Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Master Degree in Computer Science and Engineering

Dipartimento di Elettronica, Informazione e Bioingegneria

Improved Verification of Networks of Timed Automata

Supervisor: Prof. Pierluigi San Pietro

Tesi di laurea di:

Robert Lawrence Smith Matr. 914634

Academic Year: 2019-2020

Contents

List of Figures

List of Tables

1 Introduction

2 Preliminaries

2.1 Timed Automata

Timed Automata are a useful model for many interactions that require precise timing mechanisms. Each Timed Automaton has a set of states, one of which is active at any given moment, much like a Finite State Machine. Also like a FSM, a Timed Automaton has a set of transitions that allow it to move between different states. However these transitions come with powerful timing properties that allow for finer control over the progression of the automata. Included with our automata is a network of clocks and integer variables. Clocks progress as time passes, and can be used alongside variables as prerequisites for taking transitions and remaining in states. Transitions can also reset these clocks (and assign new values to variables), allowing for communication and shared state between different Timed Automata. For explicit synchronization, we define a set of synchronization primitives that can be used to coordinate transitions between different automata. Finally, states can be labeled with atomic propositions, to aid in defining model properties that we will then verify over the network. To help concretize this concept, we will introduce a simple Timed Automaton, which will be used throughout this paper to help visualize important concepts.

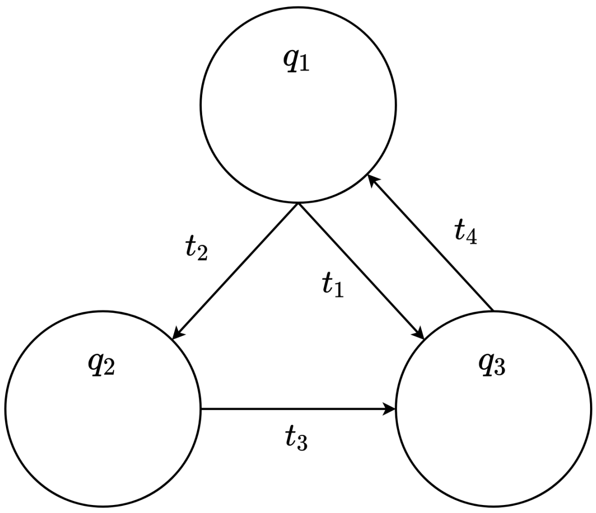

Figure 1: A basic Timed Automaton

As we can see in figure 1, Timed Automata have many similarities with Finite State

Machines and other automata. For a given automaton $\mathcal{A}_i$, the example TA consists of a finite set of states $Q_i = \{q_1, q_2, q_3\}$, and a finite set of transitions $T_i = \{t_1, t_2, t_3, t_4\}$. Like a finite state machine, at a given moment in time there is one state that is "active". The TA can take transitions that may change its state, with the condition that the source of the transition must be the currently active state. We denote the source state of a transition t as t_-, and the destination state as t_+. As an example, if the current active state is q_1, then either t_1 or t_2 can be taken. Although not shown in this example, the source and destination state of a transition may be the same state, and there may be multiple transitions with identical source states and identical destination states.

In addition to states and transitions, TA are enriched with clocks and bounded integer variables. Clocks are variables over $\mathbb{R}_{\geq 0}$ that increment with the passage of time. Their ability to continuously change value is fundamental to the ability of timed automata to model time-sentisive and real-time systems. Meanwhile the bounded integer variables do not change value on their own, and have to be explicitly modified by the TA.

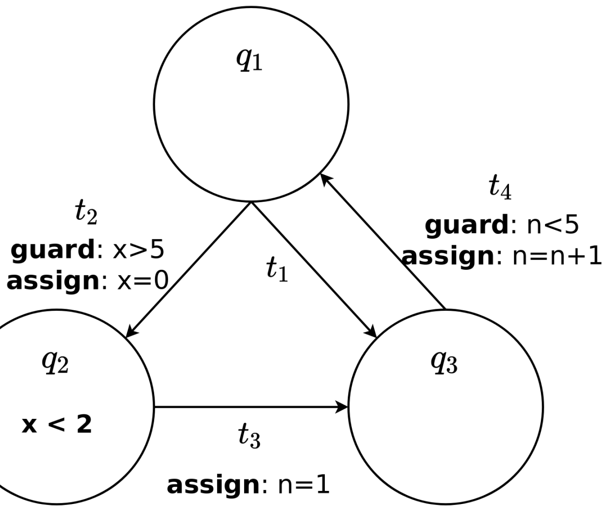

Figure 2: A Timed Automaton with clock x and variable n.

We can now show how these values are used and manipulated by a timed automaton. Figure 2 shows the same example TA modified with transition guards, transition assignments, and state invariants. Transition guards are conditions over either clocks or variables that prevent the associated transition from being taken when they do not evaluate to true. 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 perfectly valid for transition t_2 to be taken when $x = 6$, even though the assignment $x = 0$ resets the value of x to 0,

Type	**Synchronization Semantics**
One-to-one	Whenever a TA $\mathcal{A}_i$ takes a transition with a syncronization event $\alpha!$, there exists exacty one $\mathcal{A}_j$, $i \neq j$, such that in the same instant $\mathcal{A}_j$ has an active transition t' with the synchronization event $\alpha?$, and vice versa.
Broadcast	Whenever a TA $\mathcal{A}_i$ takes a transition with a synchronization event $\alpha\#$, every other TA $\mathcal{A}_j$ must not simultaneously take a transition with the same event, and must either simultaneously take a transition labelled with $\alpha@$, or there must not exist a transition $t' \in \mathcal{A}_j$ such that t_- is the currently active state, $\alpha@$ is the synchronization event, and the clock and variable guards of t' are satisfied.

