On-the-Fly Determinization of Bounded Networks of Timed Automata

Bernhard K. Aichernig
Florian Lorber
Institute for Software Technology, Graz University of Technology, Austria
{aichernig,florber}@ist.tugraz.at

Abstract—Timed Automata are established specification models for real-time systems. One of their main advantages is composability, allowing the modular specification of different aspects of a system via communicating timed automata. Composed together, these automata specify the behavior of a whole component or system. In previous work, we developed a technique to determinize a single timed automaton, by unfolding it and bounding ourselves to an observable depth \( k \). Within this paper we expand our approach, enabling the efficient bounded determinization of networks of timed automata. We realize an on-the-fly algorithm that performs at each level of unfolding the following tasks: building the product, hiding the communication, removing silent transitions, and determinizing. In contrast to the previous work, this on-the-fly algorithm only needs to traverse the state-space exactly once. We implemented the algorithm in the model-based testing tool MoMuT::TA and demonstrate and evaluate our implementation on a case study.

Keywords—timed automata; networks of timed automata; nondeterminism; silent transitions; bounded determinization

I. INTRODUCTION

Timed Automata (TA) [5] are an established formalism for specifying real-time requirements. They are used in several domains, as for instance synthesis, verification or model-based test generation. Several of these applications rely on deterministic timed automata, while the abstract specification of a system often requires non-deterministic models. Unfortunately, as non-deterministic timed automata are strictly more expressive than deterministic ones, they cannot be determined in general [5], [11], [17].

In previous work [15] we investigated a method for bounded determinizing of single timed automata. We started by unfolding it and bounding its observable traces. Because they are bounded, the resulting trees can then be effectively determined. While this approach looks very promising in the context of testing, where only finite traces are considered, it neglects one of the main advantages of TA: composability. Modern software is often split into several components \( C_1, \ldots, C_n \), so each may be designed according to its own set of requirements. This facilitates a clearer design, and reduces the complexity of modeling. The individual components can be composed into a system \( S = C_1 \ldots || C_n \). In the context of model-based testing, \( S \) is considered a black-box, where internal communication is hidden, and only external signals are observed.

The work presented in this paper applies our previous determination approach to networks of timed automata (NTA). We create an unfolded product of the network, where we on-the-fly remove all silent and communication transitions and determinize the system. This on-the-fly algorithm improves our previous work not only by its capability of processing NTA and building their product during unfolding, but also in terms of efficiency. While the previous algorithm had to traverse the unfolded tree several times (multiple times for silent transition removal, once for determinization and once for renaming all clocks), the on-the-fly algorithm does so exactly once. Note that an on-the-fly algorithm for single TA was mentioned in our previous work [15], but was never published.

The remainder of the paper is structured as follows: First, we give some preliminaries on TA and discuss MoMuT::TA, our tool for test-case generation, and bounded determinization in Section [II]. Then in Section [III] we define composition for networks of timed automata. In Section [IV] we explain the unfolding and determinization algorithm and illustrate it on the running example. In Section [V] we give some results on applicability of the tool. Finally, we discuss related work and conclude the paper in Section [VI].

II. PRELIMINARIES

A. Timed Automata with Inputs, Outputs and Silent Transitions

In this section we formally define Timed Automata with Inputs, Outputs and Silent Transitions (TAIOs), combining two previously used notions [3], [15]. Silent transitions are inner actions that are non-observable from the outside. We denote by \( \Sigma \) the finite set of actions, partitioned into two disjoint sets \( \Sigma^I \) and \( \Sigma^O \) of input and output actions, respectively, and by \( \Sigma_\epsilon = \Sigma \cup \{ \epsilon \} \) the set of Actions including the silent action \( \epsilon \). We call a TAIO without silent transitions fully-observable. A time sequence is a finite non-decreasing sequence of non-negative reals. A timed trace \( \sigma \) is a finite sequence of length \( k \) of time sequence and action pairs of the form \((t_1,a_1)\cdot(t_2,a_2)\cdots(t_k,a_k)\), where \( t_i \) is a time sequence and for all \( i \in [1,k], a_i \in \Sigma \).

Let \( X \) be a finite set of clock variables. A clock valuation \( v(x) \) is a function \( v: X \rightarrow \mathbb{R}_{\geq 0} \) assigning a non-negative real value to every clock \( x \in X \). We denote by \( H \) the set of all clock valuations and by \( 0 \) the valuation assigning 0 to every clock in \( X \). Let \( v \in H \) be a valuation and \( t \in \mathbb{R}_{\geq 0} \), we then have \( v + t \) defined by \((v + t)(x) = v(x) + t \) for all \( x \in X \). For a subset \( \rho \) of \( X \), we denote by \( v|\rho \) the valuation such that for every \( x \in \rho, v|\rho(x) = v(x) \). A clock constraint \( \varphi \) is defined by the grammar

\[
\varphi ::= z \circ z \mid \varphi_1 \otimes \varphi_2,
\]

