Compositional Abstraction Refinement for Timed Systems

Fei He†‡, He Zhu†‡, William N. N. Hung¶, Xiaoyu Song¶ and Ming Gu†‡
† School of Software, Tsinghua University, Beijing, China
‡ Tsinghua National Laboratory for Information Science and Technology
¶ Key Laboratory for Information System Security, MOE, China
¶¶ Synopsys Inc., Mountain View, California, USA
§ Department of ECE, Portland State University, Oregon, USA

Abstract—Model checking suffers from the state explosion problem. Compositional abstraction and abstraction refinement have been investigated in many areas to address this problem. This paper considers the compositional model checking for timed systems. We present an automated approach which combines compositional abstraction and counter-example guided abstraction refinement (CEGAR). The proposed approach exploits the semantics of a timed automaton to procure its over-approximative abstraction. Any safety property which holds on the abstraction is guaranteed to hold on the concrete model. In the case of a spurious counter-example, our proposed approach refines and strengthens the abstraction in a component-wise method. We implemented our method with the model checking tool Uppaal. Experimental results show promising improvements.

I. INTRODUCTION

Model checking suffers from the state explosion problem. The situation is even worse in real-time systems. Real-time systems can be modeled as a parallel composition of timed automata [1]. The reachability verification problem for timed automaton is usually based on the zone graph [2] which, in worst case, is a region graph [1]. However, the number of regions in a region graph for each location of the timed automaton is exponential to the number of clocks (as well as the maximal constants appearing in the guards and invariants).

Various approaches ranging from compositional abstraction to counter-example guided abstraction refinement (CEGAR) have been investigated in many areas to address the state explosion problem. Even in the verification of timed systems, some researchers have already made use of these techniques and achieved some encouraging improvements [3]–[6]. However, their efforts concentrate on certain aspects of the problem, without specific method to solve it as a whole. Existing methods for timed systems either are not fully automated [3] or do not provide explicit compositional support for concurrency [4]–[6]. On the contrary, our method integrates the two powerful methods to overcome the state explosion problem.

In this paper, we investigate compositional abstraction refinement for timed systems. An automated approach which combines compositional abstraction and counter-example guided abstraction refinement is presented. To verify a network of timed automata \( A_1, \ldots, A_n \) against a safety property, we first try to get an abstraction \( A_i' \) for each automaton \( A_i \) by eliminating its clock variables and merging some locations. This abstraction exploits the semantics of timed transition systems [3] and is conservative. As a consequence, the safety property that holds on the abstraction is guaranteed to hold on the concrete system. With the above facts, we are able to perform model checking on the composition of the abstracted automata \( A_1', \ldots, A_n' \) instead of the original automata. The abstracted automata are usually simpler than the original ones. In such a way, we avoid computing a large composition of the original model, thus alleviating the state explosion.

We develop a component-wise counter-example validation method without constructing the full state space of the concrete model when the model checker reports a counter-example. If the counter-example is validated, the algorithm safely concludes that the property does not hold on the concrete model. Otherwise, the proposed approach refines the model in a component specific approach to strengthen the abstractions and continues to verify the strengthened ones. This process is repeated until the property is verified or the counter-example indicates a real bug.

A. Related Work

Formalization of the abstraction technique first appeared in [7]. Conservative abstraction which preserves safety property was introduced in [8] [9]. Counter-example guided abstraction refinement [10] [11] is an iterative scheme which starts from a coarse abstraction and then iteratively strengthens the abstraction using spurious abstract counter-examples until the property is established or a concrete counter-example is found. Compositionality has been investigated by process algebra [12] [13]. On the other hand, compositionality in conjunction with abstraction refinement in timed and untimed systems has been studied both theoretically and practically in many research areas. The work in [14] [15] proposed an automated compositional CEGAR framework which differs in the communication method. In [14], they proposed a data guided (communication through shared variables) abstraction while the authors in [15] developed an action guided (communication through message-passing event) abstraction. A similar...
We present a compositional framework for verifying and a set of non-negative real-valued clock variables. Let \( \phi \) be a valuation in \( V \) to be a valuation in \( n \) or to range over be a set of constraints in which each constraint \( \phi \) \( v \) \( C \) \( R \) to be a valuation in \( n \) or to range over be a set of constraints in which each constraint \( \phi \) \( v \) \( C \) \( R \).

We implement our method and incorporate it with a model checker Uppaal. Experimental results show promising improvements.

II. PRELIMINARIES

In this section, we define the syntax and semantics of Timed Automaton. Some notations are borrowed from [1], [3].

Let \( C \) be a set of non-negative real-valued clock variables. For each \( p \in C \), its domain is \( \mathbb{R}_{\geq 0} \). Let \( X \) be a set of integer variables. Each \( x \in X \) is a typed variable defined over a finite domain of values \( D_x \). We write \( V = X \cup C \) for the universal set of typed variables including both integer variables and clock variables.

A valuation \( s: V \rightarrow D_V \) maps each variable \( u \in V \) to a certain value in its domain \( D_u \). We write \( Val(V) \) for the set of valuations for \( V \). For a valuation \( v \in Val(V) \) and a duration \( d \in \mathbb{R}_{\geq 0} \), we define \( v \oplus d \) to be a valuation in \( V \) that increments all the clock variables by \( d \) but leaves the integer variables unchanged. Given a valuation \( v \in Val(V) \), we denote \( v[u \mapsto n] \) a new valuation which maps \( u \in V \) to \( n \in D_u \), and other variables to the same values as in \( v \).