Table 1: Supported Synchronization Events

which is smaller than the value accepted by the transition guard $x > 5$. Variables can be assigned to any value, while clocks can only be reset to 0. The third feature to mention are the state invariants. In our example there is only one, $x < 2$ which is associated with state q_2. When a state is active, its invariant (if any) is required to be true. The invariant attached to q_2 requires the TA to 'leave' state q_2 before clock x reaches a value of 2.

In addition to clocks and variables, TA also offer a powerful synchronization mechanism for different automata in the same network to coordinate. Any transition may contain a synchronization event of the form $\{channel \times sync\}$, where *channel* can be any symbol and $sync \in \{!, ?, \#, @\}$. Two different timed automata can synchronize their transitions by labelling them with the same synchronization channel, and using the actions to describe the type of synchronization desired.

As shown in table 1, two types of synchronization are defined. The first type is one-to-one synchronization, which has two associated operators, one signifying one-to-one 'send' (!), and the second for one-to-one receiving (?). A transition labelled with $\alpha!$, for some channel α, can only be fired if at the same moment in time, another TA takes a transition labelled with the $\alpha?$ event. The second type of synchronization available is termed 'broadcast' synchronization, and again we have two operators, broadcast-send (#) and broadcast-receive (@). Like one-to-one synchronization, for a given channel α there can only be one active transition with the event $\alpha\#$, however the difference is that there can be 0,1, or multiple automata that sync using broadcast-receive at once. In addition, each automaton is *required* to perform a broadcast-sync 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.

With the basic concepts introduced, we will formally define the Timed Automata discussed in this paper. 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 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 $x \sim c | \neg\gamma | \gamma \wedge \gamma$, where $x \in X$, $\sim \in \{<, =\}$, 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 Timed Automaton with variables is defined as the tuple $\mathcal{A} = \langle AP, X, Act_\tau, Int, Q, q^0, v^0_{var}, Inv, L, T \rangle$. In this tuple Q is the finite set of states of the timed automaton, $q^0 \in Q$ is the initial state of the TA, $v^0_{var} : Int \to \mathbb{Z}$ is a function providing initial values for each of the variables, and $Inv : Q \to \Gamma(X)$ is a function assigning each state to a (possibly empty) set of clock constraints. The labeling function $L : Q \to \mathcal{P}(AP)$ assigns each state to a subset of the atomic propositions. Each transition $t \in T$ has the form $t = \langle Q \times Q \times Act_\tau \times \Gamma(X) \times \Gamma(Int) \times \mathcal{P}(Assign(X)) \times \mathcal{P}(Assign(Int)) \rangle$, consisting of a source and destination state, 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 states 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 network of Timed Automata is a finite list of timed automata $\mathcal{A} = [\mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_N]$. Timed Automata in the same network can refer to 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 timed automaton 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 $\mathcal{A}_i \in \mathcal{A}$.

2.2 Bounded Model Checking

Bounded Model Checking refers to the problem of evaluating if a given network of timed automata satisfies a given model, or property. To perform this evaluation, the TA network along with the property to be evaluated are transformed into a form acceptable by a Satisfiability Modulo Theories, or SMT solver. The solver then searches all possible executions of the system to determine if the property is satisfied. Before we can discuss this process, we must describe what we mean by an execution of a network of timed automata.

At a given position in time, the TA network can be described by the currently active states, as well as the values of the clocks and integer variables. To describe the values of the clocks and variables at different positions in time, we introdue 'valuations', which accept a clock or variable argument and return a value.

$$v : \mathcal{X} \to \mathbb{R}_{\geq 0}$$
$$v_{var} : Int \to \mathbb{Z}$$

Each time position has an associated clock and variable valuation. Because the execution of a TA is a series of instantaneous state transitions, interspersed throughout time, we can represent a TA execution as a series of 'snapshots' showing these moments of transitions. In order to achieve this representation, we use the concept of a trace. For the time being let us consider a trace η to be an infinite sequence

$$\eta = (l_0, v_0, v_{var,0}), \delta_0, t_0, (l_1, v_1, v_{var,1}), \delta_1, t_1, \ldots$$

Where $l_l[i]$ returns the active state $q \in Q_i$ for $i \in [1, N]$, and $t_l[i]$ returns a transition $t \in T_i \cup \sharp$, where $\sharp$ is the *null transition*, which signifies that the TA does not perform a discrete transition. We can see that the trace is made up of snapshots of the TA in a given moment of time (l, v, v_{var}), which are connected with a combined temporal δ and discrete t transition step. We can safely require that all transitions follow this pattern because two consecutive temporal transitions δ can simply be combined into one whose length is the sum of the two original transitions, and two consecutive discrete transitions can be combined **iff** no TA in the network performs a non-null transition in both transitions, in which case the trace would be illegal, for the TA would be in multiple states in the same time position.

l[1]	$=$	q_1	q_2	q_2	q_3
n	$=$	1	1	1	1
x	$=$	0	0	1	0
δ	$=$	6	1	1	3
t[1]	$=$		t_2	-	t_3
time		0	1	2	3

Figure 3: An example trace through 4 positions.

In figure 3 we can see a trace of the Timed Automaton defined in figure 2, which has been given an index of 1. To prevent the value of $t[i]$ being undefined at position 0, the value of $t[i]$ corresponds to the transition taken in the following time position. However for clarity in the traces shown here, the values of $t[1]$ have been shifted to the right by one position, so that the active transition is placed over the position in time where the transition actually occurs. At the top of the trace $l[1]$ tracks the currently active state of the timed automaton. In this simple trace the automaton begins in state q_1, transitions to q_2, remains in q_2 for another position, and then transitions to q_3. Below the active state we show the active values of both the variable n and the clock x at the given positions. The value δ shows the amount of time that passes between the current state, and the immediately following state.

One surprising observation is that the value of the clock x does not seem to change between positions 0 and 1, despite δ indicating that 6 units of time has passed. Recall that transition t_2 resets the value of the clock x to zero, and that the trace capture the values of clocks and variables *after* any assignments have been applied.

