Improved Bounded Model Checking of Timed Automata

Robert L. Smith*, Marcello M. Bersani*, Matteo Rossi†, Pierluigi San Pietro*
*Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano. Milano, Italy.
†Dipartimento di Meccanica, Politecnico di Milano. Milano, Italy.
Email: robert.smith@mail.polimi.it,
{marcellomaria.bersani, matteo.rossi, pierluigi.sanpietro}@polimi.it

***Abstract*—Timed Automata (TA) are a very popular modeling formalism for systems with time-sensitive properties. A common task is to verify if a network of TA satisfies a given property, usually expressed in Linear Temporal Logic (LTL), or in a subset of Timed Computation Tree Logic (TCTL). In this paper, we build upon the TACK bounded model checker for TA, which supports a signal-based semantics of TA and the richer Metric Interval Temporal Logic (MITL). TACK encodes both the TA network and property into a variant of LTL, Constraint LTL over clocks (CLTLoc). The produced CLTLoc formula can then be solved by tools such as Zot, which transforms CLTLoc properties into the input logics of Satisfiability Modulo Theories (SMT) solvers. We present a novel method that preserves TACK's encoding of MITL properties while encoding the TA network directly into the SMT solver language, making use of both the BitVector logic and the logic of real arithmetics. We also introduce several optimizations that allow us to significantly outperform the CLTLoc encoding in many practical scenarios.**

***Index Terms*—Formal Verification, Timed Automata, Bounded Model Checking**

I. Introduction

Timed Automata [1] (TA) are a popular tool for modeling time-sensitive systems. By combining the transition semantics of finite state automata with real-valued clocks, they are of great theoretical and practical interest for representing time-bound processes and applications. They have found common use in the domain of model checking, where system representations are evaluated against a given property of interest. Various tools and languages exist for a variety of applications and use cases. These include the current de facto standard Uppaal [2], as well as NuSMV [3].

Model Checking refers to a verification technique for solving properties of state transition systems. A wide variety of industrial applications, including circuit design, control systems, and program verification lend themselves to this representation. In the model checking process, the system is exhaustively searched to see if the given property is valid. TACK is a bounded model checker for networks of TA [4]. Properties to be verified are specified in Metric Interval Temporal Logic (MITL) [5], and are converted along with the TA network into CLTLoc [6], a variant of Linear Temporal Logic (LTL) supporting real-valued clocks.

This paper presents a novel encoding of the TA network which does not use CLTLoc as an intermediate step, instead directly transforming the network semantics into a hybrid BitVector representation. This approach has the advantage of being tailor-made for TA networks, while the previous approach relied on the general-purpose CLTLoc converter ae2sbvzot [7]. However rather than just re-create the existing encoding in a new language, we have corrected several deficiencies in the original TACK encoding, and have introduced new features to make TACK more useful for users. We have also exploited opportunities to more efficiently encode TA constructs, noticeably eliminating the need for BitVectors to track the active state of the TA, instead relying on the active transition to carry this information.

In this paper, we first present the current state-of-the-art for bounded model checking, followed by an in-depth description of both the required preliminary knowledge and the specific implementation of the TACK bounded model checker (Section II). We then introduce our novel encoding of TA networks into a form suitable for an SMT-based bounded model checker (Section III, and we present experimental results comparing the new encoding with existing ones (Section IV). Finally, we conclude with a discussion of the result and some future works (Section V).

II. Preliminaries

A. State of the Art

For many years, model checking was performed using Binary Decision Diagrams (BDDs) [8], which offer many time- and space-complexity advantages over explicit state enumeration [9]. However to efficiently handle larger state spaces, bounded model checking techniques techniques have been developed. Bounded model checking encodes the verification problem of the state transition system into a propositional satisfiability (SAT) or Satisfiability Modulo Theories (SMT) problem,

and then tasks the SAT/SMT solver with finding a valid assignment of states to time positions starting from a given initial state such that the desired property is violated (counterexample); if no such assignment is found, the property holds for the system. Because such solvers require finite state spaces, the number of time positions considered is limited by a bound k, hence the name bounded model checking. Bounded model checking analyzes traces of infinite length that can be represented in finite space. This is accomplished by limiting the search to so-called "lasso-shaped" traces. These traces begin with an initial finite sequence of states before entering an infinite loop of states. Thus only a finite number of states need to be explicitly represented by the bounded model checker, which can search for lassos of length up to the given bound.

Uppaal [2] is a de facto standard for model checking systems modeled through TA. Uppaal allows users to express properties to be checked using Timed Computation Tree Logic (TCTL), an extension of Computation Tree Logic (CTL) with real-time properties [10]. However, Uppaal and similar implementations restrict themselves to only a subset of TCTL, which focuses mostly on reachability and invariant properties.

In addition to the work done with branching-time logics, there has been interest in combining TA with the expressive power of Metric Temporal Logic (MTL), an extension of LTL with interval constraints on the 'until' operator [11]. While powerful, MTL is undecidable in general for infinite traces [12]. MITL [5] is a decidable restriction of MTL which can capture more complex properties than those supported by the subset of TCTL allowed by TA model checkers. In recent years bounded model checkers supporting MITL as property specification language have been developed, in particular MITL$_{0,\infty}$BMC [13], MightyL [14] and TACK (see [4] for a detailed comparison of the tools). In this work we improve the encoding of the TA verification problem into an SMT problem used in TACK.

B. Timed Automata

Timed Automata (TA) are a popular formalism for modeling interactions that require precise timing mechanisms [1]. In this paper, we consider an extension of TA that includes integer variables with finite ranges and mechanisms to synchronize the taking of transitions.

Let AP be a set of atomic propositions, and let Act be a set of synchronization events of the form $Act \subset \{channel \times sync\}$, where $channel$ is a finite set of symbols and $sync \in \{!, ?, \#, @\}$. In addition we define a null event τ. Act_τ is the set $Act \cup \{\tau\}$. Let X be a finite set of clocks, and Int a finite set of integer-valued variables. $\Gamma(X)$ is the set of clock constraints, where a clock constraint γ is a relation $x \sim c \mid \gamma \wedge \gamma$, where $x \in X$, $\sim \in \{<, >, \leq, \geq\}$, and $c \in \mathbb{N}$. $Assign(X)$ is the set of clock assignments, where each assignment has the form $x := 0$, where $x \in X$. $Assign(Int)$ is a set of variable assignments of the form $y := exp$, where $exp := exp + exp \mid exp - exp \mid n \mid c$, $n \in Int$ and $c \in \mathbb{Z}$. $\Gamma(Int)$ is the set of integer variable constraints, where a variable constraint γ is defined as $\gamma := n \sim c \mid n \sim n' \mid \neg\gamma \mid \gamma \wedge \gamma$, where n and n' are integer variables, $c \in \mathbb{Z}$, and $\sim \in \{<, =\}$.

A TA with variables is defined as the tuple $\mathcal{A} = \langle AP, X, Act_\tau, Int, Q, q^0, v^0_{\text{var}}, Inv, L, T\rangle$, where Q is a finite set of locations, $q^0 \in Q$ is the initial location, $v^0_{\text{var}} : Int \to \mathbb{Z}$ is a function providing initial values for each of the variables, and $Inv : Q \to \Gamma(X)$ is a function assigning each location to a (possibly empty) set of clock constraints, which are the invariants of the location. The labeling function $L : Q \to \wp(AP)$ assigns each location to a subset of the atomic propositions. Each transition $t \in T$ has the form $t = \langle Q \times Q \times Act_\tau \times \Gamma(X) \times \Gamma(Int) \times \wp(Assign(X)) \times \wp(Assign(Int))\rangle$, consisting of a source and destination location, an action, a set of clock and variable guards, a set of clocks to be reset when the transition fires, and a set of variables to assign values to. To refer to the components of a transition we will use t_- and t_+ to refer to the source and destination locations respectively, as well as $t_\epsilon, t_{\gamma_c}, t_{\gamma_v}, t_{a_c}, t_{a_v}$ to refer to the event, clock constraints, variable constraints, clock assignments, and variable assignments respectively. A transition is written as $q \xrightarrow{\gamma,\xi,\alpha,\zeta,\mu} q'$, where γ is a constraint of $\Gamma(X)$, ξ is a constraint of $\Gamma(Int)$, α is an element of Act_τ, ζ is a subset of X and μ is a set of assignments from $\wp(Assign(Int))$. Let $U(\mu)$ be the set of variables that are updated by μ—that is, that appear as the left-hand side in an assignment of μ—and let $U(t)$ indicate the set $U(\mu)$ given a transition t.

We outline the semantics of networks of TA, and we illustrate its key features through the simple example shown in Figure 1; we then show the formal definition. Transition guards are conditions over either clocks or variables that prevent the associated transition from being taken when they are not satisfied. As an example, transition t_2 can only be taken when the value of clock x is greater than 5. Assignments on the other hand modify the value of a clock or variable *after* the transition has been taken. For example, it is valid for transition t_2 to be taken when $x = 6$, even though the assignment $x := 0$ resets the value of x to 0. A transition is said to be *enabled* if the values of clocks and variables satisfy the guard, and *active* at the time when it is fired. The value is updated in the same instant as the transition, however the guards only consider the pre-transition values of the clocks when determining if the transition is valid. Variables can be assigned to any value, while clocks can only be reset to 0. When a TA is in a certain location, the

corresponding invariant (if any) is required to be true. The invariant attached to q_2 requires the TA to leave location q_2 before clock x reaches a value of 2.

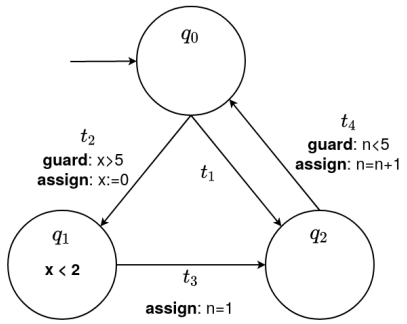

Fig. 1. A Timed Automaton with clock x and variable n.