Let \( \varphi(C, X) \) be a set of constraints in which each constraint is a conjunction of atomic formulas in the form: \( p \sim n \) or \( p - q \sim n \) or \( x \sim n \), where \( p, q \in C \), \( x \in X \), \( \sim \in \{<,\leq,>,\geq,=\} \) and \( n \) is a natural number. We use \( ClockVar(\varphi(C, X)) \) to denote the set of clock variables appearing in \( \varphi(C, X) \). We use \( v \models \varphi(C, X) \) to denote that the constraint \( \varphi(C, X) \) holds for the valuation \( v \in Val(V) \).

Let \( C_H \) be a set of channels and \( c_h \) range over \( C_H \). The set of actions is defined as \( \varepsilon \triangleq \{c_h!, c_h? | c_h \in C_H \} \). Actions of the form \( c_h \) are called input actions, while actions of the form \( c_h? \) are called output actions. We use \( \tau \) to denote a special internal action which need not be synchronized.

A. Timed Transition Systems

Definition 1: A Labeled Transition System (LTS) is a tuple \( \langle S, s_0, Act, R \rangle \), where \( S \) is a set of states, \( s_0 \in S \) is the initial state, \( Act \) is a finite set of actions, and \( R \subseteq S \times Act \times S \) is the set of transitions.

We write \( s \xrightarrow{a} t \) if \( (s, a, t) \in R \). A path of an LTS is a sequence \( s_0, a_0, s_1, \ldots \), where each \( s_i \in S \) is a state and each \( a_i \in Act \) is an event and \( R(s_i, a_i, s_{i+1}) \) holds.

Definition 2: A Timed Transition System (TTS) \( T \) is a tuple \( \langle E, H, S, s_0, Act, R \rangle \), where

- \( E \) is a set of shared variables which belong to \( T \) and its environment; \( H \) is the set of internal variables which belong to \( T \) only. The set of all variables in \( T \) are denoted as \( V = E \cup H \);
- \( S \subseteq Val(V) \) is a set of states. Each \( s \in S \) is a valuation of the variables in \( V \);
- \( s_0 \in S \) is the initial state of \( T \);
- \( Act \subseteq \varepsilon \cup \mathbb{R}_{\geq 0} \) is a finite set (alphabet) of discrete actions (events) together with a set of time-passage actions (duration);
- \( R \) is a set of transitions over \( S \times (Act \cup \{\tau\}) \times S \). There exist both action transitions \( s \xrightarrow{a} s' \) (\( a \in Act \)) and timed transitions (time durations) \( s \xrightarrow{\tau} s' \in R \).

Given a TTS \( T = \langle E, H, S, s_0, Act, R \rangle \), its corresponding LTS is \( \langle S, s_0, Act, R \rangle \). In the following, we write \( LTS(T) \) for the underlying LTS of \( T \).

Definition 3: Two TTSs \( T_1 = \langle E_1, H_1, S_1, s_0, Act_1, R_1 \rangle \) and \( T_2 = \langle E_2, H_2, S_2, s_0, Act_2, R_2 \rangle \) are comparable if they have the same set of shared variables, i.e., \( E_1 = E_2 \). Two states \( s \in S_1 \) and \( t \in S_2 \) are compatible, denoted as \( s \times t \), if \( s(v) = t(v) \) for all variables \( v \in E_1 \cup E_2 \). Two TTSs \( T_1 \) and \( T_2 \) are compatible if \( H_1 \cap V_2 = H_2 \cap V_1 = \varepsilon \) and \( s_0 \times s_0 \).

We use \( s[t] \) to denote the update execution which substitutes the value of shared variables (in \( E_1 \cup E_2 \)) for \( s \) for that value of \( t \).

Definition 4: Given two compatible TTSs \( T_1 = \langle E_1, H_1, S_1, s_0, Act_1, R_1 \rangle \), and \( T_2 = \langle E_2, H_2, S_2, s_0, Act_2, R_2 \rangle \), the parallel composition \( T_1 || T_2 \) is the tuple \( \langle E, H, S, s_0, Act, R \rangle \) where \( E = E_1 \cup E_2 \), \( H = H_1 \cup H_2 \), \( S = \{ s || t | s \in S_1 \land t \in S_2 \land s \times t \} \), \( Act = Act_1 \cup Act_2 \), \( s_0 = s_0 || s_0 \), and \( R \) is defined by the following rules:

- if \( s \xrightarrow{\tau} s' \) \( e \in \varepsilon \) then \( s[t] \xrightarrow{\tau} s'[t] \)
- if \( s \xrightarrow{\tau} s' \) then \( s[t] \xrightarrow{\tau} s'[t] \)
- if \( s \xrightarrow{\tau} t[|s'] \) while \( c, c'? \in \varepsilon \) and \( i \neq j \) then \( s[t] \xrightarrow{\tau} t'[t] \)
- if \( s[d] \) then \( s[t] \xrightarrow{\tau} s'[t] \) where \( d \in \mathbb{R}_{\geq 0} \), then \( s[t \xrightarrow{\tau} s'[t] \)

B. Timed Automaton

Definition 5: A Timed Automaton (TA) \( A \) is a tuple \( \langle E, Y, C, L, l_0, Act, I, R \rangle \), where
• $E$ and $Y$ are disjoint sets of shared and internal integer variables, respectively. We write $X = E \cup Y$ to denote all the integer variables;
• $C$ is the set of clock variables. All the variables in $A$ are denoted as $V = E \cup Y \cup C$ (which obviously equals $X \cup C$);
• $L$ is a finite set of locations;
• $l_0 \in L$ is the initial location;
• $Act \subseteq \varepsilon$ is a finite set (alphabet) of actions (events);
• $I : L \to \varphi(C,X)$ assigns each location an invariant, which must be satisfied if $A$ want to stay in that location;
• $R \subseteq L \times \varphi(C,X) \times (Act \cup \{\tau\}) \times 2^{X \cup C} \times L$ is the set of transitions.