Figure 4 shows how we can compute the value of clock x at the moment of the transition, before the reset is applied. By combining the values of x and δ, we obtain the value of x that is used to determine if the clock guard of transition t_2 is satisfied. We see that x has a value of 6, which satisfies the guard $x > 5$.

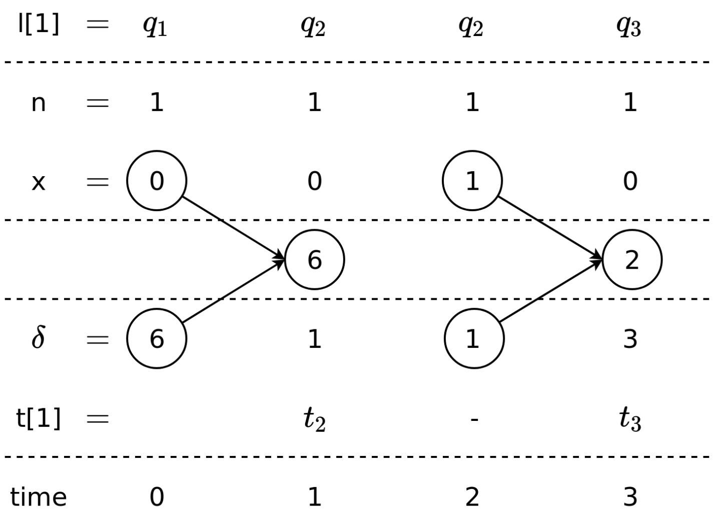

Figure 4: A trace highlighting the evaluation of a clock guard.

Notice in the above trace the value of clock x during the transition from state q_2 to state q_3. State q_2 has an invariant requiring that the value of clock x be strictly less than 2, however at the moment of transition $x(2) + \delta(2) = 2$. Is this trace therefore illegal? This raises an interesting question: at the moment of transition, what is the active state? Possible answers could include the source state, the destination, both, or neither. Different models of Timed Automata implement this differently[?]. In our model, the TA is always in exactly one state at every instant in time. If at the moment of transition the TA remains in the source state, the transition is said to be "right-closed" or equivalently "left-open", because the interval of time that the TA spends in t_- ends in a closed interval, while the inteval of time that the TA spends in t_+ begins in an open interval. Conversely if the TA is located in the destination state at the moment of transition, we say that it is "left-closed", which also implies that it is right-open. Returning to our example trace, if the timed automaton is not in state q_2 in the instance of transition, then the strict inequality in the state invariant can be satisfied with equality at the instance of transition, since the automaton is not actually in that state at that final moment. To formalize this notion we introduce the weak clock relation $\sim_w$, which is defined as follows:

$$x \sim_w c \iff (x \sim c \ \vee \ x = c) \quad \sim \in \{<, >, \leq, \geq\}$$
$$x =_w c \iff false$$

To make our trace more precise, we add for each TA $\mathcal{A}_i$ the term $edge^{RC}[i]$, which is true at a given time position iff the currently active edge is right-closed.

l[1]	$=$	q_1	q_2	q_2	q_3
n	$=$	1	1	1	1
x	$=$	0	0	1	0
δ	$=$	6	1	1	3
t[1]	$=$		t_2	-	t_3
edge	$=$		](		)[
time		0	1	2	3

Figure 5: An example trace with edge variable.

Figure 5 shows the same example trace as before, but with the addition of the edge variable. Like the transitions, it is shifted to the right by one position for ease of viewing. Notice that when transition t_3 is taken, the edge is left-closed, as is required by the invariant. The value of the edge variable is not shown during the null transition, however during a null transition either value is equivalent. We now revise the notion of a trace to include this term.

Definition 2.1. A trace η is an infinite sequence

$$\eta = (l_0, v_0, v_{var,0}), \delta_0, t_0, edge_0(l_1, v_1, v_{var,1}), \delta_1, t_1, edge_1 \dots$$

When using Timed Automata to model real-time systems, a common desire is to verify that every valid trace of the system obeys a given constraint, or property. Problems of feasibility quickly arise however, due to the infinite length of these traces. Bounded Model Checking is a process in which timed automata traces of infinite length can be efficiently verified against a property. Since TA traces are infinite in length, we restrict ourselves to traces of the form $s_0 s_1 \dots s_{l-1}(s_l s_{l+1} \dots s_{k-1} s_k)^{\omega}$, where every $s = (l, v, v_{var}, \delta, t)$. These "lasso-shaped" traces consist of an initial sequence of states up until s_{l-1}, followed by a loop that can be repeated an infinite amount of times to form the full trace. Since the beginning of the loop is allowed to occur anywhere within the sequence, the only variable is the number of distinct states k. Bounded Model Checking refers to checking if a given property is satisfied over lasso-shaped traces of up to length k. The TA system along with the desired property are converted into a format suitable for parsing by a SAT or

SMT solver, which is then tasked with finding a counterexample to the property. If a counterexample is found, then there exists at least one trace that does not satisfy the provided property. Otherwise, the property is said to have been verified over the TA network up to the bound k, as the solver has shown that no lasso-shaped traces of length k exist that contradict the property.

2.3 Constraint LTL Over Clocks

CLTLoc is a temporal logic that allows for the construction of formulas defined over atomic propositions, clocks, and arithmetic variables. A clock is a variable over $\mathbb{R}_{\geq 0}$ whose value changes between LTL positions to model the passage of time. Like clocks in timed automata, clocks can be reset back to zero.

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 $b + c = 7$; $b, c \in Int$.