Definition 1. *Given a TA $\mathcal{A}$, a* configuration *of $\mathcal{A}$ is a tuple (q, v_{var}, v) where q is the current location of $\mathcal{A}$ and v_{var} (resp., v) is a variable (resp., clock) valuation $Int \to \mathbb{Z}$ (resp., $X \to \mathbb{R}_{\geq 0}$).*

We adopt a semantics for TA based on so-called *signals*, where each instant of the time domain $\mathbb{R}_{\geq 0}$ comprising all nonnegative real numbers is associated with a configuration. The configuration of a TA changes when a transition is taken, but it does not change between transitions. Hence, we can split the time domain $\mathbb{R}_{\geq 0}$ into intervals during which the configuration of the TA remains the same. Figure 2 shows a fragment of an execution of the TA of Figure 1. The location is initially q_0 (in configuration c_0), then it changes to q_2 (and configuration c_1) when transition t_1 is taken. As Figure 2 shows, in the instant in which a transition is taken the configuration can be the old or the new one, depending on whether the *edge* of the transition is right-closed (]() or left-closed ()[). For example, in Figure 2 the switch from configuration c_0 to c_1 occurs in a right-closed manner, whereas the one between c_1 and c_2 in a left-closed one.

A network of TA is a finite set of TA $\mathcal{N} = [\mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_N]$. TA in the same network can refer to

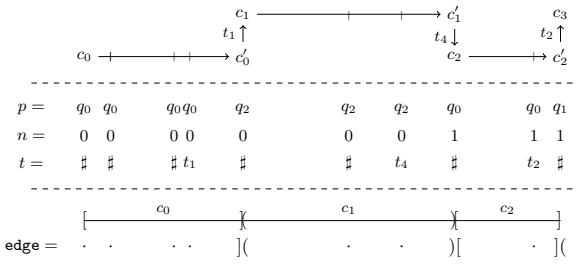

Fig. 2. Illustration of the semantics of the TA of Fig. 1.

common clocks, variables, and synchronization channels to coordinate their actions. To simplify the notation we will use the symbols T, X, Int, and Act/Act_τ to refer to the union of the respective sets of each individual TA in the network. When necessary to refer to the properties of one timed automaton in particular, we will append a numerical subscript to the set in question, for example X_i to refer to the clocks used by the specific timed automaton $\mathcal{A}_i \in \mathcal{N}$.

Before providing the formal definition of the transition relation for networks of TA, the notion of *weak* satisfaction relation $\models_w$ over clock valuations and clock constraints is introduced, where $\sim \in \{<, >, \leq, \geq\}$.

$$v \models_w x \sim d \quad \text{iff} \quad v(x) \sim d \text{ or } v(x) = d$$
$$v \not\models_w x = d \quad \text{for any } x \in X, d \in \mathbb{N}.$$

Naturally, $\models_w$ can be extended to conjunctions of formulae $x \sim d$. For instance, the formula $x < 1 \wedge \neg(y < 1) \wedge \neg(y = 1)$ is both satisfied and weakly satisfied by the clock evaluation such that $v(x) = 0.8$ and $v(y) = 1.2$, but it is only weakly satisfied if $v(x) = 1$ and $v(y) = 1$.

Definition 2. *Let $\mathcal{N}$ be a network of N TA $\mathcal{A}_1, \ldots, \mathcal{A}_N$. A* configuration *of $\mathcal{N}$ is a tuple $(\mathtt{l}, v_{\text{var}}, v)$ where $\mathtt{l}$ is a vector $[q^1, \ldots, q^N]$ such that $q^1, \ldots, q^N$ are locations of $\mathcal{A}_1, \ldots, \mathcal{A}_N$, and v_{var} (resp., v) is a variable (resp., clock) valuation for the set Int (resp., X) including all integer variables (resp., clocks) appearing in $\mathcal{A}_1, \ldots, \mathcal{A}_N$.*

When a network of TA is considered, it is possible that some automata in the network take a transition while the remaining others do not fire a transition and keep their state unchanged. Firing a transition labeled with the null event τ (i.e., a transition that does not synchronize, as explained later) is different from not taking a transition at all. Symbol _ indicates that an automaton $\mathcal{A}_i$ does not perform any transition in T_i. The notation $\mathtt{l}[i]$ indicates the location of automaton $\mathcal{A}_i$—i.e., if $\mathtt{l}[i] = j$, then automaton $\mathcal{A}_i$ is in location q_j^i, assuming that the locations of each automaton are numbered, with 0 indicating the initial one. The two kinds of configuration changes that may occur when an automaton in the network performs a transition from a location q to q' are indicated in Def. 3 with symbols `ei` (excluded-included, or left-closed) and `ie` (included-excluded, or right-closed).

Definition 3. *Let $\mathcal{N}$ be a network of N TA. Let $(\mathtt{l}, v_{\text{var}}, v)$, $(\mathtt{l}', v'_{\text{var}}, v')$ be two configurations, let $\delta \in \mathbb{R}_{>0}$ and Λ be a tuple of N symbols such that $\Lambda[k] \in \{Act_\tau \times \{\mathtt{ei}, \mathtt{ie}\}\} \cup \{\_\}$ for every $1 \leq i \leq N$. Then, a configuration change is either a transition $(\mathtt{l}, v_{\text{var}}, v) \xrightarrow{\Lambda} (\mathtt{l}', v'_{\text{var}}, v')$ or a transition $(\mathtt{l}, v_{\text{var}}, v) \xrightarrow{\delta} (\mathtt{l}', v'_{\text{var}}, v')$ defined as follows.*

1) $(\mathtt{l}, v_{\text{var}}, v) \xrightarrow{\Lambda} (\mathtt{l}', v'_{\text{var}}, v')$ *occurs if*
   a) *for each* $\Lambda[i] = (\alpha, b)$ *there is a transition* $l[i] \xrightarrow{\gamma,\xi,\alpha,\zeta,\mu} l'[i]$ *in* $\mathcal{A}_i$ *such that:*
      i) $v \models \gamma$ *and* $v_{\text{var}} \models \xi$,
      ii) $v'(x) = 0$ *holds for all* $x \in \zeta$,
      iii) $(v'_{\text{var}}, v_{\text{var}}) \models \mu$,
      iv) *when* $b = \mathtt{ei}$ *then:*
         - $v \models_w Inv(\mathtt{l}[i])$ *and*
         - $v' \models Inv(\mathtt{l}'[i])$
      v) *when* $b = \mathtt{ie}$ *then:*
         - $v \models Inv(\mathtt{l}[i])$ *and*
         - $v' \models_w Inv(\mathtt{l}'[i])$
   b) *for each* $\Lambda[i] = \_$ *it holds that:*
      i) $\mathtt{l}'[i] = \mathtt{l}[i]$;
      ii) $v \models Inv(\mathtt{l}[i])$ *and* $v' \models Inv(\mathtt{l}'[i])$.
   c) *for each clock* $x \in X$ *(resp., integer variable* $n \in Int$*), if* x *(resp.,* n*) does not appear in any* ζ *(resp., it is not assigned by any* A*) of one of the transitions taken by* $\mathcal{A}_1, \ldots, \mathcal{A}_N$*, then* $v'(x) = v(x)$ *(resp.,* $v'_{\text{var}}(n) = v_{\text{var}}(n)$*);*
2) $(\mathtt{l}, v_{\text{var}}, v) \xrightarrow{\delta} (\mathtt{l}', v'_{\text{var}}, v')$ *occurs if* $\mathtt{l}' = \mathtt{l}$, $v'_{\text{var}} = v_{\text{var}}$, $v' = v + \delta$ *and for all* $1 \leq i \leq K$, $v' \models_w Inv(\mathtt{l}[i])$.

A configuration change $(\mathtt{l}, v_{\text{var}}, v) \xrightarrow{\Lambda} (\mathtt{l}', v'_{\text{var}}, v')$, for some $\Lambda \in \{\{Act_\tau \times \{\mathtt{ei}, \mathtt{ie}\}\} \cup \{\_\}\}^K$, satisfying (1) is called a *discrete transition*. If it satisfies (2) then it is called a *time transition*. For convenience of notation, symbols $(\alpha, \mathtt{ei})$ and $(\alpha, \mathtt{ie})$, for some $\alpha \in Act_\tau$, are hereinafter denoted, respectively, with $\alpha^{)[}$ and $\alpha^{]($. The edge of a transition realized with an action α is determined by the conditions in 1(a)iv and 1(a)v and depend on the invariants of the locations involved in the transition, the clock values and the resets applied in the configuration change. Cases 1) and 2) are discussed in detail in [4].

The notions of trace and signal are now introduced.

Definition 4. *Let* $\mathcal{N}$ *be a network of* N *TA. A* trace *of* $\mathcal{N}$ *is an infinite sequence* η *of the form*

$$(\mathtt{l}_0, v_{\text{var},0}, v_0), e_0, (\mathtt{l}_1, v_{\text{var},1}, v_1), e_1, \ldots$$

such that:

1) *for all* $h \in \mathbb{N}$, $e_h = \Lambda_h$ *or* $e_h = \delta_h$;
2) *for all* $h \in \mathbb{N}$ *it holds that* $(\mathtt{l}_h, v_{\text{var},h}, v_h) \xrightarrow{e_h} (\mathtt{l}_{h+1}, v_{\text{var},h+1}, v_{h+1})$;
3) $e_0 = \delta_0$*, for some* $\delta_0 \in \mathbb{R}_{>0}$;
4) *for all* $1 \leq i \leq N$*, it holds that* $\mathtt{l}_0[i] = 0$, $v_0 \models Inv(\mathtt{l}_0[i])$*, for all* $x \in X$ *it holds that* $v_0(x) = 0$*, and for all* $n \in Int$ *it holds that* $v_{\text{var},0}(n) = v^0_{\text{var}}(n)$.
5) *discrete transitions must be followed by time transitions; that is, if* e_h *is a discrete transition (*$e_h = \Lambda_h$*), then* e_{h+1} *is a time transition (*$e_h = \delta_{h+1}$*).

Since by condition 5 there cannot be two consecutive discrete transitions, and since any finite sequence of consecutive delays $\delta_h \ldots \delta_{h+k}$, with $k \geq 0$, is equivalent to a single delay $\sum_{j=h}^{h+k} \delta_j$, a trace can always be rewritten into a new one such that discrete and time transitions strictly alternate. Moreover, by the previous property, every time transition δ_h can be replaced with a finite sequence of m pairs of time and discrete transitions $\delta_{h,0}\Lambda_{h,0}\delta_{h,1}\Lambda_{h,2}\ldots\delta_{h,m-1}$, strictly alternating, such that $\Lambda_{h,j}[i] = \_$ holds for all $0 \leq j \leq m-1$, $1 \leq i \leq N$, and $\delta_h = \sum_{j=0}^{m-1} \delta_{h,j}$.