Given a transition $\langle l, g, a, r, l' \rangle \in R$, we write $\langle g, a, r \rangle$ as the edge of the transition, i.e., $l \xrightarrow{g,a,r} l'$. $A$ can move from location $l$ to location $l'$ if the guard condition $g \in \varphi(C,X)$ is satisfied. The transition may take some input / output events in $Act \cup \{\tau\}$, and perform a sequence of reset operations on clock or integer variables. Especially, assume the valuation in $l$ is $v \in Val(V)$, we use $r(v)$ to denote the valuation in $l'$ which uses the new values defined in $r$ to update $v$.

A Timed network (NTA) consists of a set of timed automata. These automata interact with CCS semantics. We refer readers to [2], [3] for more details.

Definition 6: Given a timed automaton $A = (E, Y, C, L, l_0, Act, I, R)$, the TTS associated with $A$, denoted as $TTS(A)$, is the tuple $(E, H \cup \{loc\}, S, s_0, Act, R')$, where

- $H = Y \cup C$, and $loc$ is a fresh variable which ranges over $L$ and represents the current location in $A$;
- $S = \{s \in Val(W) | s \models I(\text{val} (loc))\}$, where $W = E \cup H \cup \{\text{loc}\}$, $s(\text{loc})$ returns the value of $\text{loc}$ in the valuation $s$.
- $s^0 = v^0 \cup \{\text{loc} = l_0\}$, where $v^0$ maps variables in $E \cup H$ to their initial valuations;
- $R'$ is a set of transitions which can be either an action transition or a timed transition:

1) Action transition:

$$ l \xrightarrow{g,a,r} l', s(\text{loc}) = l, s \models g, s' = r(s)[\text{loc} \mapsto l'] $$

2) Timed transition:

$$ s' = s \oplus d $$

Lemma 1: Given a timed network $N = (A_1, \ldots, A_n)$ with alphabets $Act_1, \ldots, Act_n$, let $Act = \bigcup_{1 \leq i \leq n} Act_i$, then

$$ LTS(N) = LTS((TTS(A_1))|\cdots|TTS(A_n))|\text{Act}. $$

Lemma 1 shows that the composition semantics of the corresponding TTSs coincides with the noncompositional semantics of the LTS models. For detailed proof, we refer to [3].

C. Approximation of Timed Automaton with and without Invariants

When considering safety properties, for a timed automaton $A$ with location invariants, there exists an alternative timed automaton without location invariants (denoted as $A'$) which in semantics is over-approximative to $A$. We can shift the invariants in a location to its immediate ingoing or outgoing transitions. Given a transition $t = \langle l, g, a, r, l' \rangle$ with $r = \{x_1, x_2, \ldots, x_n\}$, when the automaton enters $l'$ from $l$, all these variables will be reset to zeros. We use the substitution function $f_0 = \{x_1/0, \cdots, x_n/0\}$ to denote the reset operation. Note the invariant constraint $I(l')$ must be satisfied when automaton stays in $l'$. Then $I(l') \cdot f_0$ should hold such that the automaton can enter $l'$ after taking the transition $t$. For simplicity, we denote $In(I(l'), r) = I(l') \cdot f_0$, where $f_0$ is defined as above. In general, $A'$ is approximative to $A$ except that each transition $t = \langle l, g, a, r, l' \rangle$ in $A$ is mimicked by a corresponding transition $t' = \langle l, g \wedge In(I(l'), r) \wedge I(l), a, r, l' \rangle$ in $A'$ and the invariants in all locations of $A'$ are assigned true. We conclude all behaviors in $A$ are preserved in $A'$. For a proof, we refer to [17]. We describe a simple example in Fig.1.

![Fig.1](image)

Fig. 1. A simple timed automaton is shown in the left part. We shift the location invariants in $s$ and $t$ to the transition connecting them in the right part. We have $In(I(l), r) = (u < 2) [u/0] = (0 < 2)$ then the transition guard is $g' = g \wedge In(I(l), r) \wedge I(s) = (u > 3) \wedge (0 < 2) \wedge (u < 5)$. The absolute time in location $s$ to trigger the transition should be in range $(3, 5)$.

D. Uppaal Path and Counter-example

Uppaal [18] is a modeling, validation and verification tool for real timed systems modeled as networks of timed automata. It extends TA with data types such as bounded integers and arrays. When the property doesn’t hold on the given model, Uppaal reports a counter-example. In this section, we briefly and technically discuss the form of the returned counter-example as Uppaal path. An Uppaal path $t$ is a finite sequence of system locations connected by system edges formally as $t = l_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{n-1}} l_m$.

For a safety property, the counter-example path starts at $l_0$ which is the initial location of the model and ends at $l_m$, a system location which violates the given property. Each system location is a composition of the locations from all the timed automata in the system. As we shift the location invariants from the locations to the corresponding transitions, clock variables and integer variables are only bounded by the guards of the transitions. In a system location, a satisfiable variable assignments to the guard $\varphi(C,X)$ of a location’s outgoing edge $e$ would trigger that edge. We therefore exploit this feature to monitor the change of clock variables in the locations on path $t$ to attach an absolute time to each location whose value represents the time elapsed from the beginning of this path. For example, considering the Uppaal path $s \xrightarrow{u} t$ in the right part of Fig.1, to trigger the edge, the guard condition $3 < u < 5$ must be satisfied, then we can conclude $u = 4$ (note $u$ is an integer variable).
III. COMPOSITIONAL ABSTRACTION FOR TIMED SYSTEM

In this section, we survey some existing results on compositional abstraction of timed systems at first and then introduce our main theorem.