where \( z = x \mid k \mid (z \times z), x \in \{+, -, \leq, <, >\} \) and \( \otimes \in \{\wedge, \vee\} \). Given a clock valuation \( v \in H \), we write \( v \models \varphi \) when \( v \) satisfies \( \varphi \).
Definition 1: A TAIO $A$ is a tuple $(Q, \hat{q}, \Sigma_e, \chi, I, \Delta)$, where $Q$ is a finite set of locations, $\hat{q} \in Q$ is the initial location, $\Sigma_e = \Sigma_l \cup \Sigma_o \cup \{\epsilon\}$ where $\Sigma_l$ is a finite set of input actions and $\Sigma_o$ is a finite set of output actions, such that $\Sigma_l \cap \Sigma_o = \emptyset$ and $\epsilon$ is a silent unobservable action, $\chi$ is a finite set of clock variables, $I: Q \rightarrow LI$ is a mapping from locations to location invariants, where each invariant $li \in LI$ is a conjunction of constraints of the form $true$, $x < n$ or $x \leq n$, with $x \in \chi$ and $n \in \mathbb{N}$; and $\Delta$ is a finite set of transitions $(q, a, g, q', \rho)$, where $q, q' \in Q$ are the source and target locations; $a \in \Sigma_e$ is the transition action; $\rho$ is the guard (a clock constraint); and $\rho \subseteq \chi$ is a set of clocks to be reset.

We say that a TAIO $A$ is deterministic if it is fully-observable and for all transitions $(q, a, g, q, \rho, \epsilon)$ and $(q, a, g, q, \rho, \epsilon)$ in $\Delta$, $q_1 \neq q_2$ implies that $g_1 \wedge g_2$ is unsatisfiable. We denote by $\Delta_o \subseteq \Delta(\Delta_l \subseteq \Delta)$ the set $\{\delta = (q, a, g, q', \rho) | \delta \in \Delta \wedge a \in \Sigma_o(\Sigma'_l, \text{resp.})\}$ of transitions labeled by an output/input action, resp.

The semantics of a TAIO $A = (Q, \hat{q}, \Sigma_e, \chi, I, \Delta)$ is given by the timed input/output transition system (TIOTS) $[\{A\}] = (S, \hat{s}, \mathbb{R}_{\geq 0}, \Sigma_e, T)$, where $S = \{(q, v) \in Q \times \mathcal{H} | v \models I(q)\}$, $\hat{s} = (q, \hat{0})$, $T \subseteq \Sigma_o(\Sigma'_l \cup \mathbb{R}_{\geq 0}) \times S$ is the transition relation consisting of discrete and timed transitions such that:

- **Discrete transitions:** $((q, v), a, (q', v')) \in T$, where $a \in \Sigma_e$, if there exists a transition $(q, a, g, q, \rho, \epsilon)$ in $\Delta$, such that: (1) $v \models I(q)$; (2) $v' = v[\rho]$; and

- **Timed transitions:** $((q, v), t, (q, v + t)) \in T$, where $t \in \mathbb{R}_{\geq 0}$, if $v + t \models I(q)$.

A *run* $r$ of a TAIO $A$ is the sequence of alternating timed and discrete transitions of the form $(q_1, v_1) \xrightarrow{\delta_1} (q_1, v_1 + t_1) \xrightarrow{\delta_1} (q_2, v_2) \xrightarrow{\delta_2} \cdots$, ending with a discrete transition, where $q_1 = \hat{q}$, $v_1 = \hat{0}$ and $\delta_1 = (q_1, a_1, q_1, \rho_1, t_1)$, inducing the timed trace $\sigma = ((t_1, a_1) \cdot (t_2, a_2) \cdot \cdots)$. We denote an observable timed trace a timed trace without silent transitions. These traces labeled by an output/input action, resp.

A bounded runs of $A$ is the sequence of alternating timed and discrete transitions of the form $(q_1, v_1) \xrightarrow{\delta_1} (q_1, v_1 + t_1) \xrightarrow{\delta_1} (q_2, v_2) \xrightarrow{\delta_2} \cdots$, ending with a discrete transition, where $q_1 = \hat{q}$, $v_1 = \hat{0}$ and $\delta_1 = (q_1, a_1, q_1, \rho_1, t_1)$, inducing the timed trace $\sigma = ((t_1, a_1) \cdot (t_2, a_2) \cdot \cdots)$. We denote an observable timed trace a timed trace without silent transitions. These traces labeled by an output/input action, resp.

2) Determinization of Bounded TA: We now give an overview on bound determination and silent transition removal of one single TA. Figure 1 shows the workflow: it takes UPPAAAL models as input and performs a bounded input/output conformance check (tioco) check between the test model and the mutants, that is realized via bounded model checking and solved using the SMT-solver Z3. The tioco check is restricted to single deterministic automata, as the conformance check might return spurious counter examples for non-deterministic models. Thus, to be able to process networks of timed automata, non-deterministic automata or non-observable timed automata, the on-the-fly algorithm presented in this paper is a necessary preprocessing step.

![Workflow of determinizing single TA up to depth 3.](image_url)
juncture of the guards of non-deterministic transitions. This is a two step process, were we first create diagonal constraints for all non-deterministic transitions and attach them to the transitions directly following the non-deterministic transitions. These diagonal constraints are built in a way that ensures their evaluation to true in all transitions beneath the original transition, iff the original transition was enabled. Then the non-deterministic transitions can be merged by disjunction of their guards. The diagonal constraints can later on decide which of them was actually enabled. The diagonal constraints are illustrated and marked bold in Figure 3(e) and the deterministic version is shown in (f), where the disjunction is marked.

