider it less important because it corresponds to the bug finding functionality of model checkers, which carries less trust.

The recipe for certifying unreachability is simple: the model checker explores a number of states until it determines that there are no more states to be found. If none of the states fulfill the final state predicate (i.e. violates the safety property), then the model checker will answer "unreachable". We use the set of explored states as the unreachability certificate. In essence, we only need to check that the initial state is contained in this set, that there are no outgoing edges from this set, and that none of the states in the set fulfill the final state predicate.

The switch to certification holds many advantages. Timed automata model checking uses *over-approximations* of symbolic states to ensure termination. A large variety of these approximation operators has been studied [2,3,12]. Our previous work [26] has shown that, while formally proving the correctness of these approximation operations is feasible in principle with an interactive theorem prover, the effort is rather high. Instead, to *certify* unreachability, it is sufficient to only know that the approximation operator indeed yields a state that is at least as big as the precise symbolic state. Certifying this property is cheap.

Moreover, certification eases *parallelization*. Checking that a state is not final and that all its successors are covered by the state set are local properties. We show how to exploit this in a verified implementation, while only mildly increasing the verification effort and the size of the trusted code base.

Finally, the number of states explored by a model checker can vary immensely, depending on a range of factors such as the chosen approximation operator or the search order. Thus, an efficient unverified tool can exploit different heuristics and strategies to compute a state space that is as small as possible, and thereby speedup the certification effort. In this context, we also study a number of *compression techniques* to reduce the number of states in the certificate after the model checker has concluded its search.

We use a new unverified model checker called Mlunta, which is implemented in Standard ML (SML), to generate certificates for a set of standard benchmarks, and evaluate our verified certifier's performance on these benchmarks [1].

Related Work This work is based on an existing Isabelle/HOL formalization of timed automata model checking [26,27]. Other proof-assistant formalizations of timed automata focus on proving elementary properties about the basic formalism [28,29], or proving properties about concrete automata [23,9,7], but none of them are concerned with model checking.

[1] Both tools are available online: https://home.in.tum.de/~wimmers/TA_Certification.

Earlier work formalizes a model checker for the modal μ-calculus [25], and constructs a verified finite state LTL model checker [8,21,5].

The idea of extracting certificates from the model checking process has previously been studied in the context of the μ-calculus [20] and finite state LTL model checking [24]. However, these works are not accompanied by a verified certificate checker and do not attempt to scale the approach to practical examples. Only the recent work of Griggio et al. [10] provides a practical extraction mechanism and a certificate checker for LTL model checking, but the checker is not verified. To the best of our knowledge, we are the first to examine certification in the context of timed automata model checking.

Isabelle/HOL Isabelle/HOL [22] is an interactive theorem prover based on Higher-Order Logic (HOL). HOL can be thought of as a combination of a functional programming language and mathematical logic. Isabelle/HOL mostly resembles standard mathematical notation. Some conventions that are borrowed from functional programming need to be explained, however. Functions are mostly curried, i.e. of type $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau$ instead of $\tau_1 \times \tau_2 \Rightarrow \tau$. As a consequence, function application is usually denoted as $f\ a\ b$ instead of $f(a, b)$. Function abstraction with lambda terms uses the standard syntax $\lambda x.\ t$ (the function that maps x to t) and can also have paired arguments $\lambda(x, y).\ t$. Type variables are written $'a$, $'b$, etc. Compound types are written in postfix syntax: $\tau\ set$ is the type of sets of elements of type τ. We use the Isabelle/HOL convention that free variables are implicitly all-quantified throughout the paper. In parts of the paper, formulas or syntax have been simplified for readability, but we have stayed largely faithful to the Isabelle/HOL formalization.

Contributions In short, these are the main contributions of our work:

- To the best of our knowledge, we are the first to study certification of the model checking results of reachability checking for timed automata, including techniques to compress certificates.
- We construct a verified implementation of such a certificate checker.
- We give a formal semantics and a formalized on-the-fly product construction for a timed automata modeling language including broadcast channels as well as urgent and committed locations.

Outline The remainder of the paper is organized as follows. The first section briefly recalls the theory of timed automata, and sketches the state-of-the-art model checking process. The second section describes our modeling language and the verified on-the-fly product construction. Section three explains how, starting from an abstract theory, a concrete verified implementation of the certificate checker can be obtained. Section four illustrates two techniques to improve the certificate checker's performance, while only mildly increasing the formalization effort. Section five discusses two methods for certificate compression. The paper is concluded by an experimental evaluation and remarks on potential future work.

1 Timed Automata and Model Checking

Transition Systems We take a very simple view of transition systems: they are simply a relation $\rightarrow$ of type $'a \Rightarrow 'a \Rightarrow bool$ for a type of states $'a$. We write $a \rightarrow^* b$ to denote that b can be reached from a via a sequence of $\rightarrow$-transitions.

Timed Automata To make the paper self-contained, this paragraph briefly describes timed automata and is mostly reproduced from Wimmer and Lammich [26]. For a thorough introduction see the tutorial paper of Bengtsson and Yi [4].