Definition 7: Given two comparable TTSs $T_1 = (E_1, H_1, S_1, s_1, \sigma_1, Act_1, R_1)$, $T_2 = (E_2, H_2, S_2, s_2, \sigma_2, Act_2, R_2)$, we say that there exists a \textit{timed step simulation relation} $U \subseteq S_1 \times S_2$ from $T_1$ to $T_2$, denoted as $T_1 \preceq T_2$, provided that $s_1U s_2$ and for two states $s \in S_1, t \in S_2$ if $sUt$ then

1) $\forall y \in E_1$, $s(y) = t(y)$.
2) $\forall u \in Val(E_1)$, $s[u]U t[u]$.
3) if $s \xrightarrow{a} s'$ then either there exists $t'$ such that $t \xrightarrow{a} t'$ and $s'Ut'$, or $a = \tau$ and $sUt$.

Intuitively, for two TTSs, the existence of timed step simulation claims the related states must agree on shared variables and the relation must be preserved by consistently changing the shared variables. Each transition in $T_1$ is mimicked by a transition in $T_2$, except $\tau$ which can be simulated by doing nothing.

Lemma 2: Let $T_1, T_2, T_3$ be TTSs, $T_1$ and $T_2$ are comparable, $T_1 \preceq T_2$, and both $T_1$ and $T_2$ are compatible with $T_3$, then $T_1|T_3 \preceq T_2|T_3$.

Lemma 3: Given two comparable TTSs $T_1, T_2$ such that $T_1 \preceq T_2$, and a safety property $\mathbf{AG} \varphi$ defined over the external variables of $T_1$ and $T_2$. If $T_2 \models \mathbf{AG} \varphi$, then $T_1 \models \mathbf{AG} \varphi$.

The above theorems provide theoretical foundation for compositional abstraction of timed system [3]. Lemma 2 states timed step simulation can be applied to compositional abstraction. Lemma 3 states timed step simulation preserves the invariant property. It states that if an invariant hold on the abstract TTS then it also holds on the concrete TTS. Note the result of Lemma 3 can be extended to timed networks.

Corollary 1: Given a NTA $N = (A_1, \ldots, A_n)$, if $A_1||\cdots||A_n \models \mathbf{AG} \varphi$, then $A_1||\cdots||A_n \models \mathbf{AG} \varphi$.

Based on above observations, we now come to our main theorem which forms the basis of our refinement approach. It manifests the fact that counter-example validation and abstraction refinement for timed systems can be applied in a component-wise approach.

Theorem 1: Let $N = (A_1, \ldots, A_n)$ be a timed network with alphabets $Act_1, \ldots, Act_n$ respectively, define $Act = \bigcup_{1 \leq i \leq n} Act_i$. Suppose $TTS(A_1), \ldots, TTS(A_n)$ are compatible, then a path $t \in LTS(TTS(A_1)||\cdots||TTS(A_n))\backslash Act)$ exists if for each $1 \leq i \leq n, t_i \in LTS(TTS(A_i))$ exists, where $t_i$ is the path obtained by projecting $t$ on $TTS(A_i)$.

Proof: For $1 \leq i \leq n$, recall $TTS(A_i)$ is indeed a LTS. Note that we define the path in the form of a sequence $s_0, a_0, s_1, \ldots$, where $s_i$ is a state and $a_i$ is an action without noticing the shared variables. Suppose we model each shared variable in $N$ as a separate LTS which only contains one state and transitions doing read/write events. The original read/write operation in original timed automata in NTA are also changed to read/write events. These newly added LTS communicate with NTA via read/write synchronization actions. Theorem 1 is established immediately following the theorem in [13]: whether a trace belongs to a parallel composition of LTSs can be checked by projecting and examining the trace on each individual component separately.

IV. COMPOSITION ABSTRACTION REFINEMENT WITH CEGAR

In this section, we describe how we combine compositional abstraction and abstraction refinement to verify timed system. Fig. 2 illustrates the framework of our approach.

Given a timed network $N = (A_1, \ldots, A_n)$, we switch from the direct construction of $A_1||\cdots||A_n$ to the composition of each $A_i$ using abstraction $A'_i$. These abstractions are used to verify the specification instead. If Uppaal finds that the safety property holds on the above abstractions, then it also holds on the concrete model. Otherwise, a counter-example is returned. We validate whether the counter-example can be concretized on the full model. According to Theorem 1, to check if an abstract counter-example belongs to a concrete system, it is sufficient to check it on the individual component respectively by decomposing the counter-example. (Since it is proved in Theorem 1, the theorem is correct provided all shared variables are retained in the abstraction.) If the check passes, we report this counter-example is a real bug. Otherwise, the counter-example is spurious; we refine the abstract model and iterate the verification. We repeat this process iteratively to comply with the CEGAR principle.

A. Abstraction for Timed Automaton

Based on the theory of compositional abstraction, the problem remains to find an efficient abstraction for timed automata. Given a timed automaton, we first preprocess it with the technique introduced in section II-C which shifts the location invariants to the transitions. The property which holds on the preprocessed model certainly holds on the original model. In what follows, we use the preprocessed model instead. Let $A = (X, C, L, l_0, Act, I, R)$ denote the preprocessed automaton, and we write $A' = (X', C', L', l'_0, Act', I', R')$ as its abstraction.
The verification problem for a timed system is usually based on a zone graph with its scale in the worst case being equivalent to the size of the system’s region graph [2]. We have two obstructions: the number of regions in a region graph is exponential in the number of clocks, and the global state space grows exponentially to the number of processes in the timed system. The above facts directly lead our abstraction strategy around abstracting clock variables away and aggregating locations.

1. Clock variable omission guided abstraction

We abstract some clock variables away from timed automaton. We get \( C' \subseteq C \). With some clock variables omitted, the guards of some transitions are weakened.

