Verified Model Checking of Timed Automata

Simon Wimmer and Peter Lammich

Fakultät für Informatik, Technische Universität München

Abstract. We have constructed a mechanically verified prototype implementation of a model checker for timed automata, a popular formalism for modeling real-time systems. Our goal is two-fold: first, we want to provide a reference implementation that is fast enough to check other model checkers against it on reasonably sized benchmarks; second, we strive for maximal feature compatibility with the state-of-the-art tool Uppaal. The starting point of our work is an existing highly abstract formalization of reachability checking of timed automata. We reduce checking of Uppaal-style models to the problem of model checking a single automaton in this abstract formalization, while retaining the ability to perform on the fly model-checking. Using the Isabelle Refinement Framework, the abstract specification of the model checker is refined, via multiple intermediate steps, to an actual imperative implementation in Standard ML. The resulting tool is evaluated on a set of standard benchmarks to demonstrate its practical usability.

1 Introduction

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 [2]. These tools can be understood as trust-multipliers: we trust their correctness to deduce trust in the safety of systems checked by these tools. However, mistakes have previously been made. This particularly concerns an approximation operation that is used by model-checking algorithms to obtain a finite search space. The use of this operation induced a soundness problem in the tools employing it [3], which was only discovered years after the first model checkers were devised.

Our ongoing work[1] addresses this issue by constructing a fully verified model checker for timed automata, using Isabelle/HOL [4]. Our tool is not intended to replace existing model checkers, but to serve as a reference implementation against which other implementations can be validated. Thus, it must provide sufficient performance to check real world examples. To this end, we use the Isabelle Refinement Framework (IRF) [5, 6] to obtain efficient imperative implementations of the algorithms required for model checking.

Our work starts from an existing abstract formalization of *reachability* checking of timed automata [7]. To close the gap to a practical model checker, we need to address two types of issues: efficient implementation of abstract model checking algorithms, and expressiveness of the offered modeling formalism. Two kinds

[1] https://github.com/wimmers/munta

of algorithms deserve special attention here. The first are operations to manipulate Difference Bound Matrices (DBMs) [2], which represent abstract states. With the help of the IRF, we obtain efficient implementations of DBMs represented as arrays. The second are search algorithms that govern the search for reachable states. These algorithms are interesting in their own right, since they make use of *subsumption*: during the search process an abstract state can be ignored if a larger abstract state was already explored. We provide a generalized framework for different variants of search algorithms, including a version which resembles UPPAAL's unified passed and waiting list [2].

We aim to offer a modeling formalism that is comparable in its expressiveness to the one of UPPAAL. To accomplish this goal while keeping the formalization effort manageable, we opt to accept UPPAAL *bytecode* as input. At the current state of the project we have formalized the semantics of a subset of the bytecode produced by UPPAAL. We support the essential modeling features: networks of automata with synchronization, and bounded integer state variables. We apply a product construction to reduce models of this formalism to a single timed automaton. As in real model checkers, the whole construction is computed *on the fly*. However, not every bytecode input designates a valid automaton. To this end, we employ a simple *program analysis* to accept a sufficiently large subset of the valid inputs.

We conducted experiments on a small number of established benchmark models. The throughput of our model checker — the number of explored states per time unit — is within an order of magnitude of a version of UPPAAL running a comparable algorithm.

1.1 Isabelle/HOL

Isabelle/HOL [4] is an interactive theorem prover based on Higher-Order Logic (HOL). You can think of HOL as a combination of a functional programming language with logic. Although Isabelle/HOL largely follows ordinary mathematical notation, there are some operators and conventions that should be explained. Like in functional programming, functions are mostly curried, i.e. of type $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau$ instead of $\tau_1 \times \tau_2 \Rightarrow \tau$. This means that function application is usually written $f\ a\ b$ instead of $f(a, b)$. Lambda terms are written in the standard syntax $\lambda x.\ t$ (the function that maps x to t) but can also have multiple arguments $\lambda x\ y.\ t$, paired arguments $\lambda(x, y).\ t$, or dummy arguments $\lambda\_.\ t$. 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. In some places in the paper we have simplified formulas or code marginally to avoid distraction by syntactic or technical details, but in general we have stayed faithful to the sources.

1.2 Related Work

The basis of the work presented in this paper is our existing formalization of timed automata [7]. We are aware of one previous proof-assistant formalization of timed automata using PVS [8, 9]. This work has the basic decidability result using regions and claims to make some attempt to extend the formalization towards DBMs. Another line of work [10, 11] aims at modeling the class of p-automata [12] in Coq and proving properties of concrete p-automata within Coq. A similar approach was pursued with the help of Isabelle/HOL in the CClair project [13]. In contrast, our formalization [7] focuses on the foundations of timed automata model checking. In particular, it encompasses a formalization of the relevant DBM algorithms and the rather intricate developments towards the correctness proof for the approximation operation.

We are not aware of any previous formalizations or verified implementations of timed automata model checking. The first verification of a model checker we are aware of is by Sprenger for the modal μ-calculus in Coq [14]. Our important forerunner, however, is the CAVA project [15–17] by Esparza et al. It sets out for similar goals as we do but for *finite state LTL* model checking. A significant part of the refinement technology that we make use of was developed for this project, and it was the first project to demonstrate that verification of model checking can yield practical implementations. Compared to CAVA, our work offers several novelties: we target model checking of timed automata, which have an infinite state space; we use imperative data structures, which is crucial for efficient DBMs; finally, we implemented complex search algorithms with subsumption. Additionally, we operate on automata annotated with Uppaal bytecode, which has interesting ramifications: for the product construction, and because we need to ensure that the input actually defines a timed automaton.