III. NTA with Inputs, Outputs and Silent Transitions

We define a Network of Timed Automata with Inputs, Outputs and Silent Transitions $N = (A, \Sigma^I, \Sigma^O, \Sigma_I)$, where $A = \{A_1, \ldots, A_n\}$ is a set of TAIO, and the observable actions of the automata are split into three disjoint sets, where $\Sigma^I$ is the set of external input actions, $\Sigma^O$ is the set of external output actions and $\Sigma_I$ is the set of internal actions. Internal actions are exclusively used for synchronization between the TAIO, and the external output and input actions are exclusively used for communication with the environment. Formally, the three sets are defined as below:

$\Sigma^I = \{a \in A : a \in \Sigma^I \land \forall A_m \in A : a \notin \Sigma^O\}$

$\Sigma^O = \{a \in A : a \in \Sigma^O \land \forall A_m \in A : a \notin \Sigma^I\}$

$\Sigma_I = \{a \in A : a \in \Sigma_I \land \exists A_m \in A : a \notin \Sigma^O\}$

where $\Sigma^I$ and $\Sigma^O$ depict the inputs and outputs of the $m$-th automaton. We assume $\Sigma^I \cup \Sigma^O = \emptyset$.

For the coffee machine illustrated in Figure 4 we have the actions $\Sigma^I = \{\text{coin, button}\}$, $\Sigma^O = \{\text{coffee, tea}\}$, $\Sigma_I = \{t, c, paid, ready\}$ and $\epsilon$.

To define composition between TAIO, we extend the definition of parallel products between I/O Automata introduced by David et al. [9], by including silent transitions and hiding communication transitions after the product. The hiding of the communication transitions provides a black-box view on the system. Our definition differs from the one by Krichen and Tripakis [13], as our TAIO, do not contain deadlines. Furthermore, our definition is less restrictive, as we do not require disjoint sets of external input and output actions, respectively. Thus our product may produce non-deterministic automata, which will be determinized later on. Note that, in contrast to our definition below, these previous notions of composition are defined for exactly two automata: in the first definition [9] two synchronizing transitions form a new output transition, that can be used for composition with another automata. Thus two different automata can synchronize on the same output transition of a third automaton simultaneously. However, contrary to our definition, the communication transitions are never hidden, thus they are observable to the environment. In the second definition [13] communication transitions are hidden by the product. Thus, a third automaton could not synchronize with the product anymore, making the product definition not associative.

Our product definition is associative for the closed set of TAIO, it is applied to, but loses this property if an additional TAIO, is composed afterwards, due to the hiding.

**Definition 2:** The parallel product of an NTA $N$ composed of $n$ TAIO, $A_i$ is the non-deterministic TAIO, $A = (Q,(q_1,\ldots,q_n),\Sigma^O \cup \Sigma^I \cup \{\epsilon\},A_1 \cup \cdots \cup A_n,I,\Delta)$ where $Q \subseteq (Q_1 \times \cdots \times Q_n)$, $I = (Q_1 \times \cdots \times Q_n) \rightarrow LT$, s.t. $I(q_1,\ldots,q_n) = I_1(q_1) \land \cdots \land I_n(q_n)$ and $Q$ and the set of transitions $\Delta$ is defined by the following inductive rules, where $(\ldots, q_i, \ldots, a,g,p) (\ldots, q'_i, \ldots)$ means that only the state of the $i$-th automaton changes:

**Start**

\[
\begin{align*}
\text{External} & : q_i \xrightarrow{a,g,p} q'_i \quad \forall i, q_i \in Q, (\ldots, q_i, \ldots, a,g,p) (\ldots, q'_i, \ldots) \in Q \\
\text{Internal} & : q_i \xrightarrow{a,g_1,p_1} q'_i \quad q_j \xrightarrow{a,g_2,p_2} q'_j \quad a \in \Sigma^I \quad \forall i, q_i \in Q, (\ldots, q_i, q_j, \ldots) \in Q \\
\text{Silent} & : q_i \xrightarrow{c,g,p} q'_i \quad \forall i, q_i \in Q, (\ldots, q_i, \ldots, c,g,p) (\ldots, q'_i, \ldots) \in Q
\end{align*}
\]
The product of NTA can grow exponentially. Thus, for determining an NTA it does not in general make sense to build the product and do the determination afterwards, as this requires traversing through the state-space twice. In this section we will adapt our previous determination approach, so it can be directly applied on NTA. This is achieved via a bounded unfolding of the network to depth \( k \), where each level of unfolding includes on-the-fly product building, renaming of all clocks (so each transition of same depth resets the same clock), removing silent and communication transitions, and determinization. In the previous algorithm these steps were performed in separate iterations, where each iteration had to explore the whole state space. Additionally, removing a silent transition required traversing through the complete subtree below, updating all future guards to ensure their correct time behavior. The on-the-fly algorithm stores all constraints needed for these updates, and applies them to all transitions when they are reached by the unfolding. Proofs of the individual steps of the determination process for single TA can be found in our technical report [16].