2. Action guided abstraction

The locations in \( A' \) are called abstract locations and the locations in \( A \) are called concrete locations. We map arbitrary concrete locations to one location in the abstract model so each abstract location is a disjoint set of concrete locations. Our method is safe for do we not need to consider the location invariants because we already shift them to the corresponding transitions. Formally we write the abstract function \( \gamma : L \to L' \). The transitions \( R' \) are based upon \( \gamma \). For two abstract locations \( s, t \in L' \), we write \( s \xrightarrow{\gamma} t \) if there exist two concrete locations \( q, r \in L \) which guarantees \( q \xrightarrow{\gamma} r \) and \( \gamma(q) = s \land \gamma(r) = t \). The abstract initial location \( l'_0 \) satisfies \( \gamma(l'_0) = l'_0 \).

We write the concretization function \( \alpha : L' \to 2^L \). \( \alpha(l') = \{ l \in L | \gamma(l) = l' \} \). Note \( \alpha \) induces a partition on \( L \).

Let \( Z \) be the set of variables in the abstract model. Define \( abs_Z \) to be an operation on edges which eliminates the guard conditions and assignments that contain some variables not in \( Z \). Two edges are said to be equivalent if they share the same synchronization event and their guards and assignments are homomorphism under the operation \( abs_Z \). For instance, for two timed transitions, both with assignment \( y = 0 \), we say the edge with guard \( x > 2 \land y > 2 \) is equivalent to another edge with guard \( x > 2 \) if \( Z = \{ x \} \); but they are considered different if \( Z = \{ x, y \} \). After the abstract locations are established, we remove redundant equivalent edges connecting both the same source abstract location and the same target abstract location.

For a safety property \( \varphi \), the initial abstraction \( A' \) abstracts away all the clock variables except the ones appearing in \( \varphi \) using clock variable omission abstraction. Thus, we get \( C' = \text{ClockVar}(\varphi) \) and \( Z = X \cup \text{ClockVar}(\varphi) \) in the initial abstraction. For action guided abstraction, we use two different heuristics to construct \( \gamma \). One heuristic is to merge some locations together if one (or more) of their outgoing edges is equivalent. Formally, we write \( enable(l) \) for the set of edges outgoing from \( l \); given two concrete locations \( s, t \in L \), we require \( \gamma(s) = \gamma(t) \) if \( \exists e \in enable(s) \land e' \in enable(t), s.t., e =_Z e' \). In this way, it is important that we merge some transitions together as well as aggregate the locations. The other heuristic is to let the abstract function \( \gamma \) map all concrete locations into one abstract location, i.e., \( \forall s, t \in L, \gamma(s) = \gamma(t) \), if a large portion of the concrete locations have equivalent outgoing edges. It is immediate that a lot of transitions are merged in this case. Under both strategies, we consider the locations appearing in \( \varphi \) as separated abstract locations and their original invariants (but only retain the atomic constraints which involve the variables in \( Z \)) are attached to them. We write \( L' = \{ \gamma(s) | s \in L \} \) for the abstract locations. The concretization function \( \alpha \) can be obtained from \( \gamma \).

**Theorem 2:** Both clock variable omission abstraction and action guided abstraction are over-approximative abstractions.

**Proof:** Our method removes some clock variables with some guards weakened but preserves the behaviors of the original system. Our method merges concrete locations into abstract locations but preserves the original transition relations. It is obvious that our abstract and concrete model maintain the same set of shared variables and the initial states are always related. The locations in concrete model are exactly matched to locations in abstract model by partition function \( \gamma \). The values of shared variables are also matched as the simulation for the original transitions are still preserved in the abstract model. On the other hand, since we removed internal variables from concrete system weakening its guard, the abstract model is bisimilar to the concrete model differing only in the values of the abstracted internal variables. Hence, our abstraction establishes a timed step simulation from the concrete model to the abstract model in semantics of the underlying TTS; the paths of the concrete model are a subset of that of the abstract model.

### B. Automatic Abstraction Refinement for Timed Systems

Given an abstract counter-example path \( t' = l'_0 \overset{e'_0}{\rightarrow} l'_1 \overset{e'_1}{\rightarrow} \cdots \overset{e'_{n-1}}{\rightarrow} l'_n \) in the compositional abstract model \( A'_1[\cdots][A'_n] \), according to compositionality, a system location (edge) is a composition of the locations (edges) from each automaton. Formally, for \( 1 \leq j \leq p \), each \( l'_j \) is indeed a tuple of locations \( l'_j^{A'_1}, \cdots, l'_j^{A'_n} \) and each \( e'_j \) is a tuple of edges \( (e_1^{A'_1}, \cdots, e_1^{A'_n}) \).

For \( 1 \leq i \leq n \), we project \( l'_i \) on \( l'_i^{A'_i} \) and \( e'_i \) on \( e'_i^{A'_i} \) to obtain a path \( t'_i = l'_0^{A'_i} \overset{e'_0^{A'_i}}{\rightarrow} l'_1^{A'_i} \overset{e'_1^{A'_i}}{\rightarrow} \cdots \overset{e'_{m_i-1}^{A'_i}}{\rightarrow} l'_m^{A'_i} \) for \( A'_i \).

However, \( t'_i \) may still be not a legal path for \( A'_i \) because some edges on the path are possible null edges which connects two same locations in \( A'_i \). According to the composition rules in [1], the occurrence of null edges is caused by \( A'_i \)'s suspension when the rest of the system take a transition which can’t be synchronized by \( A'_i \). Based on the above fact, we remove these edges and the redundant locations to obtain a legal abstract path \( t'A'_i = l'_0^{A'_i} \overset{e'_0^{A'_i}}{\rightarrow} l'_1^{A'_i} \overset{e'_1^{A'_i}}{\rightarrow} \cdots \overset{e'_{m_i-1}^{A'_i}}{\rightarrow} l'_m^{A'_i} \) \((m \leq p)\).