2 Timed Automata and Model Checking

2.1 Transition Systems

We take a very simple view of transition systems: they are simply a relation $\rightarrow$ of type $'a \Rightarrow {}'a \Rightarrow bool$. We model *(finite) runs* as *inductive* lists, and *infinite runs* as *coinductive* streams. We write $a \rightarrow xs \rightarrow b$ to denote the $\rightarrow$-run from a to b using the intermediate states in the list xs, and $a \rightarrow^{ys}$ to denote the infinite $\rightarrow$-run starting in a and then continuing with states from the stream ys. Additionally, we define:

$$a \rightarrow^{+} b = (\exists xs.\ a \rightarrow xs \rightarrow b) \text{ and } a \rightarrow^{*} b = (a \rightarrow^{+} b \vee a = b)\ .$$

We define the five CTL properties that are supported by Uppaal, $\mathbf{A}\Diamond$, $\mathbf{A}\Box$, $\mathbf{E}\Diamond$, $\mathbf{E}\Box$, and $\dashrightarrow$, as properties of infinite runs[2] starting from a state. For instance,

$$\mathbf{A}\Diamond\ \phi\ x = (\forall xs.\ x \rightarrow^{xs} \Longrightarrow\ ev\ (holds\ \phi)\ (x \cdot xs))\ ,$$

and

$$\phi \dashrightarrow \psi = \mathbf{A}\Box\ (\lambda x.\ \phi\ x \Longrightarrow \mathbf{A}\Diamond\ \psi\ x)\ ,$$

[2] This is fairly standard in the literature [2, 3, 12, 18] but differs slightly from the implementation in Uppaal.

where *ev* specifies that a property on a stream eventually holds, and *holds* constrains *ev* to the current state instead of the remainder stream. It then is trivial to prove identities such as $\mathbf{E}\Box\ \phi\ x = (\neg\,\mathbf{A}\Diamond\ (\mathit{Not} \circ \phi)\ x)$.

2.2 Timed Automata

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 $\mathit{nat} \Rightarrow \mathit{real}$.

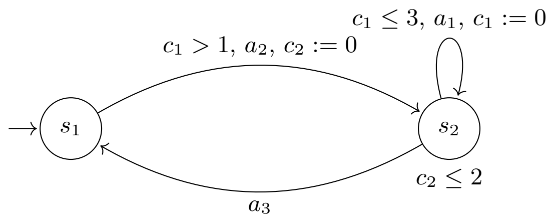


Fig. 1: Example of a timed automaton with two clocks.

Locations 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 $\sim\, \in \{<, \le, =, \ge, >\}$. We write $u \vdash cc$ 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 an assignment of clock constraints to locations (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 define two kinds of steps:

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

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

$$(l, u) \to_{A} (l', u') = (\exists d \ge 0.\ \exists a\ u''.\ (l, u) \to^{d} (l, u'') \wedge (l, u'') \to_{a} (l', u')).$$

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. We write $A, s_0 \models \phi$ if ϕ holds in state s_0 w.r.t. $\to_{A}$. Note that it is crucial to combine the two types of steps in order to reason about liveness. Consider the automaton from Fig. 1 and assume the two kinds of steps could be taken independently. Then the automaton has a run on which some predicate P holds everywhere if and only if $P\ s_1$ holds.

2.3 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 initial decidability result [1] partitioned the state space into a quotient of zones, the so-called regions, and showed that these yield a sound and complete abstraction[3]. However, practical model checking algorithms rather explore the state space in an *on-the-fly* manner, computing successors directly on zones, which are typically represented symbolically as Difference Bound Matrices (DBMs). DBMs are simply a matrix-form representation of clock constraints, which contain exactly one conjunct for each pair of clocks. To represent constraints on single clocks, an artificial **0**-clock is added, which is assumed to be assigned 0 in any valuation.

The delicate part of this method is that the number of reachable zones could still be infinite. Therefore, an over-approximation is applied to zones 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* [18]. The soundness argument for this method (due to over-approximation completeness is trivial), starts from the region construction and then introduces the notion of the *closure* of a zone, which is defined to be the union of all regions intersecting with a zone. It can be shown from the correctness of the region construction that closures yield a sound over-approximation of zones. Finally, one shows that the result of applying the over-approximation operator to zones is always contained in the closure, thus inheriting soundness from the soundness of closures. We have formalized this argument and all of the material summarized in this section in previous work [7]. It only covers the case of reachability, but we will demonstrate how to extend the soundness argument to liveness below.

3 A First Glance at the Model Checker

This section provides a first overview of our model checker, its construction, and the correctness theorem we proved. The input to our checker consists of a model, i.e. a network of Timed Automata, and a formula to be checked against the model. To achieve high compatibility with Uppaal, guards and updates can be formulated in Uppaal bytecode[4]. This intermediate representation is computed by Uppaal from the original C-style input before the actual model

[3] We use the same notions as in [7]. Soundness: for every abstract run, there is a concrete instantiation. Completeness: every concrete run can be abstracted.