With a slight abuse of notation, a trace is represented in the following way, where the numbering of configurations increases only after discrete transitions:

$$(\mathtt{l}_0, v_{\text{var},0}, v_0) \xrightarrow{\delta_0} (\mathtt{l}'_0, v'_{\text{var},0}, v'_0) \xrightarrow{\Lambda_0} (\mathtt{l}_1, v_{\text{var},1}, v_1) \xrightarrow{\delta_1} \ldots$$

Traces encode executions of TA by means of denumerable sequences of time and discrete transitions. However, the evolution of a network of TA is continuous, hence it is more naturally represented by means of *signals*. Intuitively, given a trace η, the projection over the real line of the values of its integer variables and atomic propositions associated with locations determines a signal M_η. To be able to consistently associate signals with traces of a TA, however, we impose the following restriction on traces.

Definition 5. *Let* $\mathcal{N}$ *be a network of TA. A trace* η *of* $\mathcal{N}$ *is* edge-consistent *if, for any configuration change* $(\mathtt{l}'_h, v'_{\text{var},h}, v'_h) \xrightarrow{\Lambda_h} (\mathtt{l}_{h+1}, v_{\text{var},h+1}, v_{h+1})$ *there are two transitions* $\mathtt{l}'_h[i] \xrightarrow{\gamma,\xi,\alpha,\zeta,\mu} \mathtt{l}'_{h+1}[i]$ *and* $\mathtt{l}'_h[\bar{i}] \xrightarrow{\bar{\gamma},\bar{\xi},\bar{\alpha},\bar{\zeta},\bar{\mu}} \mathtt{l}'_{h+1}[\bar{i}]$*, of two distinct TA* $i, \bar{i}$*, which both set the value of variable* n *(in a compatible manner), then the edge of the transitions is the same; that is, either they are* $\alpha^{]($ *and* $\bar{\alpha}^{]($*, or they are* $\alpha^{)[}$ *and* $\bar{\alpha}^{)[}$.

In the rest of the paper, only traces that are edge-consistent are considered.

Let $(\mathtt{l}, v_{\text{var}}, v)$ be a configuration; we denote as $c(\mathtt{l}, v_{\text{var}}, v)$ the pair $(\cup_{1 \leq i \leq N} L(\mathtt{l}[i]), v_{\text{var}}) \in \wp(AP) \times \mathbb{Z}^{Int}$ of the atomic propositions and variable assignments that hold in the configuration $(\mathtt{l}, v_{\text{var}}, v)$. Let η be an edge-consistent trace $(\mathtt{l}_0, v_{\text{var},0}, v_0) \xrightarrow{\delta_0} (\mathtt{l}'_0, v'_{\text{var},0}, v'_0) \xrightarrow{\Lambda_0} (\mathtt{l}_1, v_{\text{var},1}, v_1) \xrightarrow{\delta_1} \ldots$; we indicate by $\Upsilon(e)$ the "time" of a symbol e (where e can be either δ or Λ), defined as follows:

- $\Upsilon(\delta_0) = 0$;
- $\Upsilon(\Lambda_h) = \Upsilon(\delta_h) + \delta_h$ for all $h \geq 0$;

- $\Upsilon(\delta_h) = \Upsilon(\Lambda_{h-1})$ for all $h > 0$.

Finally, let $w(\eta)$ be the sequence $\Lambda_0\delta_1\Lambda_1\delta_2\ldots$.

Definition 6. *Let η be an edge-consistent trace of a network $\mathcal{N}$ of N TA. The* signal M_η *associated with η is the function $M_\eta : \mathbb{R}_{\geq 0} \to \wp(AP) \times \mathbb{Z}^{Int}$ such that:*

1) $M_\eta(0) = c(\mathtt{l}_0, v_{\text{var},0}, v_0)$;
2) *for all δ_h in $w(\eta)$, for all $r \in \mathbb{R}_{\geq 0}$ such that $\Upsilon(\delta_h) < r < \Upsilon(\delta_h) + \delta_h$ then $M_\eta(r) = c(\mathtt{l}_h, v_{\text{var},h}, v_h)$;*
3) *for all Λ_h in $w(\eta)$, $M_\eta(\Upsilon(\Lambda_h)) = (A, v_{\text{var}}) \in \wp(AP) \times \mathbb{Z}^{Int}$ where, for all $p \in AP$ and $n \in Int$:*
   a) *$p \in A$ if, for some $\alpha \in Act_\tau$ and for some $1 \leq i \leq N$:*
      - *$p \in L(\mathtt{l}_h[i])$ and $\Lambda_h[i] \in \{\_, \alpha^{](}\}$ holds, or*
      - *$p \in L(\mathtt{l}_{h+1}[i])$ and $\Lambda_h[i] = \alpha^{)[}$ holds*
   b) *$v_{\text{var}}(n) = v_{\text{var},h}(n)$ if one of the following conditions holds:*
      - *there is no transition $\mathtt{l}'_h[i] \xrightarrow{\gamma,\xi,\alpha,\zeta,\mu} \mathtt{l}_{h+1}[i]$ compatible with the configuration change and such that $n \in U(\mu)$;*
      - *there is $1 \leq i \leq N$ and a transition $\mathtt{l}'_h[i] \xrightarrow{\gamma,\xi,\alpha,\zeta,\mu} \mathtt{l}_{h+1}[i]$—compatible with the configuration change—such that $\Lambda_h[i] = \alpha^{](}$ and $n \in U(\mu)$.*
   c) *$v_{\text{var}}(n) = v_{\text{var},h+1}(n)$ if there is $1 \leq i \leq N$ and a transition $\mathtt{l}'_h[i] \xrightarrow{\gamma,\xi,\alpha,\zeta,\mu} \mathtt{l}_{h+1}[i]$—compatible with the configuration change—such that $\Lambda_h[i] = \alpha^{)[}$ and $n \in U(\mu)$ hold.*

When networks of TA are considered, the event symbols labeling the transitions are used to synchronize automata. Two (or more) different TA can take their transitions at the same time by labeling them with the same synchronization channel, and using the actions to describe the type of synchronization desired. Every event symbol $\alpha \in Act$ is associated with one communication channel, which can be identified with the event symbol itself—i.e., channel α. The first type of synchronization is one-to-one synchronization. A transition labeled with one-to-one send $\alpha!$, for some channel α, can only be fired if at the same moment in time, another TA takes a transition labeled with the one-to-one receive $\alpha?$. The second type of synchronization available is termed 'broadcast' synchronization. Like one-to-one synchronization, for a given channel α there can only be one active transition with the broadcast-send $\alpha\#$, however the difference is that there can be 0, 1, or multiple automata that sync using broadcast-receive $\alpha@$ at once. In addition, each automaton is *required* to perform a broadcast-receive if it is able to, meaning that there exists a transition t such that t_- is the currently active state, and all guards of the transition are satisfied. The details of the semantics of synchronizations can be found in [4].

C. Constraint LTL over clocks

Constraint LTL over clocks (CLTLoc) is an extension of LTL where formulas are defined over atomic propositions and clocks. A clock is a variable over $\mathbb{R}_{\geq 0}$ whose value changes between positions in a CLTLoc model to represent the passage of time. In addition, CLTLoc has been extended to support expressions over arithmetical variables [15].

A formula in CLTLoc consists of atomic propositions, clock formulas, and formulas over integer variables, which are combined using the standard LTL operators of $\mathcal{X}$ (next) and $\mathcal{U}$ (until), as well as the derived operators $\mathcal{G}$ (globally), $\mathcal{F}$ (future), and $\mathcal{R}$ (release). A clock formula compares the value of the clock to a given natural number, for instance $x > 7$. A variable formula, on the other hand, can compare not only individual variables but also arithmetic combinations of variables. An example would be the expression $b + c = 7$; $b, c \in Int$. CLTLoc uses a special version of the $\mathcal{X}$ operator that can be applied to variables in Int. A valid formula is, for instance, $\mathcal{X}(n) = n + 1$.[1]

Let X be a finite set of clocks and Int be a finite set of integer variables. CLTLoc formulas are defined as follows:

$$\phi := \pi \mid x \sim c \mid \exp_1 \sim \exp_2 \mid \mathcal{X}(n) \sim \exp \mid \phi \wedge \phi \mid \neg\phi \mid \mathcal{X}\phi \mid \phi\mathcal{U}\phi$$

where $\pi \in AP$, $x \in X$, $c \in \mathbb{N}$, $n \in Int$, $\sim \in \{<, =\}$ and exp are arithmetic formulas over integer variables and integers (defined in Section II-B).

Like in TA, clocks are special dense variables over $\mathbb{R}_{\geq 0}$ that "progress" between different positions along a CLTLoc model: each clock must either increment between two adjacent time positions, or it must be reset. We introduce $\delta : \mathbb{N} \to \mathbb{R}_{>0}$, which measures the amount of time that elapses between two adjacent time positions. For a given clock valuation $\sigma : \mathbb{N} \times X \to \mathbb{R}_{\geq 0}$, each clock $x \in X$ must either obey the equivalence $\sigma(l, x) + \delta(l) = \sigma(l+1, x)$, or is reset, i.e. $\sigma(l+1, x) = 0$ holds. We also define variables via the assignment function $\iota : \mathbb{N} \times Int \to \mathbb{Z}$ that assigns a value to each variable $n \in Int$ at every time position in $\mathbb{N}$. The arithmetical expressions exp can now be evaluated at a time position l by replacing every occurrence of an integer variable n with $\iota(l, n)$.

For the sake of space, we do not provide in this paper the full formalization of the semantics of CLTLoc and we refer the reader to [4], instead.

[1] It is easy to see that TA with variables and CLTLoc as defined in Sec. II-B and II-C are undecidable, unless suitable restrictions are introduced. In this paper we consider variables with finite domains.

D. TACK CLTLoc-based Translation

The TACK [4] tool allows users to perform the formal verification of TA against properties expressed in Metric Interval Temporal Logic (MITL, [5]). To this end, TACK takes as input a TA network $\mathcal{N}$ and a MITL formula ϕ to be checked, transforms both of them into suitable CLTLoc formulas, and uses the Zot tool, which supports the formal verification of CLTLoc formulas through a Bounded Satisfiability Checking approach [7], to automatically verify whether property ϕ holds for $\mathcal{N}$ or not. In the rest of this section we provide an overview of the TA-to-CLTLoc translation performed by TACK, which is the subject of the improvements presented in Section III. Notice that, instead, the encoding of MITL properties is done following the approach defined in [16], which was also applied in [4].