In the following, we formulate how our abstraction refinement scheme exploits the abstract path \( t'A'_i \) to validate the counter-example and refine \( A'_i \) if needed. We will write \( t' \) for \( t'A'_i, A' \) for \( A'_i \) and \( A \) for \( A_i \) for simplicity.

1. Action guided abstraction refinement

Given a location \( s \) and a sequence of edges \( e* \), we use \( \text{reach}(s, e*) = \{ t | s \overset{e*}{\rightarrow} t \} \) to denote the location reachable from \( s \) through \( e* \), i.e., there exist locations \( l_0, \cdots, l_n \) with \( l_0 = s \) and \( l_n = t \) such that \( l_0 \overset{e_0}{\rightarrow} l_1 \overset{e_1}{\rightarrow} \cdots \overset{e_{n-1}}{\rightarrow} l_n \).
Given a set of locations $set$, we overload this notation by writing $reach(set, e^*)$ to represent the set of locations that are reachable through $e^*$ from some locations in $set$. For a path $t'$, we use $t'_j$ to denote the sequence of locations and edges from the initial location to the $j^{th}$ location on $t'$.

Algorithm 1 Action guided abstraction refinement

Require: $t' = l'_0 \xrightarrow{e'_0} l'_1 \xrightarrow{e'_1} \cdots \xrightarrow{e'_{m-1}} l'_m$ is the abstract trace and $l_0$ is the initial location.
Ensure: Return $false$ if $t'$ is not a real counter-example (with the abstraction being refined) else return $true$ and a concretized witness $t$.

1: $set_0 \leftarrow \{l_0\}$
2: $j \leftarrow 0$
3: while $set_j \neq \emptyset \land j < m$ do
4: $j \leftarrow j + 1$
5: $set_j \leftarrow reach(set_{j-1}, e'_{j-1}) \cap \alpha(l'_j)$
6: if $set_j = \emptyset$ then
7: split $l'_j$ into new abstract locations $l''_{j-1}', l''_{j-1} \in set_j$ s.t. $\alpha(l''_{j-1}') = \alpha(l''_{j-1}) \cap \{l \in L \land reach(l, e'_{j-1}) \cap \alpha(l'_j) \land \alpha(l''_{j-1}') \neq \emptyset\}$, $\alpha(l''_{j-1}) = \alpha(l''_{j-1}') \land \alpha(l''_{j-1})$
8: update $\gamma$ according to the split and refine the abstraction by $\gamma$.
9: return $false$
10: end if
11: end while
12: return $true$ and a path $t = l_0 \xrightarrow{c_0} l_1 \xrightarrow{c_1} \cdots \xrightarrow{c_{m-1}} l_m$ which concretizes $t'$.

Algorithm 1 illustrates the procedure for our method to eliminate the spurious counter-example. It takes input of an abstract path $t'$ and the initial location $l_0$ of the concrete model $A$. It returns $false$ and splits an abstract location on $t'$ to refine $A'$ if $t'$ is a spurious counter-example. Otherwise, it returns $true$ and a concretized path $t$ in $A$ corresponding to $t'$.

Theorem 3: Algorithm 1 is correct.

Proof: An abstract counter-example $t' = l'_0 \xrightarrow{e'_0} l'_1 \xrightarrow{e'_1} \cdots \xrightarrow{e'_{m-1}} l'_m$ is a real path under the action guided abstraction if there exists a concretized path $t = l_0 \xrightarrow{c_0} l_1 \xrightarrow{c_1} \cdots \xrightarrow{c_{m-1}} l_m$ such that for each $j$, $0 \leq j \leq m$, $l_j \cap \alpha(l'_j) \neq \emptyset$. Otherwise, $t'$ is considered as a spurious counter-example.

An abstract counter-example $t'$ can’t be concretized on the full model if there exists $j$ ($1 \leq j \leq m$) such that $reach(l_0, t_{j-1}) \cap \alpha(l'_j) = \emptyset$ (line 7). First, it is straightforward that there exist some concrete locations in $\alpha(l''_{j-1})$ reachable form the initial location $l_0$ (lines 3-5). Second, according to our definition about abstract transition, there exist some concrete locations in $\alpha(l''_{j-1})$ which could reach some concrete locations in $\alpha(l'_j)$ by edge $e'_{j-1}$. Based on the above facts, we conclude there are two kinds of concrete locations in $\alpha(l''_{j-1})$, which are categorized into $\alpha(l''_{j-1})$ and $\alpha(l''_{j-1})$ respectively:

- The locations in $\alpha(l''_{j-1})$ are not reachable from the initial location in trace $l_{j-1}$ but have $e'_{j-1}$ edge to some concrete locations in $\alpha(l'_j)$.
- Some locations in $\alpha(l''_{j-1})$ are reachable from the initial location in trace $t_{j-1}$ but all of them have no $e'_{j-1}$ successor in $\alpha(l'_j)$.

Splitting these two kinds of locations would suffice to eliminate $t'$ (lines 7-8).

Hence the algorithm either returns $false$ if $t'$ is not a real counter-example and refines the abstraction $A'$ with $t'$ being eliminated (line 9) or returns $true$ and a witness (line 12).

After performing Algorithm 1, for the abstract locations each of which maps to only a single concrete location, we attach their original invariants (but only retain the atomic constraints which involves the variables that are not abstracted away) to them.

In addition, we also use a heuristic. If a location $s$, $s \in l''_{j-1}$ and $s \notin set_{j-1}$, have one (or more) of their outgoing edges being equivalent to that of a location $t$, $t \in l''_{j-1}$, we transfer $s$ from $l''_{j-1}$ to $l''_{j-1}$. In essence, we aim to aggregate some locations as well as eliminate some transitions.