Compared to standard finite automata, timed automata introduce a notion of clocks. Fig. 1 depicts an example of a timed automaton. We will assume that clocks are of type *nat*. A *clock valuation* u is a function of type $nat \Rightarrow real$. Loca-

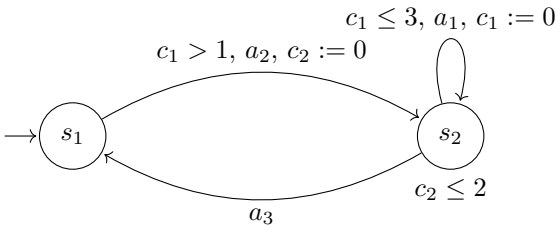

Fig. 1: Example of a timed automaton with two clocks.

tions and transitions are guarded by *clock constraints*, which have to be fulfilled to stay in a location or to take a transition. Clock constraints are conjunctions of constraints of the form $c \sim d$ for a clock c, an integer d, and $\sim \in \{<, \leq, =, \geq, >\}$. We write $u \models cc$ if the clock constraint cc holds for the clock valuation u. We define a timed automaton A as a pair $(\mathcal{T}, \mathcal{I})$ where $\mathcal{I}$ is a mapping from locations to clock constraints (also named invariants); and $\mathcal{T}$ is a set of transitions written as $A \vdash l \longrightarrow^{g,a,r} l'$ where l and l' are start and successor location, g is the guard of the transition, a is an action label, and r is a list of clocks that will be reset to zero when the transition is taken. States of timed automata are pairs of a location and a clock valuation. The operational semantics defines two kinds of steps (given as their HOL descriptions):

- Delay: $(l, u) \rightarrow^d (l, u \oplus d)$ if $d \geq 0$ and $u \oplus d \models \mathcal{I}\ l$;
- Action: $(l, u) \rightarrow_a (l', [r \rightarrow 0]u)$
  if $A \vdash l \longrightarrow^{g,a,r} l'$, $u \models g$, and $[r \rightarrow 0]u \models \mathcal{I}\ l'$;

where $u \oplus d = (\lambda c.\ u\ c + d)$ offsets all clocks by d in the valuation u, and $[r \rightarrow 0]u = (\lambda c.\ \mathsf{if}\ c \in r\ \mathsf{then}\ 0\ \mathsf{else}\ u\ c)$ resets all clocks in r to 0 in valuation u. For any (timed) automaton A, we consider the transition system

$$(l, u) \rightarrow_A (l', u') = (\exists d \geq 0.\ \exists a\ u''.\ (l, u) \rightarrow^d (l, u'') \wedge (l, u'') \rightarrow_a (l', u'))\,.$$

That is, each transition consists of a delay step that advances all clocks by some amount of time, followed by an action step that takes a transition and resets the clocks annotated to the transition. Given a final state predicate F and an initial state (l_0, u_0), we are interested in whether $(l_0, u_0) \rightarrow_A^* (l, u)$ for any l, u with $F\, l$.

Model Checking Due to the use of clock valuations, the state space of timed automata is inherently infinite. Thus, model checking algorithms for timed automata are based on the idea of abstracting from concrete valuations to *sets* of clock valuations of type $(nat \Rightarrow real)\ set$, often called *zones*. The state space is explored in an *on-the-fly* manner, computing successors on zones, which are typically represented symbolically as *Difference Bound Matrices* (DBMs). Knowledge of this data structure is not necessary to understand the rest of the paper, thus we refer the interested reader to Bengtsson and Yi [4] and to Wimmer and Lammich [26,27] for a verification of this data structure. In the remainder we will only use the term "zones" and not refer to their implementation as DBMs.

The delicate part of this method is that the number of reachable zones could still be infinite. Therefore, over-approximations of zones are computed to obtain a finite search space. We call the transition system of zones the *zone graph*, and the version where over-approximations are applied the *abstract zone graph* [11]. For a number of such approximation operators, it can be shown that the abstract zone graph is sound and complete [2]. The proofs are rather intricate, however. Thus formalizing them would be a big effort. By focusing on certification of unreachability, this problem vanishes, as we only need to ensure that any state we deem reachable is *subsumed* by (i.e. semantically a subset of) some state that is part of the certificate and that was computed by the over-approximation.

2 Incorporating Practical Modeling Features

Practical modeling with the help of timed automata relies on having a rich modeling language that at its core typically supports networks of timed automata, which share a form of discrete state, and which communicate over channels. Extensions to the typical formalism have been invented to either ease the modeling process, or to alleviate state space explosion. In the former category, the state-of-the-art tool Uppaal even supports a C-like input language that can be used to express guards and updates, and the discrete state consists of variables and arrays of different types. The features of urgent locations and broadcast channels can be considered to belong to both categories, while committed locations more clearly belong to the latter category. Previous work [27] already discusses how to incorporate the C-like features of Uppaal's input language in a verified model checker. Thus we focus on the features of urgent and committed locations and broadcast channels here. In addition, we provide a less ad-hoc modeling language such that models can be specified in a human-readable format, closing the gap between the actual model and the formalized semantics.

[2] Soundness: for every abstract run, there is a concrete instantiation. Completeness: every concrete run can be abstracted.

2.1 A Formal Semantics

This section briefly describes the formal semantics of our modeling language. We extend single automata to use transitions that include a guard and an update on the discrete state, i.e. transitions are now of the form $A \vdash l \longrightarrow^{b,g,a,f,r} l'$ for a Boolean expression b and a discrete update f. The discrete state is characterized by a number of integer state variables. We write $s \vdash_b b$ to denote that state s satisifes the Boolean expression b, $s \xrightarrow{f} s'$ to denote that f updates s to s', and finally $s \overset{fs}{\Rightarrow} s'$ to denote that the list fs updates s to s'. We do not sketch out the detailed definitions of these semantic primitives as they are in principle parametric within our formalization. Actions are of one of three types: input $a?$, output $a!$, or internal a. Each automaton is associated with a set of committed locations $\mathcal{C}$ and a set of urgent locations $\mathcal{U}$ (i.e. automata are now a quadruple $(\mathcal{C}, \mathcal{U}, \mathcal{T}, \mathcal{I})$).