In [4], the CLTLoc formula constructed from a given network of TA represents the evolution of the configuration (i.e., an execution) of the network over the continuous time. As mentioned in Section II-B, we adopt a semantics of TA based on signals, and a configuration captures the value of all the clocks, variables and current locations of the TA in the network in a specific time instant. The key aspect of the encoding described in [4] and [16] is that every time position of a model satisfying the CLTLoc formula is representative for a nonempty interval of $\mathbb{R}_{\geq 0}$. In addition, in [4], every CLTLoc model satisfying the formula encoding a network $\mathcal{N}$ represents an execution of the network, i.e., a trace η of $\mathcal{N}$, and hence, it is an exact representation of signal M_η.

The most relevant part of the CLTLoc formula is the encoding of the firing of transitions and the possible synchronization among them, which precisely capture the dynamics of the variables, clocks and location changes between any two adjacent time intervals over the signals.

At every time position of the CLTLoc model, function $p[i]$ (with $i \in [1, N]$) represents the placing of $\mathcal{A}_i$, i.e., the active location in the corresponding interval of the execution of the TA $\mathcal{A}_i$ of network $\mathcal{N}$, and function $t[i]$ represents the transition that will be taken at the end of the interval (as depicted in Figure 2 for a single TA). Each function is syntactic sugar for a finite set of atomic propositions, which encode the value of a single variable—e.g., the current location of an automaton—belonging to finite set of values. When a transition is taken, proposition $edge_i^{RC}$ represents the edge (](or)[) with which it is taken—i.e., if the current interval of the signal associated with the i-th automaton is left- or right-closed. TA clocks and variables can be represented directly as CLTLoc clocks and variables.

Table I contains a snippet of the formulas used to encode executions of networks of TA into CLTLoc. The formulas use propositions $p[i]$, $t[i]$ and $edge_i^{RC}$ introduced above. Notice that not every TA needs to transition at each time position (for example, at a given point in the execution $\mathcal{A}_1$ might change location, whereas $\mathcal{A}_2$ does not take any transition). Hence, the encoding introduces a *null transition* symbol $\sharp$ to represent the situation in which no transition is taken. So, function $t[i]$ is equal to either a transition or the symbol $\sharp$ (see Figure 2). If transition t is active in a given position l of an execution for automaton $\mathcal{A}_i$, then the TA is in location t_- in position l, and in location t_+ at the next one (i.e., in interval $l + 1$).

Formula φ_6 defines the semantics for the null transition. If TA $\mathcal{A}_i$ performs a null transition, the state invariant must hold both before and after clock resets are applied. Function r_1 replaces the value of any reset clock with 0, thus capturing the post-reset value of any clock used in the invariant.

Formula φ_7 encodes the discrete transitions. Each must respect the guards and assignments of the transitions, the TA must currently be in the source location of the transition, and must be in the destination location in the following position. Formulas φ_{γ_c}, φ_{γ_v}, φ_{α_c}, φ_{α_v} capture the guards and assignments associated with the transition. φ_{edge} encodes the two possible edge configurations, right- and left-closed, and ensures that the invariants are satisfied depending on the edge type.

The encoding includes many other formulas, for example to define the initial values of variables and clocks, or the sufficient conditions for a transition to be taken, but they are not shown here for the sake of brevity. For the same reason we do not show here the CLTLoc formulas capturing the synchronization mechanisms among the TA of a network and those related to various liveness constraints supported by the TACK tool. Interested readers can refer to [4] for further details.

III. Improved Encoding

In the TACK tool, the CLTLoc formulas produced through the encoding presented in Section II-D are fed to the Zot formal verification tool, which in turn suitably translates them into the input logics (and in particular BitVector logic) of Satisfiability Modulo Theories (SMT) solvers. In this section we present a novel method—named TA2SMT—for encoding executions of networks of TA into the logics supported by SMT solvers. The method skips the intermediate CLTLoc representation to directly produce formulas of BitVector logic to be fed to SMT solvers. This direct translation allows us to make several optimizations not possible in CLTLoc. As before, the MITL property will continue to be converted first into CLTLoc before being transformed into BitVector logic by TACK through Zot. We use a consistent naming convention for the atomic propositions to ensure that the two BitVector encodings (the one for TA and the one for MITL formulas) can be safely combined to

TABLE I
SNIPPET OF TACK ENCODING OF A TA IN CLTLOC

$$\varphi_6 := \bigwedge_{\substack{i\in[1,N]\\ q\in\mathcal{Q}_i}} \Big(\Big(p[i] = q \wedge t[i] = \sharp \Big) \rightarrow \mathcal{X}\Big(Inv(q) \wedge r_1(Inv(q)) \Big) \Big)$$

$$\varphi_7 := \bigwedge_{\substack{i\in[1,N]\\ t\in T_i}} t[i] = t \rightarrow \Big(p[i] = t_- \wedge \mathcal{X}(p[i] = t_+) \wedge \varphi_{\gamma_c} \wedge \varphi_{\gamma_v} \wedge \varphi_{\alpha_c} \wedge \varphi_{\alpha_v} \wedge \varphi_{edge}(t_-, t_+, i) \Big)$$

produce the final SMT output. This section first describes the various terms that make up our TA network, and discusses how they are encoded into BitVector logic. Then, it overviews of the constraints, defined over the terms previously defined, that capture the TA semantics. Finally, it provides an argument for the correctness of the new encoding, and highlights the improvements made over the original encoding. For ease of reading we will refer to the old encoding as TA2CLTLOC when contrasting it with TA2SMT. For the sake of space, this paper does not present the full TA2SMT encoding; interested readers can refer to [17] for further details.

A. BitVector-based representation of terms

Our novel encoding (TA2SMT) is based on the idea of directly representing the terms of the TA into BitVectors. Since we are using a *bounded* verification approach, our goal is to represent the terms over a finite number $k+2$ of discrete positions. Using BitVector logic, we can group logically connected propositions into a BitVector, which results in a more compact encoding and can grant significant speedups on operations performed over every element of the vector.

1) Transitions: Before describing the BitVector terms for the transitions, we must make one key change to our set of transitions. For reasons to be discussed we wish to represent the *null transition* (when a TA does not transition between time positions) not as the separate entity $\sharp$, but rather as a set of $|Q_i|$ transitions, one for each location $q \in Q_i$.

$$\bigwedge_{i\in[1,N]} \bigwedge_{q\in Q_i} t_{null_q} := <q, q, \tau, \varnothing, \varnothing, \varnothing, \varnothing>$$

These null transitions have the same source and destination location, and no constraints or assignments. We can now refer to the set of all transitions as $\mathcal{T}$, defined as $\mathcal{T}_i = \bigcup_{q\in Q_i} \{t_{null_q}\} \cup T_i$ for each TA $\mathcal{A}_i$. As before $\mathcal{T}$ is the union of the $\mathcal{T}_i$ sets. The motivation for this redefinition will become clear when we discuss the encoding of the active locations of the TA.

To encode $\mathcal{T}_i$, we adopt a similar approach as the one used in TA2CLTLOC. Rather than store each transition as a separate BitVector, since only one transition is active at a time in automaton $\mathcal{A}_i$, we store the currently active transition as a binary number over $\lceil \log_2 |\mathcal{T}_i| \rceil$ bits. Therefore, we create $\lceil \log_2 |\mathcal{T}_i| \rceil$ BitVectors $tb_{i,0}, tb_{i,1}, \ldots, tb_{i,\lceil \log_2 |\mathcal{T}_i| \rceil - 1}$ of length $k+2$, each one representing a single bit of a numeric identifier that encodes the transitions in $\mathcal{T}_i$, i.e., the l-th bit of vector $tb_{i,j}$ is the j-th digit (weight 2^j) of the binary number which indicates the active transition of $\mathcal{A}_i$ at the time position l (see [7] for details about the principles behind bounded BitVector-based encodings). For the sake of convenience, to easily identify the time position in which a transition is taken, we associate every transition $t \in \mathcal{T}_i$ with a BitVector, whose l-th bit has the value of 1 if t is active at position l (firing occurs at $l+1$). The vector is determined using bit-wise logical operations over $tb_{i,0}, tb_{i,1}, \ldots, tb_{i,\lceil \log_2 |\mathcal{T}_i| \rceil - 1}$. For example, suppose that a transition $t \in \mathcal{T}_i$ is active at positions 1 and 3 of a bounded sequence such that $k = 8$. That information is represented by BitVector $\overleftarrow{0000001010}$ of length $k+2$. Now, consider $\lceil \log_2 |\mathcal{T}_i| \rceil = 6$ and a transition $t \in \mathcal{T}_i$ whose identifier is 5. Since the binary representation of 5 is 000101, we express the CNF representation (maxterm) for the value 5 with BitVector variables $tb_{i,j}$ and construct formula

$$!\overleftarrow{tb}_{i,5} \;\&\; !\overleftarrow{tb}_{i,4} \;\&\; !\overleftarrow{tb}_{i,3} \;\&\; \overleftarrow{tb}_{i,2} \;\&\; !\overleftarrow{tb}_{i,1} \;\&\; \overleftarrow{tb}_{i,0}.$$

that defines a BitVector of length $k+2$ such that the l-th bit is 1 if transition $t = 5$ is active at time position l

Transition	Alias
$\mathcal{T}_i[0]$	$!\overleftarrow{tb_{i,\lceil \log_2 \mathcal{T}_i \rceil - 1}}$ & $\ldots$ & $!\overleftarrow{tb_{i,1}}$ & $!\overleftarrow{tb_{i,0}}$
$\mathcal{T}_i[1]$	$!\overleftarrow{tb_{i,\lceil \log_2 \mathcal{T}_i \rceil - 1}}$ & $\ldots$ & $!\overleftarrow{tb_{i,1}}$ & $\overleftarrow{tb_{i,0}}$
$\mathcal{T}_i[2]$	$!\overleftarrow{tb_{i,\lceil \log_2 \mathcal{T}_i \rceil - 1}}$ & $\ldots$ & $\overleftarrow{tb_{i,1}}$ & $!\overleftarrow{tb_{i,0}}$
$\vdots$	$\vdots$
$\mathcal{T}_i[\lvert\mathcal{T}_i\rvert]$	$\overleftarrow{tb_{i,\lceil \log_2 \mathcal{T}_i \rceil - 1}}$ & $\ldots$ & $(\sim \overleftarrow{tb_{i,1}})$ & $(\sim \overleftarrow{tb_{i,0}})$

TABLE II
CONSTRUCTION OF THE TRANSITION ALIASES

We use expression such as the one above to define aliases for the $|\mathcal{T}_i|$ transitions of TA $\mathcal{A}_i$, as shown in Table II, such that each transition is identified by means of a unique alias, i.e., each transition is encoded as a unique combination of the $tb_{i,j}$ vectors. We indicate the alias for a transition t whose identifier is h as $\overleftarrow{t}$ or $\overleftarrow{\mathcal{T}_i[h]}$,

depending on the case. Because the exact value of $|\mathcal{T}_i|$ is variable, for the last transition in the table we use the symbol $\sim$ to signal that whether or not the BitVector is negated depends on the exact value of $|\mathcal{T}_i|$.