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

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

Where $a \in AP$, $x \in X$, $c \in \mathbb{N}$, $n \in Int$, and exp are arithmetic formulas over integer variables. As mentioned before, clocks are special dense variables over $\mathbb{R}_{\geq 0}$ that 'progress' between different LTL positions. To be more specific, each clock must either increment between two adjacent time positions, or it must be reset. To maintain a consistent view of time, we introduce $\delta : \mathbb{N} \rightarrow \mathbb{R}_{>0}$, which measures the amount of time that elapses between two adjacent time positions. For a given clock 'valuation' $\sigma : \mathbb{N} \times X \rightarrow \mathbb{R}_{\geq 0}$, each clock $x \in X$ must either obey the equivalence $\sigma(i, x) + \delta(i) = \sigma(i + 1, x)$, or is reset, i.e. $\sigma(i + 1, x) = 0$. This ensures that all clocks progress at the same rate, and we can use $\delta(t)$ to calculate the amount of time elapsed between any two positions.

2.4 TACK CLTLoc Translation

The TACK[1] tool developed by Menghi et al. converts Bounded Satisfiability Checking problems into the CLTLoc language. TACK uses Metric Interval Temporal Logic to specify the property to be checked for satisfiability, which allows for more compact and powerful specifications of the desired properties to be checked. Once the Timed Automata network and the MITL property have been converted into CLTLoc, TACK then uses the tool Zot to convert this intermediate representation of the problem into the SMT-LIB2 language, which is supported by many modern SMT solvers. Zot was designed with a modular architecture to allow for several different strategies and algorithms that can be used to convert its input. Currently the most successful Zot plugin for CLTLoc Bounded Model Checking is ae2sbvzot. We will provide an overview of both the TACK encoding of Timed Automata in CLTLoc and the ae2sbvzot translation of CLTLoc into BitVector form.

Each time position in an infinite trace of a network of timed automata is represented as a position in the mono-infinite temporal space of CLTLoc. At every time position $l[i]$, $t[i]$ and $v_{var}[n]$ return the active state, active transition, and value of the variable at the current time position. Each function is syntactic sugar for a set of atomic propositions, one for each possible value of the function, that are constrained so that only one may evaluate to true in each time position. Each $edge_i^{RC}$, $i \in [1, N]$ is encoded as an atomic proposition. TA clocks can be represented directly as CLTLoc clocks.

Table 2 contains the formulas used to encode the Timed Automata into CLTLoc. To accomplish this encoding, several auxillary formulas are used. $l[i], i \in [1, N]$ represents the *location* of the TA i at the current time position. Likewise, $t[i], i \in [1, N]$ represents the currently active transition for TA i at the current time position. Because not every TA will transition at each time position, we introduce a *null transition* symbol $\sharp$. Therefore the function $t[i]$ may return either a transition or the symbol $\sharp$. If transition t is active in a given position i, then the TA is in state t_- at position i and state t_+ at position $i + 1$.

The first formula constrains each TA to be in the initial state at time 0. For each Timed Automaton, the states are represented as natural numbers, with the initial state as 0. The second formula initializes each variable $n \in Int$ to its initial value, and the third ensures that the invariants of each initial state hold in the initial position.

Table 2: TACK Encoding of an Automaton in CLTLoc

$\varphi_1 := \bigwedge_{i\in[1,N]} (l[i] = 0)$	$\varphi_2 := \bigwedge_{n\in Int} n = v^0_{var}(n)$	$\varphi_3 := \bigwedge_{i\in[1,N]} Inv(l[i])$
$\varphi_4 := \bigwedge_{x\in X} (x_0{=}0 \wedge x_1{>}0 \wedge x_v{=}0)$	$\varphi_5(j) := \bigwedge_{x\in X} (x_j{=}0) \rightarrow \mathcal{X}\left((x_{(j+1) \mod 2} = 0)\mathcal{R}\big((x_v{=}j)\wedge(x_j{>}0)\big)\right)$	
$\varphi_6 := \bigwedge_{\substack{i\in[1,N]\\ q\in\mathcal{Q}_i}} \Big(\big(l[i] = q \wedge t[i] = \sharp\big) \rightarrow \mathcal{X}\big(Inv(q) \wedge r_1(Inv(q))\big)\Big)$		
$\varphi_7 := \bigwedge_{\substack{i\in[1,N]\\ t\in T_i}} t[i] = t \rightarrow \big(l[i] = t_- \wedge \mathcal{X}(l[i] = t_+) \wedge \varphi_{\gamma_c} \wedge \varphi_{\gamma_v} \wedge \varphi_{\alpha_c} \wedge \varphi_{\alpha_v} \wedge \varphi_{edge}(t_-, t_+, i)\big)$ $\varphi_{edge}(a, b, i) := \varphi_{\alpha](}(a, b, i) \vee \varphi_{\alpha)[}(a, b, i)$ $\varphi_{\alpha)[}(a, b, i) := Inv_w(a) \wedge r_2(Inv(b)) \wedge \neg edge^{](}[i]$		
$\varphi_8 := \bigwedge_{i\in[1,N];q,q'\in\mathcal{Q}_i\mid q\neq q'} \Big(\big((l[i] = q) \wedge \mathcal{X}(l[i] = q')\big) \rightarrow \bigvee_{t\in T_i\mid t_-=q,t_+=q'} (t[i] = t)\Big)$		
$\varphi_9 := \bigwedge_{x\in X} \Big(\mathcal{X}(x_0 = 0 \vee x_1 = 0) \rightarrow \bigvee_{\substack{i\in[1,N]\\ t\in T_i\mid x\in t_{a_c}}} t[i] = t\Big)$		
$\varphi_{10} := \bigwedge_{n\in Int} \Big((\neg(n = \mathcal{X}(n))) \rightarrow \bigvee_{\substack{i\in[1,N]\\ t\in T_i\mid n\in t_{a_v}}} t[i] = t\Big)$		

Formula φ defines the semantics for the null transition. If a Timed Automaton performs a null transition then at the moment of transition, the state invariant must hold both before and after clock resets are applied.