The unfolding will be performed on the NTAAN = \( (A, \Sigma^L, \Sigma^O, \Sigma_i, \Sigma_f) \) where \( A = \{A_1, dots, A_n\} \) and each \( A_i = (Q_i, \delta_i, \Sigma_s, i, X_i, I_i, \Delta_i) \). The core part of the algorithm processes state tuples of the form \((q_i, q_1, i, e, CR, GLU, GS, NG)\), where \( q_i \) is the location in the unfolded tree, \( q_1 = (q(i,1), dots, q(i,m)) \) are the current locations in the NTA, \( i \) is the current depth and \( e \) is the number of silent transitions already processed at the end of the current trace. \( CR \) (clock renaming) is a partial function mapping clocks to clocks, used for renaming of the clocks. The renaming ensures the property that in each transition only one clock is renamed and that all transitions of same depth reset the same clock. \( GLU \) (guard update) is a partial function mapping clocks to constraints, storing the constraints of removed silent transitions. Keeping these constraints allows the updating of future guards that refer to a clock that was reset on a deleted silent transition, to keep the timing constraints valid. \( GS \) (guard synchronization) is a partial function mapping clocks to constraints, storing the constraints of removed silent transitions, \( C R \) mapping every clock to \( x_0 \) and \( GLU, GS, NG \) being empty.

The outermost loop in the algorithm increases the index \( i \) from 1 to \( k \). At each step, \( P_i \) contains the state tuples of the current depth, that need yet to be processed. \( P_{i+1} \) contains the state tuple of the next observable depth. Then, for every element in \( P_i \) three stages are executed. In Line 12 we process all silent transitions that can be taken in any of the automata \( A_i \) in their current location. This does not produce any new locations or transitions in the tree, but creates new state tuples to be stored in \( P_i \). Details of this stage can be found in Subsection IV-A and Alg. 3. The second stage is performed in Line 13 where communicating transitions between the automata are synchronized. The synchronization involves an action \( a \in \Sigma_i \) is only performed if one of the current locations of an automaton \( A_i \) in the NTA provides \( a \) as an output and another automaton \( A_j \) accepts it as an input in its current location. As the synchronized transitions are hidden, they are afterwards processed in the same way as the silent transitions. Details can be found in Subsection IV-B and Alg. 4. The third and last stage handles external transitions. This might introduce new locations and transitions in the tree, if one of the old automata can perform a transition with a label not yet available in the current location of the tree. If the tree already contains a transition with same label and source location, the transition is modified, by disjuncting its guard and the guard of the new transition. The state tuples produced in this step are stored in \( P_{i+1} \), to be processed in the next depth iteration. This is explained in detail in Subsection IV-C and Alg. 5.

Algorithm 1 Deterministic bounded unfolding of NTAAN.

Require: NTAAN, max. depth \( k \)
Ensure: TAIOA such that \( L_k(N) = L_k(A) \)

1: \( q_1 \leftarrow (q_1, dots, q_n) \); \( q_i \leftarrow \) new location
2: \( locations \leftarrow \{q_1\}; transition \leftarrow \emptyset \)
3: \( CR : Clock \rightarrow Clock = (x \mapsto x_0) \forall x \in X_i \) // clock renaming
4: \( GLU : Clock \rightarrow Constraint = \emptyset \) // constraints for future guard update
5: \( GS : Clock \rightarrow (Clock, Constraint) = \emptyset \) // constraints for future guard synch.
6: \( NG : \{\text{Constraints}\} = \emptyset \) // constraints to resolve non-determinism
7: \( P_1 \leftarrow \{(q_1, q_2, 0, 0, CR, GLU, GS, NG)\} \)
8: for \( i \in 1..k \) do
9: \( P_{i+1} \leftarrow \emptyset \)
10: while \( P_i \neq \emptyset \) do
11: \( Pick(q_i, q_1, i, e, CR, GLU, GS, NG) \in P_i \)
12: \( P_i \leftarrow P_{i+1} \cup CLOSURE(q_i, q_1, i, e, CR, GLU, GS, NG) \)
13: \( P_i \leftarrow P_i \cup INT(q_i, q_1, i, e, CR, GLU, GS, NG) \)
14: \( P_{i+1} \leftarrow P_{i+1} \cup EXT(q_i, q_1, i, e, CR, GLU, GS, NG) \)
15: end while
16: end for

Example 1: Figure 5 shows the first steps of applying the algorithm to the coffee machine example illustrated in

Fig. 5. Different steps of building the determinized unfolding.