[4] For the time being, the bytecode needs to be pre-processed slightly, mainly to rename textual identifiers to integers.

checking process is started. Given such an input, our tool will first determine whether the input is valid and lies in the supported fragment. This is achieved by a simple program analysis. As input formulae, our model checker accepts the same (T)CTL fragment that is supported by UPPAAL, but restricts formulae to not contain clocks. While this is not a principal limitation of our work, it reduced the complexity of our first prototype. If the input is invalid, our tool answers with "invalid input", else it determines whether

$$conv\ N, (init, s_0, u_0) \vDash_{max\_steps} \phi$$

holds for the all-zero valuation u_0 under the assumption that the automaton is deadlock-free[5], and answers with true/false. Here, N is the input automaton, *conv* converts all integer constants to reals (as the semantics are specified on reals), and ϕ is the input formula. The relation $\vDash_{max\_steps}$ is a variant of $\vDash$ lifted to networks of timed automata with shared state and Uppaal bytecode annotations. It is indexed with the maximum number of steps that any execution of a piece of Uppaal bytecode can use (i.e. *max_steps* is the *fuel* available to executions). The vector of start locations *init*, and the shared state s_0 (part of the input) describe the initial configuration.

The actual model checking proceeds in two steps. First, a product construction converts the network to a *single* timed automaton, expressed by HOL functions for the transition relation and the invariant assignment. Second, according to the formula, a model checking algorithm is run on the single automaton. We need three algorithms: a reachability checker for $\mathbf{E}\Diamond$ and $\mathbf{A}\Box$, a loop detection algorithm for $\mathbf{E}\Box$ and $\mathbf{A}\Diamond$, and a combination of both to check $\dashrightarrow$-properties. Note that the aforementioned HOL functions are simply *functional programs* that construct the product automaton's state and invariant assignments *on-the-fly*. The final correctness theorem we proved can be stated as follows:

$$
\begin{array}{l}
\{emp\}\\
\quad precond\_mc\ p\ m\ k\ max\_steps\ I\ T\ prog\ formula\ bounds\ P\ s_0\\
\{\lambda Some\ r \Rightarrow valid\_input\ p\ m\ max\_steps\ I\ T\ prog\ bounds\ P\ s_0\ na\ k \wedge\\
\quad\ (\neg\ deadlock\ (conv\ N)\ (init, s_0, u_0) \Longrightarrow\\
\quad\quad r = conv\ N, (init, s_0, u_0) \vDash_{max\_steps} formula)\\
\ \mid None \Rightarrow \neg\ valid\_input\ p\ m\ max\_steps\ I\ T\ prog\ bounds\ P\ s_0\ na\ k\}
\end{array}
$$

This Hoare triple states that the model checker terminates and produces the result *None* if the input is invalid. If the input is valid and deadlock free, it produces the result *Some r*, where *r* is the answer to the model checking problem.

4 Single Automaton Model Checking

In this section, we describe the route from the abstract semantics of timed automata to the implementation of an actual model checker. The next section will describe the construction of a single timed automaton from the Uppaal-model.

[5] Adding a check for deadlocked states to our algorithms would be conceptually simple but is left for future work.

4.1 Implementation Semantics

Although we have established that the DBM-based semantics from Section 2 can only explore finitely many zones, it is still "too infinite": the automaton and DBMs are described by real constants, and operations on DBMs are performed on infinitely many dimensions (i.e. clocks). Thus, we introduce an *implementation semantics*, in which automata are given by integer constants, and where the number of clocks is fixed. We prove equivalence of the semantics in two steps: first, we show that DBM operations need only be performed on the clocks that actually occur in the automaton; second, we show that all computations can be performed on integers, provided the initial state only contains integers.

For the former step, we simplify the operations under the assumptions that they maintain *canonicity* of DBMs. A DBM is canonical if it stores the tightest derivable constraint for each pair of clocks, i.e.

$$\mathit{canonical}\ M\ n = (\forall i\ j\ k.\ i \leq n \wedge j \leq n \wedge k \leq n \rightarrow M\ i\ k \leq M\ i\ j + M\ j\ k)\ .$$

During model checking, the Floyd-Warshall algorithm is used to turn a DBM into its canonical counterpart.

For the latter step, we use Isabelle's integrated parametricity prover [19] to semi-automatically transfer the operations from reals to integers.

As an example, Fig. 2 displays the refinement steps of the *up* operation, which computes the time successor of a zone Z, i.e. the set $\{u \oplus d \mid u \in Z \wedge d \geq 0\}$.

```
$\mathit{up}\ M = (\lambda i\ j.$
  if $i > 0$ then if $j = 0$ then $\infty$
  else $\min(M\ i\ 0 + M\ 0\ j)(M\ i\ j)$
  else $M\ i\ j)$
```

(a)

```
$\mathit{up}_1\ M = (\lambda i\ j.$
  if $i > 0 \wedge j = 0$
  then $\infty$
  else $M\ i\ j)$
```

(b)

```
$\mathit{up}_2\ M\ n = \mathit{fold}$
  $(\lambda i\ M.\ M((i, 0) := \infty))$
  $[1\,..{<}\,n+1]\ M$
```

(c)

```
$\mathit{up}_3\ M\ n = \mathit{imp\_for}'\ 1\ (n+1)$
  $(\lambda i\ M.\ \mathit{mtx\_set}\ (n+1)\ M\ (i, 0)\ \infty)$
  $M$
```

(d)

Fig. 2: Refinement stages of the *up* operation for computing time successors.