Consider now transition edges. We introduce a BitVector $\overleftarrow{edge_i^{RC}}$, $i \in [1, N]$ of length $k+2$ for each TA in the network. When a bit is set to 1 (resp., 0), it signifies that the active transition for the TA at that time position is right-closed (resp., left-closed).

2) Location: For every location $q \in Q_i$, we introduce an alias defining a vector of $k+2$ positions that indicate if the current location of automaton $\mathcal{A}_i$ is q. Since the active location of $\mathcal{A}_i$ is the source location t_- of the active transition, we define location $\overleftarrow{q}$ as the bit-wise disjunction of all the transitions whose source is q.

$$\bigvee_{q \in Q_i} \overleftarrow{q} := \bigvee_{t \in \mathcal{T}_i | t_- = q} \overleftarrow{t}$$

This is made possible by our addition of $|Q_i|$ null transitions, one for each location. This was not possible in TACK's CLTLoc encoding because of the use of a single null transition per automaton. When the CLTLoc null transition is active, it is not possible to determine the active locations without referring to variable $p[i]$.

3) Variables: Unlike location and transitions, the possible values of a bounded integer variable are not unrelated objects in a set, but their value must respect the operations of addition and subtraction. For each variable $n \in Int$ we construct a bit representation $\overleftarrow{vb_{n,j}}$, where each BitVector has length $k+2$. The values are encoded in twos complement notation, and the number of BitVectors is chosen so that the vectors are capable of representing the entire range of values for the given bounded integer variable. We will define $\lambda(n)$ as the number of bits needed for each variable n.

To refer to the complete value of a variable at a particular time position, rather than a particular bit of the variable, we make use of the *extract* and *concat* BitVector logic operators to define a second set of BitVectors $\overleftarrow{var_n(l)}$ of $\lambda(n)$ bits, defined over the vectors $\overleftarrow{vb_{n,j}}$ that represents the value of variable n at time position l, with $0 \leq l \leq k+1$.

4) Clocks: Our encoding of the clocks does not differ from TA2CLTLOC. Each clock $x \in X$ is defined as a function $x(l)$ that takes an integer argument l corresponding to a time position and returns a real number representing the value of x at position l.

5) Complete Encoding of Terms: A valid trace of the network consists of assigning values to the terms described above. To build valid traces, we define a number of constraints that make use of two helper terms, δ and $\overleftarrow{loop}$. The first one represents the amount of time that passes between two adjacent time positions (i.e., the length of the corresponding interval), and must be a positive real number. The second, the term $\overleftarrow{loop}$, has a value equal to the index of the first time position in the loop portion of the trace. From these we can represent any valid lasso-shaped trace of the network of length $k+2$, as typically done in bounded verification approaches (see also [7]). In particular, a constraint limits the position of the loop to be a positive value bounded by k. The constants 0 and k are encoded using BitVectors of length $k+2$. For instance, the value 4 over 5 bits would be written as $\overleftarrow{4}_{[5]}$ (in this case we use the subscript to make the length of the BitVector explicit) and expands to 00100. Since BitVector logic supports arithmetic, the relation $<$ can be applied to express the bounds for term $\overleftarrow{loop}$ as follows:

$$\overleftarrow{0} < \overleftarrow{loop} < \overleftarrow{k}\,.$$

In addition, we introduce aliases to more easily refer to the transitions and locations individually, and to the value of a variable at a particular time position.

B. Constraints

The terms introduced in Section III-A allow us to describe lasso-shaped traces of networks of TA, but we need to introduce suitable restrictions to avoid capturing traces that do not respect the signal-based semantics of TA. These restrictions take the form of clock guards on a transition, location invariants that prevent a TA from staying in a location indefinitely, clock progression constraints, and so on. We formalize these constraints in BitVector logic for the SMT solver to use when performing the Bounded Model Checking of TA. For brevity, in this paper we dot not present the full set of constraints; Table III shows some significant formulas, which are explained in the rest of this section to illustrate how the terms introduced above impact on the new TA2SMT encoding. Further details can be found in [17].

Formula ϕ_5 ensures that the active location of a TA correctly reflects the transition being taken. It asserts that when a transition is taken at position l, the destination location is active at position $l+1$ ($\overleftarrow{t}^{[k:0]}$ indicates that we are considering the bits of BitVector $\overleftarrow{t}$ in range $[0, k]$). Because the location BitVectors are just aliases defined over the transition BitVectors (see Section III-A2), we do not need to explicitly constrain the TA to be in location t_- at time position l, since this is true by definition.

Each transition can have multiple guards, which consist of two types, clock guards and variable guards. Formula ϕ_9 asserts that, for every clock guard, its associated transition being active at time position l implies that at the instance of transition, the relationship $\sim$ holds between the clock value and the value c. Recall that if a transition is active at position l, the transition occurs in the instant corresponding to the position $l+1$,

TABLE III
SNIPPET OF TRANSITION CONSTRAINTS FOR A NETWORK OF TA. TERMS σ AND ζ ARE BASED ON GRAMMARS PRESENTED IN SEC. II.

$\phi_5 := \bigwedge_{t\in\mathcal{T}} (\overleftarrow{t}^{[k:0]} \rightarrow \overleftarrow{t_+}^{[k+1:1]})$	$\phi_9 := \bigwedge_{t\in T} \bigwedge_{l\in[0,k]} \overleftarrow{t}^{[l]} \rightarrow \sigma_\delta(l, t_{\gamma_c})$
$\phi_{10} := \bigwedge_{t\in T} \bigwedge_{l\in[0,k]} \overleftarrow{t}^{[l]} \rightarrow \mu(l, t_{\gamma_v})$	$\phi_{11} := \bigwedge_{t\in T} \bigwedge_{x\in t_{a_c}} \bigwedge_{l\in[0,k]} \overleftarrow{t}^{[l]} \rightarrow x(l{+}1) = 0$
$\phi_{12} := \bigwedge_{t\in T} \bigwedge_{n,\exp\in t_{a_v}} \bigwedge_{l\in[0,k]} \overleftarrow{t}^{[l]} \rightarrow (\overleftarrow{var_n(l{+}1)} = \overleftarrow{\zeta(l,n,\exp)})$	
$\phi_{13} := \bigwedge_{\substack{i\in[1,N]\\ t\in T_i}} \bigwedge_{l\in[0,k]} \overleftarrow{t}^{[l]} \rightarrow \begin{pmatrix} \sigma_\delta(l, Inv(t_-)) \wedge \sigma_w(l{+}1, Inv(t_+)) \wedge (\overleftarrow{edge_i^{RC}}^{[l]} = \overleftarrow{1}) \\ \sigma_{w\delta}(l, Inv(t_-)) \wedge \sigma(l{+}1, Inv(t_+)) \wedge (\overleftarrow{edge_i^{RC}}^{[l]} = \overleftarrow{0}) \end{pmatrix} \vee$	
$\sigma(l,\gamma_c) := x(l) \sim c \mid \sigma(l,\gamma_c') \wedge \sigma(l,\gamma_c'')$	$\sigma_\delta(l,\gamma_c) := x(l) + \delta(l) \sim c \mid \sigma_\delta(l,\gamma_c') \wedge \sigma_\delta(l,\gamma_c'')$
$\sigma_w(l,\gamma_c) := x(l) \sim_w c \mid \sigma_w(l,\gamma_c') \wedge \sigma_w(l,\gamma_c'')$	$\sigma_{w\delta}(l,\gamma_c) := x(l) + \delta(l) \sim_w c \mid \sigma_{w\delta}(l,\gamma_c') \wedge \sigma_{w\delta}(l,\gamma_c'')$
$\mu(l,\gamma_v) := \overleftarrow{var_n(l)} \sim \overleftarrow{c} \mid \overleftarrow{var_n(l)} \sim \overleftarrow{var_{n'}(l)} \mid \neg\mu(l,\gamma_v') \mid \mu(l,\gamma_v') \wedge \mu(l,\gamma_v'')$	
$\zeta(l,n,\exp) := \overleftarrow{var_n(l)} \mid \overleftarrow{c} \mid \zeta(l,n,\exp') + \zeta(l,n,\exp'') \mid \zeta(l,n,\exp') - \zeta(l,n,\exp'')$	

where clock x does not have the value $x(l)$, but rather $x(l) + \delta(l)$. Note that we cannot simply use $x(l+1)$ as the value of the clock in ϕ_9, because it is possible that the transition can reset x at $l{+}1$, with $x(l{+}1) = 0$ being the post-transition value. The guard only sees the pre-transition value of the clock, thus we must explicitly add $\delta(l)$ to $x(l)$. The term $\sigma_\delta(l, t_{\gamma_c})$ is the encoding of clock constraint t_{γ_c} expressed at position l and considering the time delay between position l and $l+1$ stored in δ.

Formula ϕ_{10} captures the same semantics for variable guards, asserting that an active transition implies that the variable guard is true at that time position. Because variables, unlike clocks, do not progress with time, it is sufficient to simply use the value $var_n(l)$ to determine if the guard is satisfied. The function μ is used to encode the variable constraint grammar. If the form $\overleftarrow{var_n(l)} \sim \overleftarrow{var_{n'}(l)}$ is used and $\lambda(n') < \lambda(n)$, then $\overleftarrow{var_{n'}(l)}$ is implicitly sign-extended to a length of $\lambda(n)$ bits.

Formula ϕ_{11} models clock assignments, that are more straightforward than the clock guards. It is enough to require that if a transition is taken at time position l, then in the following time position the clock is reset.

Formula ϕ_{12} captures the semantics of variable assignments. Variable assignments can refer to both constant values and the values of other variables, and they may combine them using the operators $\{+, -\}$. To implement this in our BitVector logic, we require that if any variable n' appears in the assignment expression of variable n, then $\lambda(n') \leq \lambda(n)$ holds. We can then cast all constants and variables to BitVectors of length $\lambda(n)$, sign-extending shorter values to a length of $\lambda(n)$ bits if necessary. This allows us to use the standard BitVector addition and subtraction operators to compute the final value, which is assigned to v at time position $l{+}1$. The term $\zeta(l, n, \exp)$ encodes the expression $\exp$ with the values of arithmetical variables at position l.