\footnote{The approach is restricted to NTA that do not contain loops of purely silent and communication actions.}
Figure 4. $P_2$ is initialized with the state tuple $(q_1 = q_1, q_2 = (q_1, q_2, q_3, 1), i = 0, e = 0, CR = \{x \mapsto x_0, y \mapsto x_0, z \mapsto x_0\}, GU = \emptyset, GS = \emptyset, NG = \emptyset)$. Starting at the initial state of all three automata, there are no silent or internal transitions enabled. Thus, we start with processing external transitions, where coin? is the only enabled transition. As there is no transition leaving $q_1$ with the label coin? yet, we build a new location $q_2$ and a transition leading there, labeled by coin? (see Figure 4). This step adds the state tuple $(q_2 = q_2, q_2 = (q_1, q_2, q_3, 1), i = 1, e = 0, CR = \{x \mapsto x_1, y \mapsto x_0, z \mapsto x_0\}, GU = \emptyset, GS = \emptyset, NG = \emptyset)$. Thus no trace in the tree can violate any of the original invariants. We only consider traces ending in discrete steps, where both guards are combined via conjunction (conjunction would then reset $CR$, which is reset. First we build a combined transition, where previous entries for $x_1,0$ were empty, as $i,e$ will be substituted by true, where both guards are combined via conjunction (conjunction will be omitted), the clock resets are combined by union and the label is turned into $e$. This combined transition is shown in Figure 5 (b). This is just an illustration, actually the transition will be removed again when it is processed. Then we substitute the clocks according to $CR$, changing the guard from $x > 1$ to $x_1 > 1$ and the clock reset to $x_1$ (This transition is the first silent transition removed in the current trace on depth 1, thus we use $x_1$. A second silent transition in the same trace would then reset $x_1$ and $GU, GS$ and $NG$ are empty and thus not applied. We update $CR$ to $CR'$ by adding $x \mapsto x_1$. Thus, in future guards that refer to $x$, $x$ will be substituted to $x_1$. We update $GU$ to $GU'$ by adding $x_1 \mapsto x_1 > 1$. Thus, if a future guard $g$ refers to $x_1$ we build a constraint that refers to $x_1$ instead of $x_0$, by combining $g$ and the constraint $x > 1$ from $GU'$. Finally, we add all lower bounds occurring in the guard ($x_1 > 1$) to $NG'$. They will be attached to the following transitions and ensure they don’t occur to soon. Together that gives the state tuple $(q'_1 = q_2, q'_2 = (q_1, q_2, q_3, 1), i' = 1, e' = 1, CR' = \{x \mapsto x_1, y \mapsto x_0, z \mapsto x_0\}, GU' = \emptyset, GS' = \emptyset, NG' = \emptyset, 1 > 1\}, added to $P_2$.

For the new tuple, there are two observable transitions with the same label (button?) enabled. First we process the one without guard, leading to $q_2$. There exists no transition with label button? leaving $q_2$ in the tree, thus we add the new location $q_3$ and a transition leading there. Then we add $NG$ ( $x_1 > 1$) to the guard (see Figure 3 (c)) and change the clock reset from $\{y\}$ to $\{x\}$. We update $GS'$ by adding $x_1 \mapsto (x_2, x_2 > 1)$, where $x_2$ is the clock that is reset on the current transition, and $x_1 > 1$ is its guard. If a later transition also refers to $x_1$, it can be synchronized with the current transition, so the delay between them is valid. After adding the transition we update $NG'$ by building the diagonal constraint $x_1 - x_2 > 1$, subtracting $x_2$ (the clock reset on the current transition) from all clocks in the current guard $x_1 > 1$. $NG'$ will be conjunctioned to all following transitions, so these later transitions are only enabled, if $x_1$ was greater than 1 at the time of the button? transition. The tuple $(q'_1 = q_3, q'_2 = (q_1, q_2, q_3, 1), i' = 2, e' = 0, CR' = \{x \mapsto x_1, y \mapsto x_2, z \mapsto x_0\}, GU' = \emptyset, GS' = \emptyset, NG' = \emptyset, 1 > 1\}, NG' = \emptyset, 1 > 1\}, added to $P_3$. For the second button? transition there already exists a transition with same label in the tree, so the two transitions are determined after the second transition is updated. The determination works via disjunction, as illustrated in Figure 5 (d). The produced state tuple is $(q_1 = q_3, q_1 = (q_1, q_2, q_3, 1), i = 2, e = 0, CR' = \{x \mapsto x_1, y \mapsto x_0, z \mapsto x_2\}, GU' = \{x_1 \mapsto x_1 > 1, GS' = \{x_1 \mapsto x_1 > 1\}, NG' = \emptyset, 1 > 1\}, NG' = \emptyset, 1 > 1\})$. The disjunction produces a weaker guard, that is enabled if any of the original guards were enabled. The diagonal constraint stored in $NG'$ in the two state tuples are later attached to the following transitions (see Figure 5 (e)), to ensure that $t$ is only enabled, if $x_1 < 3$ was enabled in the combined button? transition. The combined transition is also enabled for any value $x_1 > 3$, but then only $c$ can be taken later on. Note that the $c$ and $t$ transitions in Figure 3 (e) are internal, and will be removed when processed.

The next three subsections give details on processing silent, external and external transitions. Seven operations are equally called in the first step of every of these stages. They are shown in Alg. 2 and explained below.

**Add Invariants to guard:** First, we build a conjunction of the invariants of all currently selected locations in the NTA. This conjunction is added to the guard of the processed transition, to make sure that traversing the transition is valid in all automata. The invariants are not added to the determined tree, as now all transitions have the invariants incorporated into their guard. Thus no trace in the tree can violate any of the original invariants. We only consider traces ending in discrete steps, neglecting time progress in the leaves of the tree.

**Apply $NG$:** $NG$ stores constraints (like the enabling guard) that need to be attached to all transitions directly following the transition that produced it. Applying $NG$ is done by conjuncting all its constraints to the guard.