In the step from *up* to up_1, the assumption that the input DBM is canonical is introduced. In up_2, which is the version used in the implementation semantics, the operation is constrained to clocks 1 to n. Finally, in up_3, the matrices are implemented as arrays and the fold is implemented as a foreach loop.

At this point, a naive exploration of the transitive closure of the implementation semantics would already yield a simple but inefficient model checker. The rest of this section outlines the derivation of a more elaborate implementation that is close to what can be found in UPPAAL.

4.2 Semantic Refinement of Successor Computation

We further refine the implementation semantics to add two optimizations to the computation of successor DBMs: to canonicalize DBMs it is sometimes sufficient to only "repair" single rows or columns instead of running the full Floyd-Warshall algorithm; moreover, we can terminate the computation early whenever we discover a DBM that represents an empty zone (as it will remain empty). Both arguments are again carried out on the semantic level.

4.3 Abstraction of Transition Systems with Closures

Recall that the correctness of the reachability analysis on the abstract zone graph in Section 2 was obtained arguing that the region closure of zones forms a sound over-approximation of zones, which in turn is larger than the abstract zone graph. We want to reuse the same kind of argument to also argue that there exists a cycle in the abstract zone graph if and only if there is a cycle in the automaton's transition system. This proof is carried out in a general abstract framework for transition systems and their abstractions.

We consider a concrete step relation $\rightarrow_C$ over type $'a$ and what is supposed to be its simulation, a step relation $\rightarrow_{A_1}$ over type $'a\ set$. We say that $\rightarrow_{A_1}$ is *post-stable* [20] if $S \rightarrow_{A_1} T$ implies

$$\forall s' \in T.\ \exists s \in\ S.\ s \rightarrow_C s'\ ,$$

and that $\rightarrow_{A_1}$ is *pre-stable* [20] if $S \rightarrow_{A_1} T$ implies

$$\forall s \in S.\ \exists s' \in\ T.\ s \rightarrow_C s'\ .$$

In the timed automata setting, for instance, the simulation graph is post-stable and the region graph is pre-stable.

Lemma 1. *If $\rightarrow_{A_1}$ is post-stable and we have $a \rightarrow_{A_1} as \rightarrow_{A_1} a$ with a finite and non-empty, then there exist xs and $x \in a$ such that $x \rightarrow_C xs \rightarrow_C x$.*

Proof. Let $x \rightarrow y = (\exists xs.\ x \rightarrow_C xs \rightarrow_C y)$. As $\rightarrow_{A_1}$ is post-stable, every a has an ingoing $\rightarrow$-edge. Because a is finite we can thus find an $\rightarrow$-cycle in a, and obtain the claim.

Lemma 2. *If $\rightarrow_{A_1}$ is pre-stable and we have $a \rightarrow_{A_1} as \rightarrow_{A_1} a$ and $x \in a$, then there exist xs such that $x \rightarrow_C^{xs}$ and xs passes through a infinitely often.*

Proof. By coinduction. From pre-stability we can find $x_1 \in a$ such that $x \rightarrow_C^{+} x_1$, from x_1 we find $x_2 \in a$ such that $x_1 \rightarrow_C^{+} x_2$, and so forth.

We can now consider doubly-layered abstractions as in the case for regions and zones. That is, we add a second simulation $\rightarrow_{A_2}$ and two predicates P_1 and P_2 that designate valid states of the first and second simulation layer, respectively. Then we define the *closure* $\mathcal{C}$ of a state of the second layer as

$$\mathcal{C}\ a = \{x \mid P_1\ x \wedge a \cap x \neq \emptyset\} \text{ and } a \rightarrow_{\mathcal{C}} b = (\exists x\ y.\ a = \mathcal{C}\ x \wedge b = \mathcal{C}\ y \wedge x \rightarrow_{A_2} y)\ .$$

We assume that $\rightarrow_{A_1}$ is pre-stable w.r.t. $\rightarrow_C$ and that $\rightarrow_{\mathcal{C}}$ is post-stable w.r.t. $\rightarrow_{A_1}$. Along with some side conditions on P_1 and P_2[6] we can prove:

Theorem 1. *If $a_0 \rightarrow_{A_2} as \rightarrow_{A_2} a \rightarrow_{A_2} bs \rightarrow_{A_2} a$ and $P_2\ a$, then there exist $x \in \bigcup(\mathcal{C}\ a_0)$ and xs such that $x \rightarrow_C^{xs}$ and xs passes through $\bigcup(\mathcal{C}\ a)$ infinitely often.*

Proof. We first apply $\mathcal{C}$ to the second layer states and get a path of the form: $\mathcal{C}\ a_0 \rightarrow_{\mathcal{C}} as' \rightarrow_{\mathcal{C}} \mathcal{C}\ a \rightarrow_{\mathcal{C}} bs' \rightarrow_{\mathcal{C}} \mathcal{C}\ a$ for some as' and bs'. From Lemma 1 and post-stability, we obtain a path of the form $a_{0_1} \rightarrow_{A_1} as_1 \rightarrow_{A_1} a_1 \rightarrow_{A_1} bs_1 \rightarrow_{A_1} a_1$ with $a_{0_1} \in \mathcal{C}\ a_0$ and $a_1 \in \mathcal{C}\ a$. By applying Lemma 2 and pre-stability, we obtain the desired result.