Formula ϕ_{13} captures the semantics of location invariants. Although invariants are location-specific, not transition-specific, since locations are defined by the active transitions, it is sufficient to ensure that at the moment of transition both the source and destination invariants are satisfied, taking into account the value of $\overleftarrow{edge_i^{RC}}$. Since all invariants are convex, if the invariant is satisfied at moment the TA enters the location and at the moment it leaves, it is satisfied at all positions in the interval between them. The occurrence of a transition at position l implies one of two statements, one for each possible value of $\overleftarrow{edge_i^{RC}}$. In both the statements, the invariants of the source location are evaluated by considering the pre-transition clock values at position $l{+}1$, i.e., $x(l)+\delta(l)$, hence using the terms σ_δ and $\sigma_{w\delta}$, as the clock resets have not happened yet. Conversely, the invariants of the destination location are evaluated by considering post-transition clock values at position $l{+}1$, hence using the terms σ and σ_w. In addition, the invariant of the location (either t_- or t_+, depending on $edge_i^{RC}$) that is not the current location of the automaton at the time instant in which a transition occurs, i.e., whose signal has an open-ended edge transition, are evaluated with the weak satisfaction relation $\sim_w$ (the interested

reader can find the definition in [4]).

The complete TA2SMT encoding includes, in addition to the formulas of Table III (which we can conjoin in a single formula, ϕ_{trans}), formulas that govern the initialization and progression of the TA (ϕ_{init}), formulas that capture the semantics of synchronizations (ϕ_{sync}), and formulas that guarantee the correctness of the lasso-shaped traces (ϕ_{loop}). Overall, the encoding of the semantics of a network of TA $\mathcal{N}$ is given by the following formula (we refer to [17] for details):

$$\phi_{\mathcal{N}} := \phi_{init} \wedge \phi_{trans} \wedge \phi_{sync} \wedge \phi_{loop}$$

C. Equivalence and Improvements

In this section we outline an argument showing that the TA2SMT encoding given by formula $\phi_{\mathcal{N}}$ of Section III-B is a correct and complete representation of all lasso-shaped, non-Zeno runs of length $k+2$ of network $\mathcal{N}$. More precisely, we briefly compare the TA2SMT and TA2CLTLOC encodings and show that they capture the same constraints. Hence we conclude that the TA2SMT encoding is sound and complete, since TA2CLTLOC has been proved to be so in [4]. We also highlight the points in which TA2SMT improves on TA2CLTLOC.

Both the TA2CLTLOC and the TA2SMT encodings constrain the clocks, variables, and TA to their respective initial values and locations at time position 0. For variables and clocks these constraints are identical, as both assign the desired value at time position 0. For locations TA2SMT uses the $\overleftarrow{q}$ aliases to require that the TA begins in the initial location, despite not having location BitVectors. Because the location alias is only true when one of the transitions whose source is that location is true (including the location-specific null transitions), the constraint is valid. Function $\delta(l)$ ensures that all clocks progress at the same rate, while clock resets and variable assignments are only allowed if one of the corresponding transitions are active. As for the transitions, although we have broken up φ_7 (see Table I) into several pieces (some of which are shown in Table III), the functionality remains the same. We ensure that in order for a transition to be valid, its destination location must be active in the next time position, the clock and variable constraints must be satisfied, all assignments must be enforced, and the invariants of the source and destination location must be true at the moment of transition. Like TA2CLTLOC, TA2SMT allows that at the moment of transition, only one of the two invariants must be satisfied, using the concept of weak satisfaction to formalize this relaxation. Similarly, the original TA2CLTLOC encoding contains three constraints that assert that the values of the active locations, as well as the values of the variables and clocks, can only be changed if there is an active transition that modifies them. For locations, this is accomplished with ϕ_5, which requires that the active location in the following position be equal to the value of the destination location of the active transition. Unlike in the original encoding, we have one null transition for each position, so we do not need to consider the null transitions as a special case. Therefore for the location to change, there must be a non-null transition to enable the location change. A pair of formulas, ϕ_8 and ϕ_9, not shown in this paper for brevity, assert that when no transition explicitly changes the value of a variable or resets a clock, their values remain the same. Our new encoding also respects the same loop constraints as TA2CLTLOC including the clock constraints necessary to represent all possible lasso-shaped traces.

As shown in Section IV, the new TA2SMT encoding in many cases provides significant benefits in terms of efficiency of the verification procedure. In addition, TA2SMT introduces various improvements over TA2CLTLOC concerning the range of TA features captured. TA2CLTLOC contains a limitation regarding integer variables: because they are represented as elements of a set, TA2CLTLOC can only test them for equality. This means that constraints of the form $n \sim c$ or $n \sim n'$, where $\sim \notin \{=\}$ are not supported. TA2SMT correctly represents the values of the integer variables using a twos-complement encoding, and therefore can support the full grammar of variable guards and assignments. The implementation of the TA2SMT encoding in the TACK tool has also fixed some issues that were present in the old implementation of TACK, and it has allowed us to complete the set of features supported by the tool. In particular, support for broadcast synchronization primitives in the old version of the TACK tool was faulty, and it has now been fixed in the implementation of TA2SMT, as shown by our experimental results. Finally, support for right-closed intervals, left-closed intervals, and arbitrary combinations thereof was not complete in the old implementation of TACK (only right-closed intervals were fully supported); TA2SMT, instead, fully supports all types of intervals.

IV. Experimental Results

In this section we present the results of several experimental evaluations of the new TA2SMT encoding compared with TA2CLTLOC. These tests cover several different benchmarks commonly used to evaluate formal verification techniques. For both TA2CLTLOC and TA2SMT, strong transition liveness (see [4]) was used in all of the tests, and all edges were constrained to be right-closed. These were the settings used to benchmark the original TACK application, and they remain the default settings for the tool. In all of the following tests, the measured time is the combined time taken by both the TACK program to parse the problem and convert it to SMT form and for the underlying Z3 solver [18] to

decide the satisfiability of the SMT problem. In practice, the TACK translation always took less than a second. For every test, the evaluation proceeded in several rounds, each with a larger length of traces considered by TACK.

All tests were performed running the Z3 SMT solver version 4.8.8 on a server equipped with an AMD EPYC 7551 CPU (2.5 GHz) with 2 32-core sockets, 500 GB of RAM and Debian Linux (version 4.19). Although our tests were run on a large server with (at the time of writing!) an unusually high amount of both processors and RAM, the Z3 solver is a single-threaded application, and typically uses less than a gigabyte of RAM while running. Therefore, very similar results could be obtained on a machine with more reasonable resources. To reduce instabilities in the solver and to present a clearer comparison between the encodings, we used Z3's built-in 'parallel-or' solution strategy to run two versions of each test, each copy with a different random seed. The times reported here are the shortest time of two runs, as Z3 process terminates when either thread terminates.

Fischer Mutual Exclusion Protocol: The well known Fischer benchmark [19] models a protocol for ensuring exclusive access to a shared common resource that can be requested by multiple processes. The processes are identical in their behavior, aside from a numerical id, and are modeled through single TA. The protocol uses global variables in guards and assignment statements of the TA to control access. Each TA in the network has a 'critical state', and the protocol guarantees that only one TA can be in its critical state at a time.

To measure the scalability of our new encoding, we performed multiple test runs while modifying the bound k and the number of processes that are attempting to execute their critical region. Several MITL properties, which are the same as in [4], were verified. Liveness property 1 (*live1*) requires that once process 1 enters state b, in which it sets shared variable id, it always transitions to the 'waiting' state c. Property 2 (*live2*) is similar, but it contains the additional constraint that process 1 must complete the transition to state c in at most 3 seconds. Property *live3* has a similar time bound, but requires that process one move to the critical section cs rather than c within the time bound, which we expect to not be universally true (a process can return to state b after moving to state c if another process has reset the variable id). Properties *live4* and *live5* are copies of properties *live2* and *live3*, respectively, with the sole difference of inclusion vs. exclusion at the boundaries of the interval. Property *safe* seeks to prove the "safety" of the protocol, namely that two distinct processes are never in the critical section at the same time.

Figure 3 shows the results of the comparison between TA2SMT and TA2CLTLOC. The table shows the time (sec.) that the fastest tool takes to solve the instance.

	2	3	4	5	6	7	8	9	10
live1-10	0,8	0,9	0,9	1	1	1,2	1,1	1	1,2
live1-15	0,8	0,9	1	1	1,1	1,2	1,2	1,2	1,3
live1-20	0,9	1	1,1	1,2	1,2	1,5	1,6	1,5	1,4
live1-25	1	1	1,2	1,2	1,3	1,5	1,8	1,7	2,7
live1-30	1,1	1,1	1,3	1,3	2,2	2,3	1,8	2,1	2,3
live2-10	1,4	1,6	1,7	2	2	2,4	2,5	2,7	3,1
live2-15	4,5	5,3	6	6	9,1	12,4	34,5	28,6	146,5
live2-20	10,2	13	14,4	21,6	27,4	28,8	56,8	72,3	108,6
live2-25	20,9	28,9	25,5	47,6	52,7	50,6	122,3	235,3	106,6
live2-30	36,1	37,9	56,9	63,8	87,2	123,7	198,4	146,9	283,4
live3-10	0,9	1	1,4	1,7	2,4	13,2	19,5	16,1	16,1
live3-15	1	1,3	2,2	2	3,6	11,9	10	23	26,4
live3-20	1,1	1,8	2,1	4,2	8	10,4	23,5	24,6	53,9
live3-25	2	2,3	3,7	5,9	9,5	24,1	90,8	647,5	467,8
live3-30	1,5	4,6	4,6	8,8	17,3	24,4	30,7	73,2	93,2
live4-10	1,2	1,3	1,5	1,7	1,9	2,1	2,1	2,4	2,3
live4-15	2,5	3,2	4,2	4,9	6,1	9	10,6	26,7	130,9
live4-20	6,1	6,1	7,3	12,9	13,8	20,7	30	30,9	56,9
live4-25	10,3	15,4	16,7	19,8	24,1	39,1	60,6	232,2	108,8
live4-30	27,5	21,8	33,9	44,2	67	67,7	133,5	156,4	161
live5-10	1	1,3	1,5	1,8	4,2	13,7	14,1	19,1	21
live5-15	1,1	1,7	2,5	2,3	5,2	5,7	18	20,6	29,1
live5-20	1,4	2,2	2,6	5,2	8,6	15,2	21,6	22	56,9
live5-25	2,1	3,6	4,4	10	10,6	17,2	44	259,8	445,3
live5-30	2,2	6,9	5,6	14,9	21,1	26,7	28,8	80	116,7
safe-10	0,9	1,1	1,9	2,5	3,1	4,7	10,1	7,9	15
safe-15	1	1,8	4,4	10	33,9	51,2	84	110,9	191,5
safe-20	1,5	3,9	10,2	21,6	93,3	234,8	670,4	1000,4	6662,7
safe-25	1,7	8,9	27,6	83,5	188,1	596,2	2469,2	5365	
safe-30	2,5	16,3	54,4	263,4	993,9	3354,1			

Fig. 3. Results of the comparison between TA2SMT and TA2CLTLOC on the Fischer protocol. Each column corresponds to a different number of processes involved in the Fischer protocol that share a common resource. The numbers appearing as suffixes in the property names (e.g., *20* in *live1-20*) indicate the length of the (lasso-shaped) traces considered by TA2SMT.