A full network of timed automata is a triple $(\mathcal{B}, \mathcal{N}, \mathcal{V})$ for $\mathcal{B}$ a set of labels that mark the broadcast channels, $\mathcal{N}$ a list of single automata, and $\mathcal{V}$ a mapping from state variable names to a pair of a lower and an upper bound specifying the range of the variable. An individual automaton can take an internal transition at any time provided that the transition is enabled. In a binary transition, a pair of automata can synchronize on a label a given that one of them has an enabled transition labeled $a!$ and the other one has an enabled transition labeled $a?$. This CCS-style type of synchronization is used in favour of CPS-style synchronization chiefly because this seems to be the prevalent choice in timed automata literature and model checkers. A broadcast transition can be taken when a process has an outgoing action $a!$ for $a \in \mathcal{B}$. Any process with an enabled transition labeled with $a?$ will synchronize on this broadcast. If any process is in a committed location, then the next transition has to involve at least one process that is currently in a committed location. In an urgent location, time delay is not allowed.

The following specifies the semantics for delay and broadcast transitions, omitting internal and binary transitions for brevity:

$$\frac{\begin{array}{cc}\forall p < |\mathcal{N}|.\, u \oplus d \models \mathcal{I}_p\ L_p & d \geq 0 \\ (\exists p < |\mathcal{N}|.\, L_p \in \mathcal{U}_p) \longrightarrow d = 0 & \mathit{bounded}\ \mathcal{V}\ s\end{array}}{(\mathcal{B}, \mathcal{N}, \mathcal{V}) \vdash \langle L, s, u\rangle \rightarrow_{Del} \langle L, s, u \oplus d\rangle}$$

$$\frac{\begin{array}{ccccccc} n = |\mathcal{N}| & p < n & L_p = l & ps \subseteq \{i \mid i < n\} & p \notin ps & a \in \mathcal{B} \\ (l, b, g, a!, f, r, l') \in \mathcal{T}_p & & & \forall p < n.\, u' \models \mathcal{I}_p\ L'_p & & \end{array} \\ \boxed{l \in \mathcal{C}_p \vee (\exists p \in ps.\, L_p \in \mathcal{C}_p) \vee (\forall p < n.\, L_p \notin \mathcal{C}_p)} \\ \boxed{\forall p \in ps.\, (L_p, bs\, p, gs\, p, a?, fs\, p, rs\, p, ls'\, p) \in \mathcal{T}_p} \\ \begin{array}{ccccc} s \vdash_b b & \forall p \in ps.\, s \vdash_b bs\, p & u \models g & \forall p \in ps.\, u \models gs\, p & \end{array} \\ \boxed{\begin{array}{l}\forall q < n.\, q \notin ps \wedge p \neq q \longrightarrow \\ (\forall b\, g\, f\, r\, l'.\, (L_q, b, g, a?, f, r, l') \in \mathcal{T}_q \longrightarrow \neg s \vdash_b b \vee \neg u \models g)\end{array}} \\ \begin{array}{cc} L' = L[ps := ls', p := l'] & u' = [rs \to 0][r \to 0]u \end{array} \\ \begin{array}{ccc} s \xrightarrow{f} s' & s' \xrightarrow{map\ fs\ ps} s'' & \mathit{bounded}\ \mathcal{V}\ s'' \end{array}}{(\mathcal{B}, \mathcal{N}, \mathcal{V}) \vdash \langle L, s, u\rangle \rightarrow_{Broad\ a} \langle L', s'', u'\rangle}$$

As can be seen from the definition, the semantics of broadcast transitions are rather involved for two reasons. First, we need to state that either no process currently is in a committed location, or that one of these processes is involved in the transition (first box). Second, we need to select a set of receiving processes ps. In the semantics, a number of mappings bs, gs, fs, rs, and ls' is used to select a transition for each process (second box). This set needs to be maximal (third box), i.e. if process q has an enabled transition labeled with $a?$, then $q \in ps$. Note that it is important that processes are ordered to enforce a deterministic result for the updates of state and clock variables.

2.2 Product Construction & Renaming

Typically, complex modeling features, such as provided by the formalism described in the last section, are handled by first constructing a single automaton, called *product automaton* from the input model, and then applying model checking algorithms for single automata on the product automaton. We also follow this approach here.

It is rather straightforward to provide the definition of a product automaton $A' = \mathcal{A}\,(\mathcal{B}, \mathcal{N}, \mathcal{V})$ from $(\mathcal{B}, \mathcal{N}, \mathcal{V})$, and to prove bisimulation:

$$(\exists t.\,(\mathcal{B}, \mathcal{N}, \mathcal{V}) \vdash \langle L, s, u\rangle \rightarrow_t \langle L', s', u'\rangle) \longleftrightarrow ((L, s), u) \rightarrow_{A'} ((L', s'), u')\,.$$

The formal proofs are quite automated (as they mainly involve reasoning on sets and elementary propositions). The hard part is to turn this into an executable on-the-fly construction. The goal is to describe the transitions of A' as an executable function $s_{A'}$ such that $set\,(s_{A'}\,l) = \{l' \mid A' \vdash l \longrightarrow^{g,a,r} l'\}$ for all reachable l. The implementation of $s_{A'}$ constructs a function for each of the three types of action transitions and the correctness proof is split accordingly. Each of them involves proving two directions of an equivalence to show the set equality.

While none of the individual reasoning steps is conceptually involved, it is a challenge to keep the proof manageable in Isabelle/HOL. The complexity mainly comes from the sheer size and number of terms involved in the construction. This can make Isabelle's proof tactics deteriorate and severely slow down pretty-printing of the proof state, which renders interactive development painful.

To fill in the big picture, two pieces of the puzzle are still missing. First, we assumed that all labels (for locations, actions, clocks, and variables) use (consecutive) natural numbers in the automaton description above. Second, we assumed that time constants (i.e. the range of clock constraints) are integers instead of real numbers, which is what the semantics is defined on.