Formula φ encodes the discrete transitions. Each must respect the guards and assignments of the transitions, the TA must currently be in the source state of the transition, and must be in the destination state in the following position. φ_{edge} encodes the two possible edge states, right and left-closed, and ensures that the invariants are satisfied, either weakly or strongly depending on the edge type.

The final three formulas capture the sufficient conditions for a discrete transition. The active state of a TA my not change, nor may a clock be reset, nor may a variable value change without a transition explicitly causing the change.

Table 3 contains the synchronization semantics for the two supported synchronization types. These synchronization semantics are defined in table 1. The formula *sync-on* is true if the automaton i performs a transition with the event α. *Sync-on-but* is true if an automaton not included in the set S performs a sync on α. *Same edge* is true if the to automatons have the same edge type.

Type	**Synchronization Semantics**
One-to-one	todo todo2
Broadcast	todo todo2
	$\varphi_{sync-on}$ $\varphi_{sync-on-but}$ $\varphi_{same-edge}$

Table 3: Encoding of Different Synchronization Types

Type	**Liveness Semantics**
Strong transition liveness	$\bigwedge_{i\in[1,N]} \mathcal{G}(\mathcal{F}(t[i] \neq \sharp))$
Weak transition liveness	$\mathcal{G}(\mathcal{F}(\bigvee_{i\in[1,N]} t[i] \neq \sharp))$
Unrestricted	$\top$

Table 4: Liveness Semantics

In addition to the TA and synchronization constraints, TACK includes support for an optional *liveness constraint*, termed Strong Transition Liveness, which is shown in table 4. This guarantees that every time position, eventually in the future all timed automaton in the network will take a non-null transition. Although a weaker version termed Weak Transition Liveness is defined in the TACK paper, it is not implemented in the TACK tool. It requres that at every position *at least one* timed automaton will eventually perform a non-null transition.

In addition to liveness, TACK also allows for customization of its edge constraints. As we have seen in section 2.2, the edge variables are necessary for defining which state an automaton is located in during the instant of transition. However the freedom to choose between two possible edge types is often not needed, and adds additional overhead to the solver. To speed up problems that do not have invariant edge cases, TACK provides

Type	**Edge Semantics**
Right-closed	todo
Open	todo

Table 5: Possible Edge Constraints

the open to set all edge variables to right-closed in every time position. As seen in table 5, this is contrasted with the 'open' edge semantics, which do not place any additional restrictions on the edge variables.

2.5 Bit-Vector Logic

A BitVector is an array of binary values, or bits. BitVectors are interpreted using two's complement arithmetic to produce integer values, and their length can be any positive integer ($\mathbb{Z}^+$). We use the notation $\overleftarrow{x}_{[n]}$ to represent a BitVector x of length n, but this can be simplified to $\overleftarrow{x}$ if the length is clear. Bits are numbered from right to left, with the rightmost, least significant bit labeled as 0, and the leftmost, most significant bit labeled as $n-1$. As an example, the constant vector -4 of length 5 would be written as $\overleftarrow{-4}_{[5]}$, which would expand to 11100. We can also reference individual bits in the vector using the notation $\overleftarrow{x}^{[i]}_{[n]}$ to *extract* the ith bit from the BitVector x. It is also possible to extract a sub-vector with the notation $\overleftarrow{x}^{[j:i]}_{[n]}$, where $n > j \geq i \geq 0$. This extracts a vector of length $j - i + 1$ whose rightmost bit corresponds to the ith bit of x and whose leftmost bit corresponds to the jth. Similarly, *concatenation* operates on two BitVectors by combining their bit arrays. $\overleftarrow{x}_{[n]} :: \overleftarrow{y}_{[m]}$ returns a new BitVector $\overleftarrow{z}_{[n+m]}$ where $\overleftarrow{z}^{[m-1:0]} = \overleftarrow{y}$, and $\overleftarrow{z}^{[m+n-1:m]} = \overleftarrow{x}$.

The usual arithmetic operations of addition (+) and subtraction (−) are defined over two BitVectors of the same length. BitVectors also support the bit-wise operators not (!), disjuction (|), conjunction (&), equivalence ($\iff$), and implication ($\Rightarrow$). These binary operators return a new BitVector where each bit i is the result of applying the logical operator to the ith bit of each of the input vectors, following the usual convention where 1 is true and 0 is false. For example, the expression $\left(\overleftarrow{1100} \Rightarrow \overleftarrow{1010}\right)$ would evaluate to $\overleftarrow{1101}$, since $a \rightarrow b$ is equivalent to $a \vee \neg b$.

2.6 AE2SBVZOT

The final program to mention is ae2sbvzot, which is a BitVector-based plugin for Zot. It accepts CLTL formulas and converts them to BitVector logic, which it then sends to Microsoft's Z3 to solve.

Table 6: An example ae2sbvzot trace showing loop variables.

BitVector	**4**	3	**2**	1	0
$\overleftarrow{foo}$	**1**	0	**1**	1	0
$\overleftarrow{\neg foo}$	**0**	1	**0**	0	1
$\overleftarrow{\mathcal{X} foo}$	**0**	1	**0**	1	1
$\overleftarrow{lpos}$	0	0	0	1	0
$\overleftarrow{inloop}$	1	1	1	0	0

To model the lasso shape of runs, ae2sbvzot adds an additional position to the BitVector that represents the 'loopback' position, or the first position of the next iteration of the loop. This position becomes the left-most, most significant bit of the vector. To separate the lasso from the initial portion of the trace, ae2sbvzot defines two special BitVectors, $\overleftarrow{lpos}$ and $\overleftarrow{inloop}$. In table 6 we can see an example formula, foo, along with the corresponding vectors $\overleftarrow{lpos}$ and $\overleftarrow{inloop}$. We can see that $\overleftarrow{lpos}$ has a value of 2, meaning that bit 2 is the first position in the loop. Looking at the table we can see that the columns for bits 2 and 4 are in bold, to represent that 4, being the 'loopback' position, is a copy of position 2, and therefore all formulas have identical values in these positions. Meanwhile $\overleftarrow{inloop}$ highlights that bits 0 and 1 are **not** in the loop portion of the trace, while the rest of the positions are. The infinite trace therefore would begin in position 0, move to position 1, and then repeat the infinite sequence of positions [232323...].