This is the main theorem that allows us to run cycle detection on the abstract zone graph during model checking: the other direction is trivial, and the theorem can be directly instantiated for regions and (abstracted) zones. There is a slight subtlety here since we only guarantee $x \in \bigcup(\mathcal{C}\ a_0)$. However we typically have $\mathcal{C}\ a_0 = a_0$, as all clocks are initially set to zero.

4.4 Implementation of Search Algorithms

We first implement the three main model checking algorithms abstractly in the nondeterminism monad provided by the IRF. On this abstraction level, we can use such abstract notions as sets and specify the algorithm for an arbitrary (finite) transition system $\rightarrow$. We only showcase the implementation of our cyclicity checker (used for $\mathbf{A}\Diamond$ and $\mathbf{E}\Box$). The techniques used for the other algorithms are similar. The code for our cyclicity checker is displayed in Listing 1.1.

```
$dfs\ P$ = do {
  $(P, ST, r) \leftarrow$ rec$_T$ $(\lambda dfs\ (P, ST, v)$ .
    do {
      if $\exists v' \in set\ ST.\ v' \preceq v$ then return $(P, ST, True)$
      else do {
        if $\exists v' \in P.\ v \preceq v'$ then return $(P, ST, False)$
        else do {
          let $ST = v \cdot ST$ ;
          $(P, ST', r) \leftarrow$
            foreach $\{v' \mid v \rightarrow v'\}$ $(\lambda(\_, \_, b).\ \neg\, b)$
              $(\lambda v'\ (P, ST, \_).\ dfs\ (P, ST, v'))$
              $(P, ST, False)$ ;
          assert $(ST' = ST)$ ;
          return $(insert\ v\ P, tl\ ST', r)$
        }
      }
    }) $(P, [], a_0)$ ;
  return $(r, P)$}
```

Listing 1.1: Cyclicity Checker

[6] P_1 states are distinct and there are only finitely many of them. For every P_2 state, there is an overlapping P_1 state.

We claim that this closely resembles the pseudo-code found, e.g., in [21]. The algorithm takes a passed set, and produces a new passed set in addition to the answer. This can be used in the algorithm for checking $\dashrightarrow$-properties. The crux of the algorithm is the use of the subsumption operator $\preceq$, to check whether smaller states are already subsumed by larger states that we may have discovered earlier (for timed automata, this would correspond to set inclusion on zones). We assume that $\preceq$ is a pre-order and monotone w.r.t. $\rightarrow$. Then, using the IRF's verification condition generator, we prove:

$$\begin{gathered} \mathit{dfs}\ P \leq \mathsf{SPEC}\ (\lambda(r, P').\ (r \Longrightarrow (\exists x.\ a_0 \rightarrow^* x \wedge x \rightarrow^+ x)) \\ \wedge(\neg\, r \Longrightarrow \neg\,(\exists x.\ a_0 \rightarrow^* x \wedge x \rightarrow^+ x) \wedge \mathit{liveness\_compatible}\ P')) \\ \text{if } \mathit{liveness\_compatible}\ P\ . \end{gathered}$$

The invariant we maintain for the passed set P is encoded in the predicate *liveness_compatible* P. We say that a state x is covered by P if there exists $x' \in P$ such that $x \preceq x'$. Then, informally, *liveness_compatible* P states that the successors of every node that is covered by P are also covered, and that there is no cycle through nodes that are covered by P. After specifying the correct loop invariant (using *liveness_compatible* as the main insight) and the termination relation, together with some key lemmas about the invariant, the verification conditions can be discharged nearly automatically.

In subsequent steps, we gradually refine this implementation to use more efficient data structures. The final version uses a function to compute a list of successors, and is able to *index* the passed set and the stack according to a key function on states (this corresponds to the location part of states in the abstract zone graph). The refinement theorem can be stated as:

$$\mathit{dfs\_map}\ P \leq \Downarrow (\mathit{Id} \times_r \mathit{map\_set\_rel})\ (\mathit{dfs}\ P') \text{ if } (P, P') \in \mathit{map\_set\_rel}\ .$$

That is, *dfs_map* is a refinement of *dfs*, where the passed set is data-refined w.r.t. the relation *map_set_rel*. This relation describes the implementation of passed sets indexed by keys.

These refinement steps are conducted inside the nondeterminism monad of the IRF. The final step leads into the heap-monad of Imperative HOL [22], which supports imperative data structures. Here, the *Sepref* tool [6] replaces functional by imperative data structures and generates a refinement theorem automatically.

Maps are implemented via hash tables, which poses a challenge for the implementation as the maps contain objects stored on the heap. This was not supported by the existing implementation in the Imperative Collections Framework, due to sharing issues: when retrieving a value from the map, we cannot obtain ownership of the value while the map also retains ownership. This is even true if the value is read-only. One way to solve this problem would be to extend the separation logic that underlies the IRF to fractional permissions or read-only permissions. Our solution, however, is more ad-hoc: we simply restrict the operations that we perform on the hash map to insertions and an *extract* operation, which deletes a key-value pair from the map and returns the value (i.e. it combines lookup and delete). To define the map implementation, we use a trick similar to Chargueraud's ideas from [23]: we use a *heap assertion* that first connects an abstract map m with an intermediate map m_h of pointers to the elements, and then implements the map of pointers by a regular hash map m_i. Formally, the assertion is defined as:

$$\mathit{hms\_assn}\ A\ m\ m_i = (\exists_A m_h.\ \mathit{is\_map}\ m_h\ m_i * \mathit{map\_assn}\ A\ m\ m_h)\ .$$

Here *is_map* is the assertion for an existing map implementation from the Imperative Collections Framework (which cannot store heap objects, but supports pointers), and $\mathit{map\_assn}\ A\ m\ m_h$ connects a map of abstract values with a map of pointers, where the relation between abstract values and pointed-to objects is defined by A.