2. Clock variables omission guided abstraction refinement

When $t'$ is concretized on the abstract model in action guided abstraction refinement, there still lacks guarantee that it is a real bug for it may be brought by clock variable omission abstraction. Algorithm 2 illustrates our refinement strategy. It takes input of $t' = l'_0 \xrightarrow{e'_0} l'_1 \xrightarrow{e'_1} \cdots \xrightarrow{e'_{m-1}} l'_m$ and the concretized path $t = l_0 \xrightarrow{c_0} l_1 \xrightarrow{c_1} \cdots \xrightarrow{c_{m-1}} l_m$ which is obtained by the use of Algorithm 1. Algorithm 2 returns $true$ if $t'$ is valid. Otherwise, it puts some clock variables back once being abstracted away.

As shown in II-D, we attach the absolute time to the locations on $t'$. The attached absolute time indicates the right time when the immediate outgoing edge is triggered. Our goal is to ascertain if the absolute time which trigger an edge on the abstract path would also trigger the corresponding edge on the concretized path.

Therefore, we need to evaluate each clock variable (including abstracted clock variable) on each concrete location in the concretized path. Considering the clock variable $c_j$, let $c_j$ denote the timed value of $c$ on the $j^{th}$ location. The value of $c_j$ is determined by the function $update(c_{j-1}, l''_j, l''_{j-1})$, which is defined as follows:

$$c_j = \begin{cases} l_j.t & \text{if } j = 0 \\ l_j.t - l_{j-1}.t & \text{if } j > 0 \text{ and } e_{j-1} \text{ reset } c \\ c_{j-1} + l_j.t - l_{j-1}.t & \text{otherwise.} \end{cases}$$

On initial location, the value of $c_0$ just equals the absolute time; otherwise, if the clock $c$ is reset when taking transition from the $(j-1)^{th}$ location to $(j)^{th}$ location, the timed value of $c_j$ equals the difference of the absolute timed values on these two locations; otherwise, the timed value of $c_j$ equals the sum of $c_{j-1}$ and the difference of absolute timed values.

Theorem 4: Algorithm 2 is correct.

Proof: The timed value of each clock variable in each location on the concretized path is assigned by the time duration on the abstract path (line 3-9). Remember that we
Algorithm 2 Clock variable omission guided abstraction refinement

Require: \( t' = l_0' \xrightarrow{e_0'} l_1' \xrightarrow{e_1'} \cdots \xrightarrow{e_{m-1}'} l_m' \) is the abstract trace and \( t = l_0 \xrightarrow{c_0} l_1 \xrightarrow{c_1} \cdots \xrightarrow{c_{n-1}} l_n \) is the corresponding concrete trace.

Ensure: Return true if \( t' \) is a real counter-example or return false (with the abstraction being refined).

1: \( j \leftarrow 0 \)
2: while \( j < m \) do
3:   for each Clock \( c \in C \) do
4:     \( c_j \leftarrow \text{update}(e_j) \)
5:   end for
6:   if assignment of \( C \) makes guard \( (e_j) \) false then
7:     \( U \leftarrow \{ h \in \text{ClockVar}(e_j) \mid \text{guard}(h) \} \)
8:     put \( U \) back to refine \( A' \)
9:   end if
10: end while
11: return true

We derive the Train-gate protocol from Uppaal tool set. Our industrial partner provided us the Steeve Control system model. All verification tasks lead to positive results. For the Mutual Exclusion protocol and the Steeve Control system model, we use the first strategy mentioned in Section IV-A to get its initial abstraction. For the other benchmarks, we use the second strategy. We compare our CAREF technique with the Uppaal tool (version 4.0.8) on these benchmarks by total verification time. All the experiments were performed on Linux with 1GB memory and a 3GHz Intel processor.

In the Fischer protocol, the timed system consists of \( n \) processes. We verify a safety property that, at any time, at most one process is given permission to enter the critical resource. We iteratively increase \( n \) to evaluate our approach. The complexity of the problem explodes quickly as \( n \) increases. All these cases are verified through 3 abstraction refinement iterations. Experimental results are shown in Table I. The leftmost column denotes the processes in the current system. Upaal fails to construct the global state space when verifying 12 processes. In contrast, our method not only outperforms it in terms of efficiency but runs until \( n = 15 \).

The Train-gate protocol models a scene that 9 trains line up to enter a single gate. We again verify a safety property that any two trains are forbidden to enter the gate simultaneously. Table II shows the experimental result. The leftmost column denotes the number of iteration in CEGAR loop. CE length denotes the length of the trace of counter-examples returned by Uppaal. time is in seconds. The proposed method in CAREF verify the property through 2 abstraction refinement iterations in 89.736s while Upaal uses 133.613s. Note in CAREF all clock variables in Train-gate protocol are abstracted away.

V. EXPERIMENTS

We implemented our algorithm in a prototype called CAREF written in Java and incorporated it with the model checking tool Uppaal [18].

We apply our method to verify some well known benchmarks mainly from MCTA tool set [19], i.e., Fischer protocol, Arbiter Tree protocol, Mutual Exclusion protocol.

\[ T \xrightarrow{1} X \xrightarrow{2} W \xrightarrow{3} Z \xrightarrow{4} U \xrightarrow{5} Y \xrightarrow{6} V \xrightarrow{7} S \xrightarrow{8} Q \xrightarrow{9} P \xrightarrow{10} N \xrightarrow{11} M \xrightarrow{12} L \xrightarrow{13} K \xrightarrow{14} J \xrightarrow{15} I \xrightarrow{16} H \xrightarrow{17} G \xrightarrow{18} F \xrightarrow{19} E \xrightarrow{20} D \xrightarrow{21} C \xrightarrow{22} B \xrightarrow{23} A \]

The Arbiter Tree protocol models a mutual exclusion protocol based on a tree of binary arbiter processes. Client processes are situated at the leaves of the tree. In order to gain access to the shared resource, they may send a request to their respective parent recursively. When the root processes
of the tree receives a request, it generates a grant which is then propagated back down. We verify that two client processes in the Arbiter Tree protocol would not gain access simultaneously. Benchmarks A4, A5 and A6 contains 32, 64, 128 automata respectively. Our method verify these examples in 9 or 10 abstraction refinement iterations and lead to orders of magnitude speedup on verification time while Uppaal fails to verify all of them within 1440s as shown in Table III.