**Update $CR$:** Every clock $x$ that is reset on the currently processed transition will be added to $CR$. If the current transition is a silent transition (or one produced by synchronizing two communication transitions) $x$ will in the future be substituted by the clock $x_{1,e}$, otherwise it will be substituted by $x_1$. Thus, if for example the two clocks $x$ and $y$ are reset on an observable transition on depth 5, both $x \mapsto x_5$ and $y \mapsto x_5$ will be added to $CR$, where previous entries for $x$ and $y$ are deleted. Note that all clocks $x_{1,e}$ will be removed in later steps, leaving only the clocks $x_1 - x_k$ in the final automaton.

**Substitute according to $CR$:** All clocks that appear in the guard of a processed transition $t_i$ need to be substituted according to the function $CR$. In the first steps, this means that all clocks are substituted by $x_0$, later on, as $CR$ will be

---

Note that in the explanation we swapped the order, to ease comprehensibility, however the order in the algorithm is the correct order to call them.
Algorithm 3 \(\epsilon\)-Closure

Require: \((q_l, q_i, i, e, CR, GU, GS, NG, NG')\)
Ensure: \(\{(q_l, q_i, i, e + 1, CR', GU', GS', NG')\}\)
1: \(P \leftarrow \emptyset\)
2: for each location \(q_{(i,j)} \in q_i\) do
3:   for each transition \(t_i = (q_{(i,j)}, e, \rho_i, q_{(i,j)})\) do
4:     \((g', CR', GS') \leftarrow \text{update transition}(t_i)\)
5:     \(GU' \leftarrow \text{update guard}(GU)\)
6:     \(e \leftarrow \text{calculate enabling guard}(g')\)
7:     \(NG' \leftarrow \text{update enabling guard}(e) \cup \text{taken guard}\)
8:     \(P \leftarrow P \cup \{(q_l, q_i[q_{(i,j)}\backslash q_{(i,j)}], i, e + 1, CR', GU', GS', NG')\}\)
9: end for
10: end for
11: return \(P\)

**Substitute according to \(GU\):** If a clock \(x_{j,e}\) is already defined in \(GU\) this means it was reset on a silent transition that was processed earlier in the trace leading to the current transition. If such a clock appears in the guard \(g\) of the currently processed transition \(t_i\), the guard constraints referencing \(x_{j,e}\) will be updated using the constraints stored in \(GU(x_{j,e})\). This update removes the lower bound \(x_{j,e} > m\) from \(g\) and for each lower bound \(x > n\) in \(GU(x_{j,e})\) it adds the constraint \(x > n + m\) to \(g\) via disjunction. Upper bounds are treated equally, and is treated as \(<=\) and \(>=\). Note that the built constraint contains strict inequality as long as one of the original constraints contained strict inequality.

**Update \(GS\):** \(GS\) needs to be updated every time we process a transition that refers to a clock that was reset on an internal or silent transition. It is used (see next item), to synchronize all transitions that refer to that clock. For these synchronization constraints, \(GS\) needs to store the constraints referring to the clock and the clock that is reset on the current transition.

**Apply \(GS\):** \(GS\) needs to be applied to every transition referring to a clock \(x_{j,e}\) that was reset by a silent transition (despite the first one, where it is only updated) to synchronize these transitions. The synchronization is done by building the following constraint: For each upper bound \(x_{j,e} < n\) stored in \(GS(x_{j,e})\), linked to the clock \(y\) (the clock reset on the transition from which we extracted \(x_{j,e} < n\)) and each lower bound \(x_{j,e} > m\) in the current guard, we build the constraint \(y > m - n\), ensuring that the time that passed between the two transitions is not larger than the difference between the lower bound of the first transition and the upper bound of the latter one. Assume the constraint \(3 \leq x\) of one transition, and the constraint \(x \leq 5\) at a later transition. Then at most \(2(5-3)\) seconds may pass between the two transitions. Similar constraints are built for the lower bounds in \(GS(x_{j,e})\) and the upper bounds in the guard.

More details on the constraints can be found in the paper on determination for single TA [13].

A. \(\epsilon\)-Closure

Given a tuple \((q_l, q_i, i, e, CR, GU, GS, NG)\), Algorithm 2 depicts the workflow of processing all silent transitions that can be taken in any location \(q_{(i,j)} \in q_i\) (Lines 9–11). Most of this workflow correlates to the step of removing silent transitions presented in [16]. The main difference is that the previous approach had to traverse the whole subtree beneath each silent transition, to update it. Contrary to that, we only keep all constraints we need for the updating stored, so we can update the future transitions at the time we process them.

**Update transition:** In the first step, the guard of the transition is updated according to the constraints of previous silent transitions or non-determinism. Details are given in Algorithm 3. This step also computes \(CR'\) and \(GS'\).

**Update \(GU\):** Every time a silent transition is processed, we store all upper and lower bound constraints occurring in its guard, mapped to the clock that was reset on the silent transition. The constraints are later used to update guards that refer to that clock, ensuring they keep valid time constraints.

**Calculate Enabling Guard:** When removing a silent transition, we can express the upper bound until when it is enabled via diagonal constraints, by pairwise comparing the upper and lower limits of all clocks appearing in the guard, to identify the time interval when all clocks are enabled at once. The exact details on building the constraints can be found in [16].