More precisely, we assume that we are given an input model $(\mathcal{B}, \mathcal{N}, \mathcal{V})$ that uses e.g. strings as labels, and integers for time constants. The semantics of this model are defined on $(\mathcal{B}, map\ conv_A\ \mathcal{N}, \mathcal{V})$ where $conv_A$ converts all time constants of a single automaton from integers to reals[3]. As a pre-processing step, we convert the model such that all strings are renamed to consecutive natural

[3] $map\ f\ xs$ is the list that is obtained by applying f to every element of xs.

numbers, using some renamings r_A, r_N and r_V. We establish the following chain of bisimulations (where $\sim$ means bisimulation w.r.t. some suitable relation):

$$\begin{aligned}
&(\mathcal{B}, \mathit{map}\ \mathit{conv}_A\ \mathcal{N}, \mathcal{V}) \\
&\sim (r_B\ \mathcal{B}, \mathit{map}\ r_N\ (\mathit{map}\ \mathit{conv}_A\ \mathcal{N}), r_V\ \mathcal{V}) && (1)\\
&= (r_B\ \mathcal{B}, \mathit{map}\ \mathit{conv}_A\ (\mathit{map}\ r_N\ \mathcal{N}), r_V\ \mathcal{V}) && (2)\\
&\sim \mathcal{A}\ (r_B\ \mathcal{B}, \mathit{map}\ \mathit{conv}_A\ (\mathit{map}\ r_N\ \mathcal{N}), r_V\ \mathcal{V}) && (3)\\
&= \mathit{conv}\ (\mathcal{A}\ (r_B\ \mathcal{B}, \mathit{map}\ r_N\ \mathcal{N}, r_V\ \mathcal{V})) && (4)
\end{aligned}$$

This allows us to decide model checking properties of $(\mathcal{B}, \mathit{map}\ \ \mathit{conv}_A\ \mathcal{N}, \mathcal{V})$ on $\mathit{conv}\ (\mathcal{A}\ (r_B\,\mathcal{B}, \mathit{map}\ r_N\ \mathcal{N}, r_V\,\mathcal{V}))$ using the existing verified model checker implementation for a single timed automaton [27]. Note that the converted model $(r_B\,\mathcal{B}, \mathit{map}\ r_N\ \mathcal{N}, r_V\,\mathcal{V})$ is what we apply the on-the-fly product construction to, to get an executable description of $\mathcal{A}\ (r_B\,\mathcal{B}, \mathit{map}\ r_N\ \mathcal{N}, r_V\,\mathcal{V})$.

Step (3) is the bisimulation theorem for the product construction, which we already stated above. The equalities of steps (2) and (4) can be proved automatically with the help of Isabelle's rewriting facilities. Step (1) requires r_B, r_V, and r_N to be injective functions on the corresponding domains given by $(\mathcal{B}, \mathcal{N}, \mathcal{V})$. Note that this condition can simply be checked programmatically. Our technique for this step is to first use a general theorem that states that any injective function can be extended to a bijection between the full domain type and the natural numbers, provided that the domain is of a countably infinite type (such as the type of strings).

It is crucial to use such fine-grained reasoning and to not try to prove the complete bisimulation in one go. Any attempts where we tried to fuse multiple reasoning steps into one lead to unbearably complex and unmanagable proofs.

3 From Model Checking to Certifying Unreachability

As stated above, this work starts from an existing formalization of timed automata model checking. This section describes how it is extended to turn it into a verified certifier. We study the case of a single timed automaton, which is sufficient due to the reduction described in the previous section.

3.1 An Abstract Correctness Theorem

To work towards a rigorous justification of the certification process, we first study the problem on a more abstract level. Consider a transition system $\rightarrow$ on states of type $'l \times 's$ where $'l$ corresponds to the finite state part of timed automata and $'s$ corresponds to zones. We assume an invariant P on states, i.e.:

$$P\ (l_1,\ s_1)\ \wedge (l_1,\ s_1) \rightarrow (l_2,\ s_2) \longrightarrow P\ (l_2,\ s_2)\,.$$

The interesting feature that sets timed automata model checking apart is subsumption. Recall that during the model checking process, it is possible to first

discover some state (l, Z) (a pair of a discrete state l and a zone Z), and to find at some later point that another reachable state (l, Z') subsumes (l, Z) because Z' semantically contains Z', i.e. $Z \subseteq Z'$. At this point the state (l, Z) can be discarded as we know that anything that is reachable from (l, Z) is also reachable from (l, Z'). Abstractly, subsumption is modeled by some fixed preorder (i.e. reflexive and transitive relation) $\preccurlyeq$ on $'s$ which is monotone w.r.t. $\rightarrow$:

$$\begin{array}{ll} & s_1 \preccurlyeq s_2 \wedge (l_1, s_1) \rightarrow (l_2, t_1) \wedge P\ (l_1, s_1) \wedge P\ (l_2, s_2) \\ \longrightarrow & (\exists t_2.\ t_1 \preccurlyeq t_2 \wedge (l_1, s_2) \rightarrow (l_2, t_2)) \end{array}$$

In the abstract setting, a certificate consists of a set of discrete states L of type $'l\ set$, and a mapping M of type $'l \Rightarrow 's\ set$ that gives the set of reachable symbolic states that were computed for any discrete state $l \in L$. The certificate (L, M) needs to be *invariant* under P:

$$l \in L \wedge s \in M\, l \longrightarrow P\,(l,\, s)\,.$$

Moreover, it needs to be *closed*. Following Herbreteau et al. [11], we call a state *covered* if it is subsumed by another state in the certificate. A certificate is closed if for each state in the certificate all its successors are covered:

$$l_1 \in L \wedge s_1 \in M\, l_1 \wedge (l_1,\, s_1) \rightarrow (l_2,\, s_2) \longrightarrow l_2 \in L \wedge (\exists s_3 \in M\, l_2.\, s_2 \preccurlyeq s_3) \quad (*)$$

We can easily prove the following key theorem stating that all reachable states are covered if the initial state is covered:

Theorem 1. *Let (L, M) be closed and invariant under P. Assume $l_0 \in L$, $s_0' \in M\, l_0$, $s_0 \preccurlyeq s_0'$, and $(l_0,\, s_0) \rightarrow^* (l_1,\, s_1)$. Then $l_1 \in L$ and there exists s_2 such that $s_2 \in M\, l_1$ and $s_1 \preccurlyeq s_2$.*

Proof. By induction on the number of steps in $(l_0,\, s_0) \rightarrow^* (l_1,\, s_1)$.