Then, the final implementation is produced by proving that all map-related operations in *dfs_map* can be replaced by insert and extract operations, and letting Sepref synthesize the imperative variant, making use of our new hash map implementation. The key theorem on the final implementation is the following Hoare triple:

$$\begin{array}{l}\{\mathit{emp}\}\\ \quad \mathit{dfs\_map\_impl'}\ \mathit{succsi}\ a_0 i\ \mathit{Lei}\ \mathit{keyi}\ \mathit{copyi}\\ \{\lambda r.\ r = (\exists x.\ a_0 \rightarrow^* x \wedge x \rightarrow^+ x)\}\end{array}$$

It is expressed in a locale (Isabelle's module system) that assumes that $a_0 i$, *succsi*, etc., are the correct imperative implementations of a_0, the successor function, and so forth. Versions of the search algorithm for concrete transition systems are obtained by simply instantiating the locale with the operations of the transition system and their implementations.

4.5 Imperative Implementations of Model Checking Operations

Recall the refinement of the *up* operation (Fig. 2). It is crucial that up_2 is expressed as a fold-operation with explicit updates, as only then the IRF can extract an efficient imperative version with destructive updates and a foreach loop. The imperative implementation up_3 is, again, synthesized by the Sepref tool. As can be witnessed for up_3, the pattern $\mathit{fold}\ f\ [1\,..\!<n+1]$ is turned into a foreach loop. Technically, this is achieved by a set of rewrite rules that are applied automatically by the Sepref tool at the end of the synthesis process. The only hurdle for this kind of synthesis is that the dimension of DBMs needs to become a parameter of the refinement relations. For n clocks, we define

$$\mathit{mtx\_assn} = \mathit{asmtx\_assn}\ (n+1)\ \mathit{id\_assn}\ .$$

This specifies that our DBMs are implemented by square-arrays of dimension $n+1$, and their elements are refined by the identity relation.

The refinement theorem for up_3 is proved automatically by the Sepref tool:

$$(up_3, up_2) \in [\lambda(\_, i).\ i \leq n]\ \mathit{mtx\_assn}^d * \mathit{nat\_assn}^k \rightarrow \mathit{mtx\_assn}\ .$$

This theorem states that, if the specified dimension is in bounds, up_3 refines up_2. The $\cdot^d$ annotation indicates that the operation is allowed to overwrite (destroy) the input matrix on the heap. Symmetrically, the $\cdot^k$ annotation means that the second parameter is not overwritten (kept).

4.6 Code Extraction

Finally, Isabelle/HOL's code generator [24] is used to extract imperative Standard ML code from the Isabelle specifications generated by Sepref. Code generation involves some optimizations and rewritings that are carried out as refinement steps and proved correct, followed by pretty printing from the functional fragment of HOL and the heap monad to Standard ML.

5 From UPPAAL-style Semantics to a Single Automaton

5.1 UPPAAL-style semantics

Due to the lack of documentation on the UPPAAL intermediate format, we define an approximation of this assembler-like language by reverse engineering. This is sufficient to check typical benchmarks, and gives a clearly defined semantics to the fragment that we cover. The language is defined as a simple data type *instr*. A *step* function of type *instr* $\Rightarrow$ *state* $\Rightarrow$ *state option* computes the successor state after executing an instruction, or fails. A state consists of an instruction pointer, the stack, the state of the shared integer variables, the state of the comparison flag, and a list of clocks that have been marked for reset. Using a fuel parameter, we execute programs by repeatedly applying the *step* function until we either reach a halt instruction, fail, or run out of fuel, which we also regard as a failed execution.

A special instruction *CEXP* is used to check whether an atomic clock constraint holds for a given valuation u. However, this instruction cannot simply be executed during model checking as it would need to work on zones instead of valuations. Unconstrained use of the *CEXP* instruction would allow for disjunctions of clock constraints on edges, which is not part of the standard timed automata formalism. Thus, in the same way as UPPAAL, we restrict the valid input programs to those that only yield conjunctions of clock constraints on edges. We then replace every *CEXP* instruction by a special meta instruction that sets the comparison flag to true. This amounts to enforcing a program execution where the clock constraint, which is expressed by a piece of bytecode, holds for a valuation. Edges are annotated with the conjunction of the atomic clock constraints encountered during execution. In the current version of our tool, we separate concerns for locations by using a state predicate, which is not allowed to use *CEXP* instructions, and a separate clock constraint. The two could be merged by using the same approach as for edges.