**Update Enabling Guard:** In the previous work [15] we built a bypass transition for every silent transition. That was a copy of the transition preceding the silent one, augmented by the enabling guard and leading to the target location of the silent transition. For example, the transition \(q_1 \to q_3\) in Figure 2(d) is such a bypass transition. During determination, this transition was removed again and only the enabling guard, attached to the following transitions, remained. Now we can avoid building the transition, by immediately attaching the enabling guard to the following transitions: we create the constraint \(NG'\) by subtracting in the enabling guard the clock that was reset on the preceding transition from all clocks appearing in the enabling guard. Let \(x_2 < 5\) be an enabling guard, and \(x_3\) be the clock reset on the preceding transition, then \(x_2 - x_3 < 5\) is the processed enabling guard. This builds a time invariant that, in all following transitions, can decide whether the enabling guard was enabled at the time of the preceding transition or not. \(NG'\) will be attached to all transitions that can leave the new target states in the NTA. We also add the taken guard to \(NG'\), that is a conjunction of all lower bounds in the guard. This ensures that no following transition can be taken earlier than allowed by the silent transition.

**Storing the next tuple:** Finally, the next state tuple is created and stored in \(P\), the set of tuples returned to the main algorithm (Line 8). This tuple reflects the next combination of locations in the NTA that needs to be processed. As no externally observable transition was processed, it is still linked to the same location in the tree and \(i\) is not increased. In the set of current locations in the NTA, only the location \(q_{(i,j)}\) in the j-th automaton (the automaton containing the processed silent transition) changes to \(q_{(i,j)}\). Finally, \(e\) is increased by one.

B. Internal Transitions

Internal transitions are processed similarly to silent transitions. The main difference is that they are only processed pairwise, and only if an action is enabled in one automaton as an input and in another as an output. Lines 8[2][3] collect all input and output transitions with label \(a \in \Sigma\) that are enabled in the different automata. Then for each combination of an input and an output transition (Line 10), we build a combined transition \(t\) (Line 11), where the action label is \(e\), the guard is the conjunction of both guards and the set of reset clocks is the union of the two set \(\rho_i\) and \(\rho_o\). The transition is processed equally as silent transitions until Line 16 where we the current locations of both involved automata is updated.
Algorithm 4 Processing Internal Transitions in the NTA.

Require: \((q_i, q'_i, i, e, CR, GU, GS, NG)\)
Ensure: \(\{(q, q'_i, i + 1, CR', GU', GS', NG')\}\)
1: \(P \leftarrow \emptyset\)
2: for each \(a \in \Sigma_i\) do
3: \(I \leftarrow \emptyset\); \(O \leftarrow \emptyset\)
4: for each location \(q_{i,j} \in q_i\) do
5: for each transition \(t_i = (q_{i,j}, a, g_i, \rho_i, q_{i,j})\) do
6: if \(\text{isOutputTransition}(t_i)\) then \(O \leftarrow O \cup \{(t_i, j)\}\)
7: else \(I \leftarrow I \cup \{(t_i, j)\}\) end if
8: end for
9: end for
10: for each \((t_o, k) \in O\) and each \((t_i, l) \in I\) do
11: \(t(q, e, g_o \land g_i, \rho_o \cup \rho_i, q') \leftarrow \text{combineTrans.}(t_i, t_o)\)
12: \((g', CR', GS') \leftarrow \text{update transition}(t)\)
13: \(GU' \leftarrow \text{update GU}\)
14: \(e \leftarrow \text{calculate enabling guard}(g')\)
15: \(NG' \leftarrow \text{update enabling guard}(e) \cup \text{taken guard}\)
16: \(P \leftarrow P \cup \left\{(q, g_i, q'_{i,k}\{q(h,k), q(i,l)\}\{q(h,l)\}, i, e + 1, CR', GU', GS', NG')\right\}\)
17: end for
18: end for
19: return \(P\)

C. External Transitions

For each enabled external transition we do the following:

Update transition: First we update the transition according to the constraints of the previous silent transitions, as done for silent and internal transitions.

External transition in the tree: If there already exists a transition with the same label in the current location of the tree, we disjunct the current guard to it, to ensure that the transition is always enabled when the currently processed transition is (Line 7). If it does not yet exist, we create a new location in the tree (Line 9). Then we create a new transition leading to this new location, with the label of the currently processed transition, resetting the clock \(x_i\) and \(y\) (the guard) we gain from updating the transition) as guard (Line 9).

Update \(NG\): We update \(NG\) by building diagonal constraints in \(g\). This is done by subtracting \(x_i\) from every clock that appears in \(g\). These diagonal constraints can in later transitions decide, whether \(g\) was enabled in the current transition. By attaching them to the following transitions, we can ensure that even though we possibly disjuncted the guard \(g\) to other guards (and thus weakened the guard), the following transitions will only be enabled iff \(g\) was enabled.

V. Applicability

We implemented the presented algorithm as an additional feature of our test-case generation tool MoMuT::TA. The tool and the examples are available (https://momut.org/?page_id=396). All tests were executed on a MacBook Pro (2.8 GHz Intel Core i7, 8 GB RAM). The implementation also contains improvements like storing all state tuples and only processing new tuples if there was no equal tuple before.