Table 7: ae2sbvzot definition of a proposition φ

ϕ	Encoding
$\neg\varphi$	$\overleftarrow{(\neg\varphi)} = !\overleftarrow{(\varphi)}$
$\varphi_1 \wedge \varphi_2$	$\overleftarrow{(\varphi_1 \wedge \varphi_2)} = \overleftarrow{(\varphi_1)} \,\&\, \overleftarrow{(\varphi_2)}$
$\varphi_1 \vee \varphi_2$	$\overleftarrow{(\varphi_1 \vee \varphi_2)} = \overleftarrow{(\varphi_1)} \mid \overleftarrow{(\varphi_2)}$
$\mathcal{X}\varphi$	$\overleftarrow{(\mathcal{X}\varphi)}^{[k:0]} = \overleftarrow{(\varphi)}^{[k+1:1]}$
$\varphi_1\mathcal{U}\varphi_2$	$\overleftarrow{(\varphi_1\mathcal{U}\varphi_2)}^{[k:0]} = \overleftarrow{(\varphi_2)}^{[k:0]} \mid \Big(\overleftarrow{(\varphi_1)}^{[k:0]} \,\&\, \overleftarrow{(\varphi_1\mathcal{U}\varphi_2)}^{[k+1:1]}\Big)$ $\wedge\Bigg(\Big(\big(\overleftarrow{(\varphi_1)}^{[k+1]} \mid \overleftarrow{(\varphi_2)}^{[k+1]} \mid !\overleftarrow{(\varphi_1\mathcal{U}\varphi_2)}^{[k+1]}\big)\,\&$ $(!\overleftarrow{(\varphi_2)}^{[k+1]} \mid \overleftarrow{(\varphi_1\mathcal{U}\varphi_2)}^{[k+1]})\Big) = 1\Bigg)\wedge$ $\Big(\overleftarrow{(\varphi_1\mathcal{U}\varphi_2)}^{[k+1]} \Rightarrow\Uparrow \big(\overleftarrow{(\varphi_2)} \,\&\, \overleftarrow{(inloop)}\big) = 1\Big)$

Given a formula with a bound of k, ae2sbvzot constructs a BitVector for the formula and for each subformula with a length of k+2. It adds an extra position both at the beginning (bit 0) to represent the initial configuration of the system, as well as at the end (bit k+1) to represent the loopback position as discussed. Each formula is then represented as a formula over its component subformulas based on the rules in table 7. The first three formulas show the encoding of logical operators $\neg, \wedge$, and $\vee$. The rest of the table shows the encoding of the temporal operators. Extra care is taken in the definition of $\mathcal{U}$ to ensure that the properties of the lasso are taken into account.

Atomic propositions (including the active states, transitions, and variable valuation) are encoded as BitVectors, and form the basic atoms from which more complex expressions are built upon. For each state, transition, and possible variable assignment (i.e. $n = 4$), there is a BitVector defined whose value is 1 if the corresponding state or transition is active, or if the variable assignment is true. One limitation of this approach is that arithmetic operations over variables cannot be represented. As a result, in this implementation, variable guards may only test for equality, and different variables cannot be added or subtracted in a variable assignment statement.

Clocks, being real-valued functions, cannot be easily represented in BitVector logic. Instead ae2sbvzot takes advantage of SMT-LIB2's support for multiple logics, and encodes clocks directly as functions that accept an integer argument, representing the time position, and return a real number. For a given clock guard a BitVector is constructed to represent the truth value of the clock expression. These can then be used to contruct larger expressions using the rules shown in table 7.

3 TA Encoding

The previous work in this field has relied on first translating the TA network and property to be verified into the intermediate CLTLoc representation, to be later translated into BitVector logic and solved. Presented here is a novel method in which the Timed Automata network is directly encoded into BitVector logic. 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 ae2sbvzot. We use a consistent naming convention for the atomic propositions to ensure that the two BitVector encodings can be safely combined to produce the final SMT output.

Before discussing the encoding, a few differences between the representations must be addressed. First and foremost, 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 transitions that obey the same rules as the discrete transitions. We explicitly add a null transition for each state $q \in Q$ to the set of transitions.

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

These null transitions have the same source and destination state, and no constraints or assignments. We can now refer to the set of all transitions as $\mathcal{T}$, defined as $\mathcal{T}_\rangle = T_i \cup \{ \underset{q\in Q_i}{\cup} t_{null_q} \}$ for each TA $\mathcal{A}_i$. As before $\mathcal{T}$ is the union of the $\mathcal{T}_i$ sets.

Using BitVector logic, we have the ability to group logically connected propositions into a Vector, granting significant speedups on operations performed over every element of the vector. We wish to use this property to group together logically connected constraints of the encoding. One common source of constraint duplication is the transition constraints. These constraints enforce the semantics of Timed Automaton transitions, requiring that, for example all guards and assignment statements have been fulfilled. These constraints must be upheld at every transition in the trace, which can be dozens of discrete transitions long. This motivates us to use the BitVectors to represent a piece of information changing over time, i.e. representing its value at different time positions in the trace.