The color indicates how this time compares with that of TA2SMT. If TA2SMT is the fastest tool of the two, the cell is colored green, and the shade of green indicates how much faster TA2SMT is compared to TA2CLTLOC (dark green means $>$ 2x speedup, light green means between 1.05x and 2x speedup). Otherwise, the cell is colored orange when TA2SMT is between 1.05x and 2x slower than TA2CLTLOC, and red when it is more than 2x slower (if the difference between TA2SMT and TA2CLTLOC is less than 5% either way, the cell is left white). Figure 4 shows the speed up/slow down factor for each experiment with the Fischer protocol. Empty cells indicate a timeout for both tools, set at 2 hours. For property *safe*, TA2CLTLOC is the fastest tool. Indeed, TA2SMT is consistently faster than TA2CLTLOC for greater values of the bound k and higher numbers of processes, except for property *safe*. This property is peculiar in that the MITL formula grows in size with the number of TA in the network. It is possible that at larger sizes, the MITL encoding becomes a bottleneck

	2	3	4	5	6	7	8	9	10
live1-10	1.125	1.000	1.111	1.000	1.200	1.000	1.364	6.000	1.417
live1-15	1.125	1.000	1.100	1.400	1.455	1.417	1.250	11.833	1.615
live1-20	1.111	1.100	1.182	1.833	1.583	1.733	1.500	2.400	2.786
live1-25	1.100	1.600	1.250	1.750	146.000	2.267	2.722	2.647	1.704
live1-30	1.000	1.364	1.538	2.385	2.909	1.957	2.722	2.238	2.087
live2-10	0.875	0.889	0.944	0.800	0.667	0.686	0.833	0.628	0.775
live2-15	0.938	1.189	0.909	1.617	1.648	2.177	1.403	2.357	0.638
live2-20	1.059	0.985	2.604	1.023	2.066	1.278	0.928	7.855	14.633
live2-25	1.560	1.360	2.247	2.769	1.943	46.822	13.200	5.091	59.209
live2-30	1.175	2.129	2.244	0.804	13.178	1.018	Inf	Inf	Inf
live3-10	1.111	1.100	1.143	2.118	3.333	0.539	0.750	1.093	1.658
live3-15	1.200	1.000	1.136	0.833	5.139	2.319	3.060	2.583	1.561
live3-20	1.091	0.857	0.636	0.778	0.889	0.759	1.017	1.346	1.200
live3-25	0.870	1.000	0.446	0.831	0.646	0.923	7.506	1.163	1.768
live3-30	0.789	1.000	0.780	0.889	1.890	0.755	0.821	1.404	1.611
live4-10	1.000	0.929	0.938	0.773	0.905	0.875	0.724	0.960	0.852
live4-15	1.160	1.125	1.190	1.633	1.836	2.178	3.198	2.801	0.456
live4-20	1.082	2.541	1.973	1.372	5.630	4.164	6.117	16.320	26.654
live4-25	0.579	1.156	1.371	2.298	6.282	63.494	39.703	28.331	Inf
live4-30	0.658	1.202	0.899	1.267	0.959	3.207	12.863	1.864	2.873
live5-10	1.100	1.000	1.267	0.429	1.286	0.797	1.702	1.199	1.129
live5-15	1.182	1.000	1.000	0.697	1.019	0.704	1.128	Inf	1.265
live5-20	1.214	0.710	0.565	0.703	0.796	0.899	1.380	3.236	Inf
live5-25	1.000	1.000	0.978	1.350	0.576	0.610	17.757	3.141	1.279
live5-30	0.880	0.622	0.629	0.745	0.925	0.678	0.436	1.758	2.045
safe-10	0.900	0.786	0.864	0.391	0.237	0.143	0.373	0.556	1.080
safe-15	0.769	0.600	0.463	0.410	0.552	0.535	0.492	0.320	0.352
safe-20	1.000	0.476	0.190	0.152	0.254	0.310	0.353	0.356	
safe-25	0.739	0.197	0.225	0.116	0.118	0.177			
safe-30	0.806	0.164	0.124	0.320	0.365				

Fig. 4. Speedup/slow down between TA2SMT and TA2CLTLOC on the Fischer protocol.

that limits the utility of further TA optimizations.

Gearbox: The Gearbox TA models an automatic gearshift which utilizes a gearbox controller [20]. Upon receiving a gear change (reverse, neutral, as well as gears 1-5 are modelled), the controller coordinates changes to the state of the engine, gearbox, and clutch to perform the desired gear transition. Property 0 asserts that in the absence of any errors, the elapsed time required to change gears after an input is no greater than 1500 ms. Property 1 similarly asserts that for certain specific gear transitions, the absence of errors implies a transition time of at most 1000 ms. Property 2 concerns error propagation from the clutch and gearbox to the gear controller. Depending on the specific error, the gear controller is required to respond accordingly within 200 or 350 milliseconds. Property 3 also concerns error states in the controller. It asserts that each error state in the controller is active only when the related error has occurred in either the clutch or the gearbox. Thus, the controller never reports a false error. Property 4 asserts that whenever the gearbox is not in neutral and no gear shift is occurring, the engine module is supplying torque to the rest of the drive system. These properties were evaluated over the gearbox model using time bounds

	10	15	20	25	30	35	40	45	50
Prop0	2.4	6	22.6	136.3	1943	16573			
Prop1	2.1	3.5	19.7	101.7	286.7	1288.3	6355	30088	
Prop2	3.6	8.1	15.6	29.1	41.9	67.6	78.2	101	171.7
Prop3	1.7	3.2	8.7	18.5	31	49.3	73.7	85	80.8
Prop4	1.9	3.5	5.8	12.5	20.9	28.6	28.9	83.1	51.1

Fig. 5. Results of the comparison between TA2SMT and TA2CLTLOC on the Gearbox model.

	10	15	20	25	30	35	40	45	50
Prop0	1.0833	0.6452	0.2417	0.1692	0	0			
Prop1	1.1905	2.2	1.4619	1.0315	0.787	0.5868	0.2873	0	
Prop2	1.1667	0.8804	0.7464	1	1.4988	0.9883	2.0537	1.7941	1.4403
Prop3	1.4118	1.3438	1.1034	1.0162	0.9091	1.1075	1.0326	1.4388	0.4432
Prop4	1.2632	1	0.6744	0.5274	1.2297	1.8601	3.3841	1.1649	2.8102

Fig. 6. Speedup/slow down between TA2SMT and TA2CLTLOC on the Gearbox model.

between 10 and 50 steps. Figure 5 reports, for each instance, the time (sec.) taken by the fastest tool of the two to solve the model. As for Figure 3, cells colored green (resp., red/orange) are those for which TA2SMT (resp., TA2CLTLOC) was fastest (see Figure 6 for the speed up/slow down factors).

Token Ring: The Token Ring protocol [21] models a ring of agents that pass a token between themselves, along with a process that models the ring itself. The token moves in either direction along the ring (the ring process controls the token). The agents may choose to return the token in either a synchronous or asynchronous manner. In both cases, channel-based synchronization among TA coordinates ownership of the token. The property checked asserts that agents 1 and 2 never simultaneously synchronize with the token. Figure 7 contains the results of the Token Ring tests (time in sec.), while Figure 8 shows the speed up factors.

Philips Audio Protocol: The Philips Audio Protocol models the transmission of data over a single shared bus between two entities. An interesting property is that the message can be decoded by a receiver that can only detect rising edges, that is a transition from a low to high signal over the bus. This algorithm was translated into a TA representation for the Uppal tool by Larsen et al. [22]. Property 1 expresses the correctness of the protocol,

	2	3	4	5	6	7	8	9	10
10	0.9	1	1	1	1.2	1.2	1.3	1.4	1.5
15	1	1	1.1	1.2	1.4	1.4	1.5	1.7	1.7
20	1	1.2	1.3	1.3	1.5	1.7	1.8	2.1	2.1
25	1.1	1.4	1.4	1.5	1.7	1.8	2.1	2.7	2.4
30	1.3	1.4	1.6	1.6	2.1	2	2.4	2.6	3
35	1.3	1.4	1.6	2.1	2.2	2.1	3	3	3.2
40	1.4	1.5	2.1	2.6	2.4	2.4	2.9	3.4	4.4

Fig. 7. Results of the comparison between TA2SMT and TA2CLTLOC on the Token Ring model

	2	3	4	5	6	7	8	9	10
10	1.111	1.200	1.300	1.400	1.250	1.500	1.462	1.500	1.467
15	1.600	1.500	1.455	1.500	1.357	1.357	1.533	1.471	1.706
20	2.800	2.083	1.846	1.538	1.667	1.588	1.611	1.429	1.667
25	3.000	3.357	3.214	2.267	2.059	1.833	1.857	1.556	1.833
30	3.846	5.000	4.625	5.063	3.429	2.650	1.708	2.269	2.033
35	4.615	6.429	7.000	4.762	7.864	4.857	3.133	1.900	2.031
40	7.071	7.000	6.762	6.077	7.833	5.042	4.034	3.735	2.773

Fig. 8. Speedup/slow down between TA2SMT and TA2CLTLOC on the Token Ring protocol.

k		15	25	35	40	45	50
prop.	1	5.1	155.3	3071.9	9420.8	27947.4	-
	2	3.8	9.0	29.8	29.8	42.5	65.0
	3	7.8	285.8	6440.4	21426.9	-	-

TABLE IV
TIME (SEC.) TO CHECK THE PROPERTIES OF THE PHILIPS PROTOCOL WITH TA2SMT (− MEANS NO RESULT AFTER 12 HOURS).

that with a properly functioning sender and receiver the signal will be interpreted correctly. This is represented by asserting that the receiving agent never enters an error state. Property 2 expresses that the sender will never send two rising edges within 400 units of time, regardless of the message being sent (required to ensure a 5% timing error tolerance). Property 3 expresses that when the sender has completed the message, the receiver enters the stop state within 900 time units. For this model, the use of arithmetic operations on variables made creating the model for TA2CLTLOC (which supports such operations through a workaround) difficult, so Table IV shows only the results obtained with TA2SMT on the verification of three properties.