Table 1 shows the results of applying the tool to the running example, in terms of size of the unfolded network and calculation time for different depths. We extended the 2nd and 3rd automata in the network, to allow selecting a broader variety of drinks (still all buttons are non-deterministically labeled by button’?), producing automata with 2–5 different choices. As expected, we experience an exponential increase in complexity depending on depth and amount of choices (2–5 drinks). Fortunately, in most realistic cases much lower depths are sufficient, e.g., here, depth 3 covers all observables and thus a test suite covering all traces of length 3 already gives us transition coverage. For the last two industrial case studies we processed with MoMuT::TA, that is, a car alarm system [4] and an automated speed limiter (the studies are not yet published), depth 15 was always sufficient. Note that in these case studies we performed test-case generation from single deterministic automata, as they were done prior to the extension described in this paper. Hence we only refer to them, to provide insight into the needed depths for test-case generation. For the car alarm system we also performed the determinization process later on [15], but again for a single automaton.

The biggest challenge in the workflow of model-based mutation testing is not the determinization, but the many conformance checks applied afterwards (see Section 1). The checks get more complex for models with larger amounts of transitions, and thus suffer from the unfolding. For the coffee machine with 2 drinks a conformance check with an equivalent mutant (that needs to explore the complete state-space, as there is no counter example allowing early termination) takes 2.3 seconds on depth 10, 105.8 seconds on depth 20 and runs into a timeout (2 hours) on depth 30. The determinization only took 0.1, 0.4 and 1.7 seconds, respectively.

The complexity of the conformance checks can be reduced in several ways, as e.g. using tio-co-conform partial models. These can either be produced manually, or automatically during the unfolding, as we did in previous work [3]. We also plan to incorporate the conformance checks into the on-the-fly algorithm, to tackle the full extent of the state-space explosion.

We want to emphasize that the new on-the-fly algorithm performs a lot faster for single automata than the original implementation. We already reported the improved results [15].

Algorithm 5 Processing external transitions in the NTA.

Require: \((q_i, q'_i, i, e, CR, GU, GS, NG)\)
Ensure: \(\{(q, q'_i, i + 1, 0, CR', GU', GS', NG')\}\)
1: \(P \leftarrow \emptyset\)
2: for each location \(q_{i,j} \in q_i\) do
3: for each action \(a \in \Sigma_i \cup \Sigma_O\) do
4: for each transition \(t_i = (q_i, a, g_i, \rho_i, q_{i,j})\) do
5: \((g, CR', GS') \leftarrow \text{update transition}(t_i)\)
6: if there already exists a trans. \(t_i\) labeled by \(a\) in the tree then
7: \(g_i \leftarrow g_i \lor g\)
8: else
9: \(q'_i \leftarrow \text{new location}\)
10: \(t_i \leftarrow \text{new transition}(q_i, a, g, \{x_i\}, q'_i)\)
11: end if
12: \(NG' \leftarrow \text{update } NG\(g)\)
13: \(P \leftarrow P \cup \left\{(q_i, q'_i, \{q_{i,j}\} \{q_{i,j}\}, i + 1, 0, CR', GU, GS', NG')\right\}\)
14: end for
15: end for
16: return \(P\)
yet without giving details on the algorithm. For the three different examples presented in that paper (one of them is an industrial model of a car alarm system), the on-the-fly algorithm reduced the runtime for the highest feasible depth (a) from 350 minutes to 10 seconds, (b) from 33 minutes to 0.5 seconds, and (c) from 250 minutes to 10 seconds.

VI. CONCLUDING REMARKS

We presented an efficient algorithm for building a bounded, fully-observable and deterministic TA from non-deterministic networks of timed automata. We defined composition for NTA with inputs, outputs and silent transitions, with hiding of all communication actions. The algorithm performs several operations in parallel, and thus also works more efficiently on single automata than our previous technique.

Various definitions for composition of Timed Automata with Inputs and Outputs exist [9], [13]. In Section III we discussed the differences to our approach.

Bouyer et al. [8] define unfolding of networks of timed automata. Contrary to our work, they do not determinize and they unfold into timed Petri nets instead of TA. Bérand et al. [7] showed that silent transitions extend the expressive power of TA and cannot be removed in general. The authors identify a class of TA with silent transitions that have the same expressive power as general TA, and propose a procedure for removing silent transitions. Baier et al. [6], first propose a procedure for translating non-deterministic TA to infinite deterministic trees, and then identify several classes of NTA that can be effectively determinized into finite DTA. The difference between our bounded determinization, that works on any TA, and these approaches, that work on determinizable subclasses, was already discussed in our paper about bounded determinization of single timed automata [16]. The contribution of our current paper to this previous one is both the support for NTA and on the more efficient on-the-fly algorithm. Wang et al. [19] presented a highly optimized zone-based language inclusion check for TA. However it doesn’t support silent transitions, and only terminates for determinizable classes of TA.

In recent work [3] we investigated ways to reduce the state space caused by the unfolding. Our next steps will be the implementation of these reduction methods, and the evaluation of the complete process on industrial case studies.

ACKNOWLEDGMENT

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement N° 332830 and from the Austrian Research Promotion Agency (FFG) under grant agreement N° 838498 for the implementation of the project CRYSTAL, Critical System Engineering Acceleration.

REFERENCES