3.1 Transitions

When encoding the constraints of the system, it is convenient to write that a constraint will hold over every discrete time position in the trace. As an example, consider a transition with a variable assignment $n = 5, n \in Int$. When formalizing the constraints, it would be simpler to have a formula of the type $\overleftarrow{transition}_{[k+2]} \Rightarrow \overleftarrow{assignment}_{[k+2]}$ that we can assert over every time position at once. In this model we are using BitVectors of length $k+2$ where each position in the BitVector represents the formula at a different moment in time. This allows us to use the BitVector implication operator to assert that the transition BitVector implies a given constraint at every time position. At a given time position i, if the corresponding bit in the transition BitVector is set to 1, then the same bit in the variable BitVector would be required to be set to 1 as well. Otherwise the SMT solver would dismiss the trace as invalid.

Transition Bit	BitVector
0	$\overleftarrow{tb_{i,0}}_{[k+2]}$
1	$\overleftarrow{tb_{i,1}}_{[k+2]}$
$\vdots$	$\vdots$
$(\lceil\log_2 \mathcal{T}_i\rceil - 1)$	$\overleftarrow{tb_{i,(\lceil\log_2 \mathcal{T}_i\rceil-1)}}_{[k+2]}$

Table 8: Construction of the Transition BitVectors for $\mathcal{A}_i \in \mathcal{A}$

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

Table 9: Construction of the Transition Aliases

This encoding, while convenient, is not very efficient. Using one BitVector per each transition yields a space complexity of $O(|\mathcal{T}|k)$. Since only one transition is active at a time, it is more compact to store the currently active transition as a binary number over $\lceil\log_2|\mathcal{T}|\rceil$ bits. Therefore we will create $\lceil\log_2|\mathcal{T}|\rceil$ BitVectors of length $k+2$ to represent the active transition of the TA over time. In order to be able to conveniently refer to individual elements of the set, we will define aliases which refer to unique combinations of the BitVectors. This will give us the convenience of the individually-named BitVectors while retaining the efficiency of the compact approach. This method will be formalized below for the encoding of the states, transitions, and variables of the Timed Automata.

For a model with a time bound of k, and a timed automaton with n distinct transitions, we represent the active transition of the automaton at different time positions as follows:

After defining the BitVectors to store the bits of the active transition, we define aliases for the $|\mathcal{T}_i|$ transitions as shown in table 9. Transition 0 is simply defined as the bit-wise 'and' of the negations of each of the *tb* BitVectors. Transition 1 differs in that the last BitVector is not negated. This corresponds to the BitVector representation of the number 1, which is $\overleftarrow{00\ldots001}$. When viewed in the table, the pattern becomes more clear. Each transition is encoded as a unique combination of the *tb* vectors. 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 $|\mathcal{T}_i|$. The key point is that each alias $\overleftarrow{trans_t}$ represents when transition t is active - a bit at position l is set to 1 iff the transition occurs between positions l and l+1.

For clarity, let us consider an example TA with $\lceil \log_2 |\mathcal{T}| \rceil = 5$ and a transition $t \in \mathcal{T}$ with $\mathcal{T}_i[5] = t$. The base two representation of 5 is 00101, and therefore $\overleftarrow{trans}_{t[k+2]}$ is equivalent to $(\neg tb_5 \ \& \ \neg tb_4 \ \& \ tb_3 \ \& \ \neg tb_2 \ \& \ tb_1)$.

Another consideration we must make is for the transition edges. We add an additional term $\overleftarrow{edge_i^{RC}}_{[k+2]}$, $i \in [1, N]$ for each Timed Automaton in the network. When a bit is set to 1, it signifies that the active transition for the Timed Automaton at that time position is right-closed, and conversely a value of 0 means that it is left-closed. Although not every transition will be impacted by this term (for instance, the null transitions have no invariants that can be affected), it is a necessity for the correctness of our system.

3.2 States

For each TA $\mathcal{A}_i \in \mathcal{A}$, we need a way to represent the currently active state of the timed automaton. Like with the transition encoding, we wish to minimize the number of BitVectors that the SMT solver must compute. Since we have already encoded the active transition into BitVector form, we can exploit the fact that given the active transition t, the active state of the TA is simply the source state of the transition, or t_-. Therefore all that is needed is to define a set of aliases that exploit this equivalence. To this end we define each state as the bit-wise disjunction of all the transitions whose source is that state.

$$\underset{q \in Q}{\forall} \ state_q := \underset{t \in \mathcal{T} | t_- = q}{\Big|} \ trans_t$$

3.3 Variables

Bounded integer variables are treated slightly differently, because unlike states and transitions, the possible values of a bounded integer variable are not unrelated objects in a set, but integers that must respect the operations of addition and subtraction. For each variable $n \in Int$ we still construct a bit representation $\overleftarrow{vb}_{n,j[k+2]}$, where each BitVector has length $k + 2$. However the difference is that the values are encoded in twos complement notation, and the number of BitVectors is chosen so that the vectors are capable

Terms		
Transitions	$\underset{i\in[1,N]}{\forall}\ \underset{j\in[0,\lvert\mathcal{T}_i\rvert-1]}{\forall}$	$\overleftarrow{tb_{i,j}}_{[k+2]}$
Variables	$\underset{n\in Int}{\forall}\ \underset{j\in[0,\lambda(n)-1]}{\forall}$	$\overleftarrow{vb_{n,j}}_{[k+2]}$
Clocks	$\underset{x\in X}{\forall}$	$x:[0,k{+}1]\rightarrow\mathbb{R}_{\geq 0}$
δ		$\delta:[0,k{+}1]\rightarrow\mathbb{R}_{>0}$
Edges	$\underset{i\in[1,N]}{\forall}$	$\overleftarrow{edge_i^{RC}}_{[k+2]}$
Loop		$\overleftarrow{0}_{[k+2]} < \overleftarrow{loop}_{[k+2]} < \overleftarrow{k}_{[k+2]}$
Aliases		
Transitions	$\underset{i\in[1,N]}{\forall}\ \underset{t\in[1,\mathcal{T}_i]}{\forall}$	$\overleftarrow{trans_t} = \underset{j\in[0,\lceil\lvert\log_2\mathcal{T}_i\rvert\rceil-1]}{\&} !^{(t\backslash 2^j) mod\ 2}\ (!(\overleftarrow{tb_{i,j}}))$
States	$\underset{i\in[1,N]}{\forall}\ \underset{q\in Q_i}{\forall}$	$\overleftarrow{state_q} = \underset{t\in\mathcal{T}_i}{\vert}\ \overleftarrow{trans_t}$
Variables	$\underset{n\in Int}{\forall}\ \underset{l\in[0,k+1]}{\forall}$	$\overleftarrow{var_n(l)} = \underset{j\in[\lambda(n),0]}{:}\ \overleftarrow{vb_{n,j}}^{[l]}$
inloop	$\underset{l\in[0,k+1]}{\forall}$	$\left(\overleftarrow{inloop}^{[l]}_{[k+2]} = \overleftarrow{1}_{[1]}\right) \leftrightarrow (\overleftarrow{loop} \leq \overleftarrow{l}\ )$