5.2 Program Analysis

As stated in the last section, we need to ensure that successful program executions can only induce conjunctive clock constraints. That is, we need to ensure that program executions can only be successful when all *CEXP* instructions that are encountered during execution evaluate to true. To this end, we use a naive analysis, which recognizes a subclass of these programs that is sufficiently large to cover common timed automata benchmarks. This analysis tries to identify what we call *conjunction blocks*. A conjunction block reaching from addresses pc_s to pc_t ends with a *halt* instruction at pc_t, starts with a *CEXP* instruction at pc_s and then is extended to pc_t by repeatedly using one of the following two patterns:

– a *copy* instruction to push the flag on the stack, followed by *CEXP* and an *and* instruction;
– a *copy* instruction, followed by a *jump-on-zero* instruction with pc_t as the destination, followed by *CEXP* and an *and* instruction.

We simultaneously show the two key properties of conjunction blocks via induction: if there is a conjunction block from pc_s to pc_t, then any successful execution starting from pc_s ends in pc_t, and every *CEXP* instruction that is encountered has to evaluate to true. Given a start address pc_s, the whole analysis works by computing an approximation of the set of possible addresses that can be reached from pc_s, say S, and then checking whether

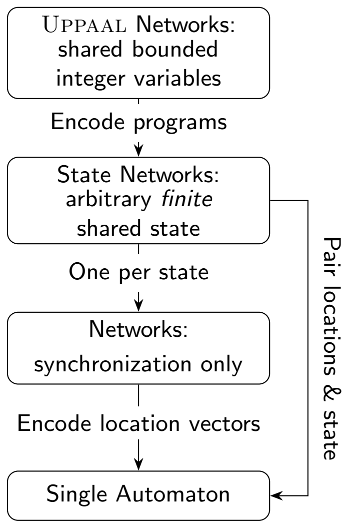
$$\mathit{Min}\ \{pc \mid pc \in S \wedge (\exists ac.\ P_{pc} = \mathit{CEXP}\ ac)\} \text{ to } \mathit{Max}\ S$$

is a conjunction block, where P_{pc} is the program instruction at address pc. A major limitation of this analysis is that it cannot approximate the reachable set for *call* and *return* instructions, so we are not able to handle inputs that are compiled from Uppaal programs with sub-routines. However, as the main objective of our work is not program analysis, we consider the current naive analysis sufficient to demonstrate the general viability of our approach.

5.3 Product Construction

The general shape of our product construction is outlined in Fig. 3. The first stage of the construction encodes the bytecode annotations as guards and updates on the shared state. The subsequent stage constructs a network of automata for *each shared state* by essentially filtering the transitions that are valid for a given state. For a simple network, the product can be constructed in the obvious way. However, this is only used in the correctness proof of the final step, which directly constructs a single automaton by pairing the location vector and the state.

The result of this construction is a highly contrived description of the single automaton. To obtain an efficiently executable version of this description, we specify an alternative functional implementation and prove the equivalence of the two.



UPPAAL Networks: shared bounded integer variables
Encode programs
State Networks: arbitrary *finite* shared state
One per state
Networks: synchronization only
Encode location vectors
Single Automaton
Pair locations & state

Fig. 3: Outline of the product construction.

6 Experimental Evaluation

We conducted experiments on some standard benchmark models for timed automata: a variant of Fischer's mutual-exclusion protocol, the FDDI token ring protocol, and the CSMA/CD protocol used in Ethernet networks. We tested one reachability and one liveness property for each model: $\mathbf{E}\Diamond(c > 1)$ and $P_1.b \dashrightarrow P_1.c$ for Fischer's protocol; $\mathbf{E}\Diamond(\neg P_1.\mathit{idle} \land \neg P_2.\mathit{idle})$ and $\mathit{true} \dashrightarrow \mathit{z\_async_1}$ for FDDI; and $\mathbf{E}\Diamond(P_1.\mathit{abort} \land P_2.\mathit{send})$, and $\mathit{collision} \dashrightarrow \mathit{active}$ for CSMA/CD. We compare (c.f. Table 1) our tool against UPPAAL configured with two different approximation operators: difference (UPPAAL$_1$) and location-based (UPPAAL$_2$) extrapolation. We give the computation time in seconds and the number of explored states, as reported by our tool and UPPAAL[7]. Since the number of explored states differs significantly, we also calculated *throughput*, i.e. the number of explored states per second. The ratio of UPPAAL's throughput and our tool's throughput is given in the column *TR*. We specify the problem size as the number of automata in the network.

The results indicate that our tool's throughput is around one order of magnitude lower than UPPAAL's. Encouragingly, the gap seems to decrease for larger models. However, for larger problem sizes of some models, we also start to run out of memory because our tool is not tuned towards space consumption. We do not have a convincing explanation for the difference in states explored by our tool and UPPAAL — particularly, because our tool already implements location-based extrapolation. Nevertheless, we conclude that the performance offered by our tool is reasonable for a reference implementation against which other tools can be validated: we can check medium sized instances of common benchmark

[7] UPPAAL comes with a note suggesting that these numbers might be wrong for liveness properties.

Table 1: Experimental results on a set of standard benchmarks.