We will now say that a certificate (L, M) is *admissible* iff

- it is invariant under P,
- it is closed,
- it covers the initial state (i.e. there is an $s_0' \in M\, l_0$ such that $s_0 \preccurlyeq s_0'$),
- and there is no $l \in L$ with $F\, l$.

Corollary 1. *If F is monotone w.r.t. $\preccurlyeq$ and the certificate (L, M) is admissible, then $\nexists l\, s.\, (l_0,\, s_0) \rightarrow^* (l,\, s) \wedge F\, l$.*

3.2 An Abstract Certificate Checker

From the theoretical analysis laid out in the last section, we can derive the following strategy for certifying unreachability:

- An unverified model checker explores the reachable state space of a given model symbolically and checks that none of the discovered states (l, s) fulfills $F\, l$.

- The set of explored states is emitted as a certificate, possibly followed by compression (see section 5).
- The model, the certificate, and a description of the renaming that was used for the states are passed to the verified certifier.
- The certifier checks that the given renaming is injective, renames the model accordingly, applies the product construction and checks that the certificate is admissible.

If the process is successful, we can conclude by Corollary 1 that no "bad" state (l, s) (i.e. with $F\,l$) is reachable symbolically. We will argue that this really implies that the model is safe in the concrete case of timed automata in section 3.3.

We now lay out how a verified certificate checker that implements said strategy for an abstract transition system can be constructed in Isabelle/HOL. Listing 1.1 displays the definition of the core of the checker that checks whether the certificate is closed in the sense defined above.

```
1  definition check (L, M) ≡
2    monadic_list_all L (λl. do {
3      let S = M l;
4      let next = succs l S;
5      monadic_list_all next (λ(l', S'). do {
6        xs ← SPEC (λxs. set xs = S');
7        if xs = [] then return True else do {
8          b1 ← return (l' ∈ L);
9          ys ← SPEC (λxs. set xs = M l');
10         b2 ← monadic_list_all (λx.
11           monadic_list_ex (λy. return (x ≼ y)) ys
12         ) xs;
13         return (b1 ∧ b2)
14       }
15     })
16   })
```

Listing 1.1: Monadic program to check whether a certificate is closed.

The program is defined in the non-determinisim monad of the Imperative Refinement Framework (IRF) [17]. Some parts, such as checking set membership or converting a (finite) set to a list are still left abstract. A non-deterministic specification $\mathsf{SPEC}\,Q$ returns some value v with $Q\,v$.

The body of the program (lines 2-16) iterates over all discrete states in the certificate L and checks that all corresponding symbolic states are covered. Line 3 retrieves the symbolic states corresponding to discrete state l and in line 4 their symbolic successor states are computed. The result (*next*) is a list of pairs of a discrete state and the set of its corresponding symbolic states. The loop ranging from lines 5 to 15 iterates over this list to ensure that all the successor states are covered. Given a discrete state l' and a set of symbolic states S', line 6 first converts it into a list xs that can be iterated over. This turns into a vacuous operation when the algorithm is refined to an executable version where sets are implemented as lists. Line 8 checks that l' is also part of the certificate. Then, in

line 9 the set of corresponding symbolic states is retrieved and converted to a list ys. Finally, lines 10-12 ensure that all states in xs are subsumed by some state in ys.

To prove soundness of *check*, we mainly need correctness theorems for the combinators *monadic_list_all* and *monadic_list_ex* that do what their names suggest. This is the correctness theorem for *monadic_list_all*, for instance:

$$\begin{array}{ll} & (\forall x.\, P_i\, x \leq \mathsf{SPEC}\,(\lambda r.\, r \longleftrightarrow P\, x)) \\ \longrightarrow & \mathit{monadic\_list\_all}\, xs\, P_i \leq \mathsf{SPEC}\,(\lambda r.\, r \longleftrightarrow \mathit{list\_all}\, xs\, P) \end{array}$$

where $\mathit{list\_all}\, xs\, P$ holds if and only if P holds for all elements in xs. After setting up the IRF's verification condition generator with this rule and the corresponding rule for *monadic_list_ex*, it is easy to prove that *check* is sound:

$$\mathit{check}\,(L, M) \leq \mathsf{SPEC}\,(\lambda r.\, r \longrightarrow \mathit{closed}\,(L, M))$$

where the property $\mathit{closed}\,(L, M)$ corresponds to condition $(*)$ from above.

We then use standard refinement techniques to obtain an algorithm check_i that refines *check* replacing sets by lists. However, the algorithm is still specified in the non-determinisim monad and therefore not executable. We use a simple technique to make it executable. Consider the following theorem for *monadic_list_all*:

$$\mathit{monadic\_list\_all}\, xs\, (\lambda x.\, \mathsf{return}\,(P\, x)) = \mathsf{return}\,(\mathit{list\_all}\, xs\, P)\,.$$

It allows us to push a return to the outside of *monadic_list_all*. By exhaustively applying a set of such rewrite rules we obtain an alternative definition of check_i where return appears only on the outermost level, and the inner term is deterministic and thus executable. Using these techniques, we obtain a simple certificate checker that is executable, provided that we can implement the elementary model checking primitives such as the subsumption check or computing the list of successors of a state.