Carrier Sense Multiple Access / Collision Detection: The CSMA/CD protocol [23] is a well known protocol for allowing multiple agents to share a communication channel, and was popularized by its inclusion in the Ethernet standard. The protocol includes one process to manage a shared communication bus, as well as a number of processes that wish to obtain exclusive access to the bus in order to send a message. When two processes attempt to send at the same time, the bus process detects the collision and uses the broadcast synchronization primitive to force the processes to wait a randomized amount of time before attempting to communicate again. The property checked asserts that after process 1 has been sending for 52 units of time, process 2 cannot begin sending until process 1 has finished. Table V shows the results of the execution of the verification runs on the CSMA/CD model using TA2SMT. In this case, a comparison with the TA2CLTLOC encoding has not been carried out, because we modified the CSMA/CD TA model to make it more accurate with respect to the real-world behavior of the protocol. This entailed using variable comparisons that are not fully supported

n		3	5	7	9	10
k	10	3.5	5.1	5.8	7.8	9.5
	15	31.2	52.7	121.7	275.4	324.0
	20	109.5	721.5	2206.1	5097.3	5068.9
	25	813.7	4772.4	–	–	–
	30	2882.0	–	–	–	–

TABLE V
TIME (SEC.) TO CHECK PROPERTY FOR THE CSMA/CD PROTOCOL WITH TA2SMT (− MEANS NO RESULT AFTER 2 HOURS).

in TA2CLTLOC, so Table V only reports executions times obtained through TA2SMT.

For the Fischer benchmark, in addition to the two-way comparison between TA2SMT and TA2CLTLOC discussed above and summarized by figures 3 and 4, we also carried out a three-way comparison between the two TACK encodings and MITL$_{0,\infty}$BMC (similarly to [4]). Figure 9 shows the results of the three-way comparison. More precisely, it shows the time taken by the fastest of the three tools, and the color of each cell represents how TA2SMT compares against the best of the other two tools: if the cell is colored green, TA2SMT was the fastest tool, otherwise the cell is colored orange/red, with the same meaning of the coloring as for figures 3 and 4. Notice that the *live3* and *live5* properties *do not hold*—i.e., a counterexample exists, as the model is *satisfiable*. In these cases, the incremental approach of MITL$_{0,\infty}$BMC, which explores the bounds k starting from 1 until it determines that the model is satisfiable, is very efficient, since it stops the search as soon as possible. Indeed, the portions of Figure 9 corresponding to properties *live3* and *live5* show that the best tool (MITL$_{0,\infty}$BMC) uses a constant time to solve the problem, even as the bound k increases (the model is satisfiable with a bound less than 10). For property *safe*, for higher bounds k, both TA2CLTLOC and TA2SMT are faster than MITL$_{0,\infty}$BMC.

Notice that, in the cases of the Gearbox and Token Ring benchmarks, we only compared TA2SMT against TA2CLTLOC, since models for the MITL$_{0,\infty}$BMC tool were not available (and building new ones was not possible, as explained in [4]).

V. DISCUSSION AND FUTURE WORKS

Empirical testing has revealed that the novel TA2SMT encoding can provide significant speedups across several benchmarks when compared to TA2CLTLOC. In particular, TA2SMT is consistently better than TA2CLTLOC in the Token Ring case, and mostly better in the Gearbox and Fischer cases, especially for increasing values of the bound and of the number of processes. These results seem to indicate that the TA2SMT encoding is better suited to exploring models with larger bounds, as the time needed to solve larger and larger bounds grows more slowly compared to TA2CLTLOC.

	2	3	4	5	6	7	8	9	10
live1-10	0.5	0.7	0.9	1	1	1.2	1.1	1	1.2
live1-15	0.8	0.9	1	1	1.1	1.2	1.2	1.2	1.3
live1-20	0.9	1	1.1	1.2	1.2	1.5	1.6	1.5	1.4
live1-25	1	1	1.2	1.2	1.3	1.5	1.8	1.7	2.7
live1-30	1.1	1.1	1.3	1.3	2.2	2.3	1.8	2.1	2.3
live2-10	0.9	1.4	1.7	1.8	2	2.4	2.5	2.7	3.1
live2-15	4.5	5.3	6	6	9.1	12.4	30.1	28.6	139.8
live2-20	10.2	13	14.4	21.6	27.4	28.8	56.8	72.3	108.6
live2-25	20.9	28.9	25.5	47.6	52.7	50.6	122.3	235.3	106.6
live2-30	36.1	37.9	56.9	63.8	87.2	123.7	198.4	146.9	283.4
live3-10	0.4	0.5	0.6	0.7	0.9	1	1.1	1.3	1.4
live3-15	0.4	0.5	0.6	0.7	0.8	1	1.1	1.3	1.4
live3-20	0.4	0.5	0.6	0.7	0.8	1	1.1	1.3	1.4
live3-25	0.4	0.5	0.6	0.7	0.8	1	1.1	1.3	1.4
live3-30	0.4	0.5	0.6	0.7	0.9	1	1.1	1.3	1.4
live4-10	0.9	1.3	1.5	1.7	1.9	2.1	2.1	2.4	2.3
live4-15	2.5	3.2	4.2	4.9	6.1	9	10.6	26.7	117.2
live4-20	6.1	6.1	7.3	12.9	13.8	20.7	30	30.9	56.9
live4-25	10.3	15.4	16.7	19.8	24.1	39.1	60.6	232.2	108.8
live4-30	27.5	21.8	33.9	44.2	67	67.7	133.5	156.4	161
live5-10	0.4	0.5	0.6	0.7	0.8	1	1.1	1.3	1.5
live5-15	0.4	0.5	0.6	0.7	0.8	1	1.1	1.3	1.5
live5-20	0.4	0.5	0.6	0.7	0.8	1	1.2	1.3	1.5
live5-25	0.4	0.5	0.6	0.7	0.8	1	1.1	1.3	1.5
live5-30	0.4	0.5	0.6	0.7	0.8	1	1.1	1.3	1.5
safe-10	0.4	0.6	0.9	1.3	1.5	2	2.4	2.7	3.2
safe-15	1	1.8	4.4	10	21.4	31.9	40.6	52.6	81.4
safe-20	1.5	3.9	10.2	21.6	93.3	234.8	491.1	760.3	1026
safe-25	1.7	8.9	27.6	83.5	188.1	596.2	2469	5365	
safe-30	2.5	16.3	54.4	263.4	993.9	3354			

Fig. 9. Results of the comparison between TA2SMT, TA2CLTLOC and MITL$_{0,\infty}$BMC on the Fischer protocol.

In addition, TA2SMT was able to solve models (the new, more realistic CSMA/CD and the Philips protocol) that were more difficult to tackle in TA2CLTLOC due to limitations in the way the old encoding deals with integer operations and with synchronizations among TA.

Future work will focus on two main objectives. Firstly, we will seek to achieve a better integration of the two translations, which will take the form of a BitVector encoding specific to MITL formulas that does not rely on the CLTLoc translation. Secondly, given the considerable impact of incremental approaches in verification we will attempt to understand how to make the BitVector-based encoding incremental.

REFERENCES

[1] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994.

[2] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," 1997.

[3] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *Computer Aided Verification*, E. Brinksma and K. G. Larsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 359–364.

[4] C. Menghi, M. M. Bersani, M. Rossi, and P. San Pietro, "Model checking MITL formulae on timed automata: A logic-based approach," *ACM Trans. Comput. Logic*, vol. 21, no. 3, pp. 1–44, 2020.

[5] R. Alur, T. Feder, and T. A. Henzinger, "The benefits of relaxing punctuality," *Journal of the ACM*, vol. 43, no. 1, pp. 116–146, 1996.

[6] M. M. Bersani, M. Rossi, and P. San Pietro, "A tool for deciding the satisfiability of continuous-time metric temporal logic," *Acta Informatica*, vol. 53, no. 2, pp. 171–206, 2016.

[7] M. M. Pourhashem Kallehbasti, M. Rossi, and L. Baresi, "On how bit-vector logic can help verify LTL-based specifications," *IEEE Transactions on Software Engineering*, pp. 1–15, 2020.

[8] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 677–691, 1986.

[9] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 1020 states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142 – 170, 1992.

[10] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[11] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi, *Modeling Time in Computing*, ser. EATCS Mon. in Theoretical Computer Science. Springer, 2012.

[12] P. Bouyer, "Model-checking timed temporal logics," *Electronic Notes in Theoretical Computer Science*, vol. 231, pp. 323–341, mar 2009.

[13] R. Kindermann, T. Junttila, and I. Niemelä, "Bounded model checking of an MITL fragment for timed automata," *Proceedings - International Conference on Application of Concurrency to System Design, ACSD*, 04 2013.

[14] T. Brihaye, G. Geeraerts, H.-M. Ho, and B. Monmege, "Mightyl: A compositional translation from MITL to timed automata," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 421–440.

[15] F. Marconi, M. M. Bersani, M. Erascu, and M. Rossi, "Towards the formal verification of data-intensive applications through metric temporal logic," in *Formal Methods and Software Engineering*, K. Ogata, M. Lawford, and S. Liu, Eds. Cham: Springer International Publishing, 2016, pp. 193–209.

[16] M. M. Bersani, M. Rossi, and P. San Pietro, "An SMT-based approach to satisfiability checking of MITL," *Information and Computation*, vol. 245, pp. 72 – 97, 2015.

[17] R. L. Smith, "Improved verification of networks of timed automata," Master's thesis, Politecnico di Milano, 2020.

[18] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[19] M. Abadi and L. Lamport, "An old-fashioned recipe for real time," *Transactions on Programming Languages and Systems*, pp. 1543–1571, 1994.

[20] M. Lindahl, P. Pettersson, and W. Yi, "Formal design and analysis of a gear controller," *International Journal on Software Tools for Technology Transfer*, vol. 3, pp. 353–368, 08 2001.

[21] R. Jain, *FDDI handbook: high-speed networking using fiber and other media*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[22] K. G. Larsen, P. Pettersson, and W. Yi, "Model-checking for real-time systems," in *Fundamentals of Computation Theory*, H. Reichel, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 62–88.

[23] "IEEE 802.3 Ethernet Working Group," http://www.ieee802.org/3.