				Our Tool		Uppaal$_1$			Uppaal$_2$		
Model	Prop	SAT	Size	time	#states	time	#states	TR	time	#states	TR
Fischer	R	N	5	6,61	38578	0,31	12363	6,83	0,04	3739	16,02
	L	Y	5	7,52	42439	0,31	20340	11,8	0,04	8149	40,1
		Y	6	485,9	697612	42,85	249295	4,1	1,53	67325	30,7
FDDI	R	N	8	16,04	6720	0,34	5416	37,6	0,31	5416	42,0
		N	10	142,8	29759	6,63	24210	17,5	6,44	24120	18,0
	L	Y	6	2,58	2083	0,05	2439	61,7	0,04	2439	68,7
		Y	7	6,50	3737	0,15	4944	57,0	0,14	4944	62,3
CSMA/CD	R	N	5	4,48	9959	0,03	2704	45,3	0,03	2769	40,6
		N	6	71,70	81463	1,70	17613	9,2	1,79	17939	8,8
	L	Y	5	4,93	11526	0,04	3802	42,4	0,04	3867	42,4
		Y	6	76,83	96207	1,78	23128	10,4	1,86	12603	10,1

models, which should be sufficient to scrutinize the functionality of a model checker.

7 Conclusion

We have derived an efficiently executable and formally verified model checker for timed automata. Starting from an abstract formalization of timed automata, we first reduced the problem to model checking of a single automaton, and then used stepwise refinement techniques to gradually replace abstract mathematical notions by efficient algorithms and data structures. Some of the verified algorithms and data structures, e.g. search with subsumption and Difference Bound Matrices, are interesting in their own right. Our experiments demonstrate that our tool's performance is suitable for validating other model checkers against it on medium sized instances of classic benchmark models. Using a simple program analysis, we can cover a subset of the Uppaal bytecode that is sufficient to accept common models as an input.

Following the construction we expounded above, our checker can be improved on two different axes: advanced modeling feature such as broadcast channels or committed locations can be enabled by elaborating the product construction; using the refinement techniques that we demonstrated above, further improvements of the model checking algorithms can achieve better performance.

An alternative approach to tackle performance problems is to resort to certification of model checking results. For the simple CTL properties that are supported by our tool and Uppaal, passed sets could be used as the certificates and the model checking algorithms could be reused for certificate checking. As the model checking algorithms for timed automata make use of subsumption, passed sets can contain significantly less states than the total number of states explored during model checking. We plan on exploring this avenue in the future.

Data Availability Statement

The datasets generated during and analyzed during the current study are available in the figshare repository [25]: https://doi.org/10.6084/m9.figshare.5917363.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2) (1994) 183–235
2. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Lectures on Concurrency and Petri Nets: Advances in Petri Nets. Volume 3908 of LNCS., Springer (2004) 87–124
3. Bouyer, P.: Untameable timed automata! In: STACS 2013, Proceedings. Volume 2607 of LNCS., Springer (2003) 620–631
4. Nipkow, T., Lawrence C. Paulson, Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
5. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Proc. of ITP. Volume 7406 of LNCS., Springer (2012) 166–182
6. Lammich, P.: Refinement to Imperative/HOL. In Urban, C., Zhang, X., eds.: ITP 2015, Proceedings. Volume 9236 of LNCS., Springer (2015) 253–269
7. Wimmer, S.: Formalized timed automata. In Blanchette, J.C., Merz, S., eds.: ITP 2016, Proceedings. Volume 9807 of LNCS., Springer (2016) 425–440
8. 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. (2006) 107–112
9. Xu, Q., Miao, H.: Manipulating clocks in timed automata using PVS. In: Proc. of SNPD'09. (2009) 555–560
10. Paulin-Mohring, C.: Modelisation of timed automata in Coq. In: Proc. of STACS'01. LNCS 2215 (2001) 298–315
11. Garnacho, M., Bodeveix, J., Filali-Amine, M.: A mechanized semantic framework for real-time systems. In: Proc. of FORMATS'13. LNCS 8053 (2013) 106–120
12. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Proc. of the Twenty-Fifth Annual ACM Symposium on Theory of Computing. (1993) 592–601
13. Castéran, P., Rouillard, D.: Towards a generic tool for reasoning about labeled transition systems. In: TPHOLs 2001: Supplemental Proceedings. (2001) http://www.informatics.ed.ac.uk/publications/report/0046.html.
14. Sprenger, C.: A verified model checker for the modal μ-calculus in Coq. In: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems. TACAS '98, London, UK, Springer (1998) 167–183
15. 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. Volume 8044 of LNCS., Springer (2013) 463–478
16. Neumann, R.: Using promela in a fully verified executable ltl model checker. In Giannakopoulou, D., Kroening, D., eds.: Verified Software: Theories, Tools and Experiments, Springer (2014) 105–114
17. Brunner, J., Lammich, P.: Formal verification of an executable ltl model checker with partial order reduction. Journal of Automated Reasoning **60**(1) (Jan 2018) 3–21

18. 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 2016. Volume 65 of LIPIcs., Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2016) 48:1–48:14
19. Huffman, B., Kuncar, O.: Lifting and transfer: A modular design for quotients in Isabelle/HOL. In: CPP. (2013) 131–146
20. Bouajjani, A., Tripakis, S., Yovine, S.: On-the-fly symbolic model checking for real-time systems. In: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA. (1997) 25–34
21. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Beyond liveness: Efficient parameter synthesis for time bounded liveness. In: Proceedings of the Third International Conference on Formal Modeling and Analysis of Timed Systems. FORMATS'05, Springer (2005) 81–94
22. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: TPHOL. Volume 5170 of LNCS., Springer (2008) 134–149
23. Charguéraud, A.: Higher-order representation predicates in separation logic. In: CPP. (2016) 3–14
24. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: FLOPS 2010. LNCS, Springer (2010)
25. Wimmer, S., Lammich, P.: Verified model checking of timed automata – artifact (2018)