3.3 Transferring the Correctness Theorem

For timed automata, the abstract transition system studied above is the zone graph $\rightarrow_{ZG(A)}$ of a given (single) automaton A. We can show that it simulates $\rightarrow_A$:

$$(l, u) \rightarrow_A (l', u') \wedge u \in Z \longrightarrow (\exists Z'.\, (l, Z) \rightarrow_{ZG(A)} (l', Z') \wedge u' \in Z')\,.$$

This simulation property is sufficient to establish that if there is no reachable state (l, Z) in $\rightarrow_{ZG(A)}$ with $F\, l$, then no final state (l, u) is reachable in $\rightarrow_A$:

$$\begin{array}{ll} & (\nexists l, Z.\, (l_0, Z_0) \rightarrow_{ZG(A)} (l, Z) \wedge F\, l) \wedge u_0 \in Z_0 \\ \longrightarrow & (\nexists l, u.\, (l_0, u_0) \rightarrow_A (l, u) \wedge F\, l) \end{array}$$

In the formalization, these proofs rely on instantiating a general theory of simulations in transition systems that is derived from the theory of Wimmer and

Lammich [27]. From Corollary 1 we get that there is no reachable final state in $\rightarrow_{ZG(A)}$ if the certificate check is passed. Finally, with the bisimulation theorem from section 2.2, we can conclude that there is no final reachable state in the input model if there is no final reachable state in $\rightarrow_A$.

3.4 Implementing a Concrete Checker

All the elementary model checking primitives we need for certification have already been implemented [27]. The abstract implementation presented above assumes that the model checking primitives are implemented in a purely functional manner (as they are just regular HOL functions). The existing (verified) model checker [27], however, is an imperative implementation in the Imperative HOL framework. Imperative HOL [6] is a framework for specifying and reasoning about imperative programs in Isabelle/HOL. It provides a *heap monad* in which imperative programs can be expressed. Usually, once we have used an imperative implementation anywhere, the whole program would need to be stated in the heap monad. However, we can employ a technique similar to the one that is used for Haskell's ST monad [18] to erase the heap monad in a safe way under certain circumstances. As a consequence, we are able to reuse the existing verified model checking primitives, while being able to state the certificate checking algorithm purely functionally.

In the concrete checker, the mapping M is implemented using a verified functional hash table implementation based on so-called *diff arrays* [16]. This data structure provides a purely functional interface to an underlying imperative array. When a diff array is updated it performs the update on the imperative array, and stores a difference that can be used to re-compute the old state of the array. Reading from the most recent version of a diff array is fast as the value can directly be read from the underlying imperative array. If an old version is accessed, the whole array has to be copied to recompute the old version. This gives diff arrays good performance characteristics, as long as they are mostly used linearly. This is the case in our application as the hash table is filled in an initial phase, after which the hash table is used in a read-only manner.

3.5 Parallel Execution

The attentive reader may wonder why we care about a purely functional implementation of the certificate checker at all. Indeed, we could use existing techniques [27] to obtain an imperative implementation of the certificate checker in the heap monad. However, in this setting it would be hard to justify the soundness of executing parts of the checker in parallel. In the purely functional setting, this is much simpler. Our approach to parallel execution is minimalist: we only provide means to execute the *map* combinator on lists in parallel. This is achieved by another custom code translation that is part of the trusted code base. The parallel implementation of *map* uses a task queue that will contain the individual computations that need to be run for each element of xs, and uses a fixed number of threads to work through this list and assemble the final result.

We exploit this *map* implementation to work through the list of discrete states L in parallel, using the equivalence:

$$list\_all\, Q\, xs = list\_all\, id\, (map\, Q\, xs)\,.$$

In doing so, we lose the ability to stop execution early once a list element does not satisify Q. For the certificate checker, however, we assume that usually the certificate is correct, meaning that we have to go through the whole list anyway. We only parallelize the outermost loop of $check_i$ because this should yield reasonably-sized work portions, given that the size of L will typically be in the hundreds or thousands.

4 Scaling Performance

In this section we discuss two techniques to improve the performance of the certificate checker without increasing the verification effort significantly.

4.1 Monomorphization

Isabelle/HOL supports polymorphism and type classes, which are valuable features for sizeable formalization efforts. Large parts of our formalization also make use of these features, e.g. most of the timed automata semantics are formalized for a general time domain, and operations on DBMs are applicable on DBMs whose entries are formed from more general algebraic structures than the ring of integers. While this yields an abstract and general formalized theory, it can get in our way when trying to obtain efficient code.

When generating SML code from HOL, Isabelle uses a so-called dictionary construction to compile out type classes, which are not supported by SML. This means that most functions carry a large number of additional parameters, which are used to look up elementary operations, such as addition of two numbers. These additional lookup operations degrade performance. One solution is to ensure that all relevant constants that are exported to SML are monomorphic (i.e. specialized to the integer type), eliminating the need for the dictionary construction in most places. Thus, we apply a semi-automated procedure to achieve this monomorphization.

4.2 Integer Representation

Types such as *int* or *nat* are unbounded in Isabelle/HOL meaning they are implemented with the help of big integers in the target languages. To improve performance, we want to use machine integers instead, and instruct Isabelle/HOL's code generator to do that. This is still sound: SML's standard integer operations throw an exception if an overflow occurs instead of silently wrapping around. The code generator can only achieve partial correctness anyway: if program execution does not fail, then its result is consistent with the evaluated HOL term.

5 Certificate Compression

In this section, we present two techniques to compress the unreachability certificate. By compression we mean reducing the number of zones that are present in the certificate for each discrete state, using the unverified model checker. The first technique relies on subsumption. As explained above, it is possible that the model checker adds a zone Z to the set of explored states and later another zone Z' with $Z \preccurlyeq Z'$. Thus we filter the set w.r.t. to $\preccurlyeq$ in the end.