<table>
<thead>
<tr>
<th>A4 model</th>
<th>CE length</th>
<th>time</th>
<th>Model</th>
<th>CE length</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstr 1</td>
<td>3</td>
<td>0.01s</td>
<td>Abstr 2</td>
<td>7</td>
<td>0.206s</td>
</tr>
<tr>
<td>Abstr 3</td>
<td>7</td>
<td>0.243s</td>
<td>Abstr 4</td>
<td>11</td>
<td>0.286s</td>
</tr>
<tr>
<td>Abstr 5</td>
<td>12</td>
<td>0.405s</td>
<td>Abstr 6</td>
<td>13</td>
<td>0.845s</td>
</tr>
<tr>
<td>Abstr 7</td>
<td>13</td>
<td>0.53s</td>
<td>Abstr 8</td>
<td>18</td>
<td>0.772s</td>
</tr>
<tr>
<td>Abstr 9</td>
<td>19</td>
<td>4.491s</td>
<td>Abstr 10</td>
<td>verified</td>
<td>66.863s</td>
</tr>
</tbody>
</table>

**TABLE III**

Verification results for mutual property on A4, A5, A6 benchmark

Benchmarks N3 and N4 come from a case study of the Mutual Exclusion protocol. It models a real-time protocol to ensure mutual exclusion of states in a distributed system via asynchronous communication. Benchmarks N3 has 4 automata while N4 has 5 automata. Both benchmarks have 7 clock variables. Table IV shows the experimental result. In our proposed abstraction refinement scheme, we can prove the correctness of the model using an abstraction with a decrease of 2 clock variables. The verification time is significantly reduced by half compared to that of the full model.

<table>
<thead>
<tr>
<th>A5 model</th>
<th>CE length</th>
<th>time</th>
<th>Model</th>
<th>CE length</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstr 1</td>
<td>3</td>
<td>0.022s</td>
<td>Abstr 2</td>
<td>7</td>
<td>0.366s</td>
</tr>
<tr>
<td>Abstr 3</td>
<td>7</td>
<td>0.418s</td>
<td>Abstr 4</td>
<td>11</td>
<td>0.575s</td>
</tr>
<tr>
<td>Abstr 5</td>
<td>12</td>
<td>0.908s</td>
<td>Abstr 6</td>
<td>13</td>
<td>2.362s</td>
</tr>
<tr>
<td>Abstr 7</td>
<td>13</td>
<td>1.349s</td>
<td>Abstr 8</td>
<td>18</td>
<td>2.479s</td>
</tr>
<tr>
<td>Abstr 9</td>
<td>19</td>
<td>16.287s</td>
<td>Abstr 10</td>
<td>verified</td>
<td>295.26s</td>
</tr>
</tbody>
</table>

**TABLE IV**

Verification results for mutual property on N3, N4 benchmark

The Steeve Control system models an industrial platform that combines a control console and a set of motor-driven booms lifting freight. The system completes some given scenes through the movement of booms. Real time information such as timers are modeled by clock variables in timed automata to control the start, pause and stop of the motors and boom controllers. We model a mutual scene in which all booms are in place they must be exclusively processed so that the system must guarantee only one boom can enter the processing gate and the remaining booms are locked waiting for the time being. As it is indicated in Table V, when the system becomes increasingly complex, including more booms, our method is significantly faster than Uppaal. In these examples, the abstraction refinement loop have to iterate 3 times and put all clock variables back to generate a coarser abstraction to prove the property.

<table>
<thead>
<tr>
<th>A6 model</th>
<th>CE length</th>
<th>time</th>
<th>Model</th>
<th>CE length</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstr 1</td>
<td>3</td>
<td>0.057s</td>
<td>Abstr 2</td>
<td>7</td>
<td>0.793s</td>
</tr>
<tr>
<td>Abstr 3</td>
<td>7</td>
<td>0.873s</td>
<td>Abstr 4</td>
<td>11</td>
<td>1.336s</td>
</tr>
<tr>
<td>Abstr 5</td>
<td>12</td>
<td>2.562s</td>
<td>Abstr 6</td>
<td>13</td>
<td>7.888s</td>
</tr>
<tr>
<td>Abstr 7</td>
<td>13</td>
<td>4.134s</td>
<td>Abstr 8</td>
<td>18</td>
<td>7.477s</td>
</tr>
<tr>
<td>Abstr 9</td>
<td>verified</td>
<td>92.856s</td>
<td>CAREF Total</td>
<td>verified</td>
<td>117.689s</td>
</tr>
</tbody>
</table>

**TABLE V**

Verification results for mutual property on Steeve Control System comprised of 8, 10 booms.

In all these benchmarks, we are tackling both asynchronous and synchronous systems. We believe this fact witnesses our method’s scalability. The experiments demonstrate that our method is more capable to verify larger applications than Uppaal.

**VI. Conclusions**

In this paper we studied the compositional verification for timed systems. An automated approach which combines compositional abstraction and counter-example guided abstraction refinement (CEGAR) was presented. We implemented our method with the model checking tool Uppaal. Experiment results show some promising improvements.

**References**