The second technique relies on the following idea: we replace one or more zones by their union, and check that the state space is still closed. This means that we have to check that all the successors of the larger zone are still covered by the current set of states. In that case, we can discard the old zones, and replace them by their union. We do not compute a precise union of zones but their convex hull. This operation is rather cheap as it amounts to taking the pointwise maximum of DBM entries. After computing the convex hull of a number of zones (in canonical form), we only need to apply the expensive operation to restore a canonical form once.

The latter technique yields a whole family of compression algorithms by iterating one of the following operations for each discrete state until a fixed-point is reached: a) the convex hull of all zones is computed; b) the convex hull of the first two zones is computed; c) the convex hull of the first two zones that can successfully be joined is put to the front of the list; d) same as c) but considering only discrete states for which compression was successful in the last round; e) same as d) but iterating the operation until saturation. An evaluation of these algorithms can be found in the next section. Analogously to the certificate checker, each iteration of the compression algorithm could in principle be parallelized by working on the discrete states in parallel.

6 Experimental Evaluation

We evaluate the checker on a set of benchmarks that is derived from Uppaal's standard benchmark suite [19]. Additionally, to cover the new modeling features, we use a set of benchmarks that is derived from the Penn pacemaker models [14] and a modified version of the HDDI benchmark with broadcast channels. A prototype SML implementation of a timed automata model checker (Mlunta) is used to compute the certificates. We use reachability properties of the form $\mathbf{E}\Diamond\ \mathit{false}$ to enforce that the model checker explores the complete state space. The results are given in Table 1. The problem size is specified as the number of automata in the network. We report the total runtime of the tandem consisting of Mlunta (using the first compression technique) and the (verified) certifier, and the individual runtime of certificate checking for a varying number of threads for parallel computation. The former is compiled with MLton, while the latter are compiled with Poly/ML as it is the only SML compiler that supports multi-threading. For comparison, we report the runtime of Uppaal on the same benchmarks, the runtime of an unverified SML implementation of the certificate checker based on Mlunta (compiled with Poly/ML), and the runtime of the fully verified model checker (Munta) [27] extended with the improvements from sections 2 and 4.

						Certifier for #threads				
Model	Size	Uppaal	Tandem	Munta	Unverif.	1-MLton	1	2	4	8
FDDI	8	0.43	1.42	1.46	0.46	0.62	2.17	1.70	1.46	1.35
FDDI broad	8	0.42	0.44	1.47	0.13	0.22	1.31	1.14	1.09	1.04
Fischer	5	0.36	3.38	7.54	2.03	1.71	3.17	2.31	1.95	1.74
CSMA	5	0.04	1.07	5.46	1.01	0.60	2.83	1.86	1.44	1.26
	6	2.06	15.3	69.30	13.98	7.24	20.40	11.60	7.49	6.14
	Mode									
Pacemaker	1	0.03	0.28	0.51	0.12	0.17	1.14	0.95	0.86	0.85
	2	0.04	1.05	4.02	0.84	0.54	2.46	1.78	1.42	1.27
	3	0.06	1.91	5.22	1.33	0.93	3.43	2.21	1.72	1.55
	4	0.03	14.30	0.96	10.31	8.37	16.90	9.81	6.75	5.83
	5	0.04	37.50	1.17	25.06	21.80	38.20	22.30	14.50	12.80

Table 1: Benchmarks results on a machine with 132 GB RAM and an Intel Xeon CPU E5-2699 v4 at 2.20GHz with 22 cores. The column labeled "Tandem" gives the runtime for a combination of the unverified SML tool and the verified certificate checker. The next column gives the runtime of the unverified SML certifier, followed by the runtimes of the verified checker for a varying number of threads. All times are given in seconds.

As can be seen from the results[4], the tandem is still one order of magnitude slower than Uppaal, but certificate checking in isolation is also up to one order of magnitude faster than the previous verified model checker [27]. Multi-core scale beyond two threads is relatively unsatisfactory, however. In micro-benchmarks, we have identified that the problem appears to be with memory allocation on the heap, even if no data is shared among threads (in our case, only the certificate is shared but successors are computed locally). There does not seem to be an obvious way to improve on this situation for SML implementations. Finally, one can see that the verified certifier is not drastically slower than the unverified implementation based on Mlunta, indicating that the verified certifier is not missing any obviously significant optimizations.

Table 2 gives the results of evaluating the different compression algorithms on the same set of benchmarks. The second variant is always applied to the compression result of the first variant to avoid trivial computations of the convex hull. Variant 2c (the most expensive one) can produce drastically smaller certificates than any other variant, and its minimum compression factor is an order of a magnitude higher than for any other variant. Nevertheless, only variants 1 and 2a appear to be useful in practice, as they are relatively cheap to compute. The other variants could prove useful if the certificates were produced by a significantly more efficient implementation, such as Uppaal or TChecker [13]. On a final note, we have constructed a more than 95% smaller but valid certificate for the Fischer benchmark, suggesting that there is room for improvement on the compression algorithms.

[4] Mlunta explores significantly more states than Uppaal and Munta for "Pacemaker".

		Variant					
Model	Size	1	2a	2b	2c	2d	2e
FDDI	8	0.21	0.21	1.72	69.53	3.65	3.43
FDDI broadcast	8	0.00	48.94	48.94	48.94	1.06	1.06
Fischer	5	22.03	22.03	22.72	43.06	30.40	30.40
CSMA/CD	5	26.06	41.54	43.84	81.16	58.94	47.54
	6	24.86	41.91	44.02	88.35	63.24	47.02
	Mode						
Pacemaker	1	16.07	25.00	30.80	58.04	29.02	29.02
	2	24.00	26.38	30.37	58.68	35.87	35.22
	3	12.96	17.62	19.23	46.92	25.30	25.01
	4	13.82	20.02	23.60	41.48	26.16	24.71
	5	17.14	22.48	25.46	39.69	28.18	26.88
Average		15.71	26.61	29.07	57.58	30.18	27.03

Table 2: Certificate compression factors (given in %).

7 Conclusion and Future Work

We have presented a verified certifier of unreachability certificates for a rich timed automata modeling formalism. The certificates are ought to be produced by an unverified model checker. Experimentation shows that verified certificate checking in isolation is up to an order of magnitude faster than what was previously possible with a verified model checker [27]. The performance of a tandem of an unverified model checker and the verified certifier could be improved by replacing the certificate-producing part with a highly optimized tool, possibly opening room to use some of the more powerful certificate compression techniques we suggested above. Moreover, more sophisticated tools also employ more powerful abstraction techniques, for which our proposed certification technique is still suitable.

Future Work We intend to extend this work to certification of Büchi emptiness in the future, using the idea of *subsumption graphs* [11] and using an unverified LTL model checking implementation for timed automata to produce the certificates [11,15]. As we pointed out above, there appears to be further room for improvement on the certificate compression algorithms.

In terms of the modeling language, an interesting possibility is to integrate the Uppaal-style bytecode language [27] for manipulating state variables. If clocks are not dealt with in the bytecode, this should be comparatively simple as the discrete state is mostly parametric in the current formalization. Furthermore, the concept of synchronization over channels could further be generalized by following the approach taken by TChecker. This would not only enrich the input language but could also simplify the product construction, by unifying the construction for binary and broadcast synchronization.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
2. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static guard analysis in timed automata verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. pp. 254–270. Springer Berlin Heidelberg (2003)
3. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. pp. 312–326. Springer Berlin Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_25
4. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Lectures on Concurrency and Petri Nets: Advances in Petri Nets. LNCS, vol. 3908, pp. 87–124. Springer (2004). https://doi.org/10.1007/978-3-540-27755-2_3
5. Brunner, J., Lammich, P.: Formal verification of an executable LTL model checker with partial order reduction. Journal of Automated Reasoning **60**(1), 3–21 (2018)
6. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Theorem Proving in Higher Order Logics (TPHOLs 2008). pp. 134–149 (2008). https://doi.org/10.1007/978-3-540-71067-7_14
7. Castéran, P., Rouillard, D.: Towards a generic tool for reasoning about labeled transition systems. In: TPHOLs 2001: Supplemental Proceedings (2001)
8. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: Proc. of CAV'13. LNCS, vol. 8044, pp. 463–478. Springer (2013)
9. Garnacho, M., Bodeveix, J., Filali-Amine, M.: A mechanized semantic framework for real-time systems. In: Proc. of FORMATS'13. pp. 106–120. LNCS 8053 (2013)
10. Griggio, A., Roveri, M., Tonetta, S.: Certifying proofs for LTL model checking. 2018 Formal Methods in Computer Aided Design (FMCAD) pp. 1–9 (2018)
11. Herbreteau, F., Srivathsan, B., Tran, T.T., Walukiewicz, I.: Why liveness for timed automata is hard, and what we can do about it. In: Lal, A., Akshay, S., Saurabh, S., Sen, S. (eds.) FSTTCS. LIPIcs, vol. 65, pp. 48:1–48:14. Schloss Dagstuhl (2016)
12. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Better abstractions for timed automata. Information and Computation **251**, 67–90 (2016)
13. Herbreteau, F., Point, G.: TChecker (2019), https://github.com/fredher/tchecker
14. Jiang, Z., Pajic, M., Moarref, S., Alur, R., Mangharam, R.: Modeling and verification of a dual chamber implantable pacemaker. In: Flanagan, C., König, B. (eds.) TACAS 2012. pp. 188–203. Springer Berlin Heidelberg (2012)
15. Laarman, A., Olesen, M.C., Dalsgaard, A.E., Larsen, K.G., van de Pol, J.: Multi-core emptiness checking of timed büchi automata using inclusion abstraction. In: Sharygina, N., Veith, H. (eds.) CAV. pp. 968–983. Springer Berlin Heidelberg (2013)
16. Lammich, P.: Collections framework. Archive of Formal Proofs (Nov 2009), http://isa-afp.org/entries/Collections.html, Formal proof development
17. Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015, Proceedings. LNCS, vol. 9236, pp. 253–269. Springer (2015). https://doi.org/10.1007/978-3-319-22102-1_17
18. Launchbury, J., Peyton Jones, S.: Lazy functional state threads. Proceedings of PLDI **29** (07 1998)
19. Möller, M.O.: Uppaal benchmarks (2017), https://www.it.uu.se/research/group/darts/uppaal/benchmarks

20. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. pp. 2–13. Springer Berlin Heidelberg (2001)
21. Neumann, R.: Using promela in a fully verified executable LTL model checker. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. pp. 105–114. Springer
22. Nipkow, T., Lawrence C. Paulson, Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9
23. Paulin-Mohring, C.: Modelisation of timed automata in Coq. In: Proc. of STACS'01. pp. 298–315. LNCS 2215 (2001)
24. Peled, D., Pnueli, A., Zuck, L.: From falsification to verification. In: Hariharan, R., Vinay, V., Mukund, M. (eds.) FST TCS 2001. pp. 292–304. Springer Berlin Heidelberg (2001)
25. Sprenger, C.: A verified model checker for the modal μ-calculus in Coq. In: TACAS 1998. pp. 167–183. Springer, London, UK (1998)
26. Wimmer, S.: Formalized timed automata. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016, Proceedings. LNCS, vol. 9807, pp. 425–440. Springer (2016). https://doi.org/10.1007/978-3-319-43144-4_26
27. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. pp. 61–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_4
28. Xu, Q., Miao, H.: Formal verification framework for safety of real-time system based on timed automata model in PVS. In: Proc. of the IASTED International Conference on Software Engineering, 2006. pp. 107–112 (2006)
29. Xu, Q., Miao, H.: Manipulating clocks in timed automata using PVS. In: Proc. of SNPD'09. pp. 555–560 (2009)