Table 10: Terms and Aliases used in BV encoding of TA

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.

However sometimes it is more convenient to refer to the complete value of a variable at a particular time position, rather than a particular bit of the variable over every time position. We make use of the 'extract' and 'concat' operators to define a second set of BitVectors that are defined over the first set. $\overleftarrow{var_n(l)}_{[\lambda(n)]}$, $0 \leq j \leq k+1$ is a vector of $\lambda(n)$ bits that represents the value of variable n at time position j.

3.4 Clocks

Each clock $x \in \mathcal{X}$ is defined as a function x that takes an integer argument and returns a real number, where the argument represents the time position and the return value is the value of the clock at that position in time.

3.5 Complete Encoding

Now that each piece of the timed automaton has been discussed, we can present all of the terms used in the encoding of the TA network. A valid trace of the network consists of assigning values to the following terms in table 10. As we can see, the terms that our SMT solver will need to assign values to are the compacted transition and variable BitVectors, as well as the real-valued clock functions. In addition we define two special terms, which will be used to help define our constraints. The first, δ, represents the amount of time that passes between two adjacent time positions, and must be a real number greater than 0. 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. In addition, for ease of comprehension we have aliases to more easily refer to the transitions and states individually, and to refer to the value of a variable at a particular time position.

4 Constraints

Of course, in addition to being able to represent all valid lasso-shaped traces, the terms defined can represent many illegal traces as well. Timed Automata in our network cannot simply take any transition as they please, they must obey certain restrictions. These restrictions take the form of clock guards on a transition, a state invariant that prevents a TA from staying in a state indefinitely, clock progression constraints, as well as many others. To ensure that our encoding only allows for valid traces, we will formalize these constraints in BitVector logic for our SMT solver to use when performing the Bounded Model Checking.

4.1 Initialization & Progression

In the definition of a Timed Automaton, we included the initialization terms q^0 and var^0, and mentioned that all clocks are equal to 0 at the initial instance of the trace. Here we formalize those contraints over the BitVector terms that make up our encoding. Also included are contraints on how these three groups of values (transitions, variables, and clocks) evolve throughout the trace.

Initialization and Progression Constraints

$$\phi_1 := \mathop{\&}_{i\in[1,N]} \overleftarrow{1}_{[1]} = \overleftarrow{state_{q_i^0}}^{[0]} \qquad \phi_2 := \mathop{\&}_{n\in Int} \overleftarrow{v_{var}^0(n)} = \overleftarrow{var_n(0)} \qquad \phi_3 := \bigwedge_{x\in X} x(0) = 0$$

$$\phi_4 := \bigwedge_{i\in[0,k+1]} \delta(i) > 0 \qquad \phi_5 := \mathop{\&}_{t\in\mathcal{T}} (\overleftarrow{trans_t}^{[k:0]} \Rightarrow \overleftarrow{state_{t_+}}^{[k+1:1]}) \qquad \phi_6 := \{\phi_{wtl} \mid \phi_{stl} \mid \top\}$$

$$\phi_7 := \bigwedge_{x\in X} \bigwedge_{j\in[0,k]} \left(\mathop{\&}_{t\in\mathcal{R}(x)} (!\overleftarrow{trans_t})^{[j]} \right) \Rightarrow x(j+1) = x(j) + \delta(j)$$

$$\phi_8 := \mathop{\&}_{n\in Int} \mathop{\&}_{t\in assign(n)} (!\ \overleftarrow{trans_t}^{[k:0]}) \Rightarrow \mathop{\&}_{j\in[1,\lambda(n)]} (\overleftarrow{v_n b_j}^{[k:0]} = \overleftarrow{v_n b_j}^{[k+1:1]})$$

$$\phi_{wtl} := \overleftarrow{0}_{[k+2]} \neq \left(\overleftarrow{inloop}\ \&\ (\mathop{|}_{i\in[1,N]} !\ (\mathop{|}_{q\in Q_i} \overleftarrow{trans_{null_q}}))\right)$$

$$\phi_{stl} := \bigwedge_{i\in[1,N]} \overleftarrow{0}_{[k+2]} \neq \left(\overleftarrow{inloop}\ \&\ !\ (\mathop{|}_{q\in Q_i} \overleftarrow{trans_{null_q}})\right)$$

$$\phi_{Init} := (\phi_1\ \&\ \phi_2\ \&\ \phi_5\ \&\ \phi_8 =!\ \overleftarrow{0}_{[k+2]}) \wedge \phi_3 \wedge \phi_4 \wedge \phi_6 \wedge \phi_7$$

Properties $\phi_1 - \phi_3$ The initialization constraints are similar for states, clocks, and bounded variables. For states, we assert that the initial state holds in the first time position by comparing the vector for the initial state $state_{q_i^0}$ to the constant vector $\overleftarrow{1}_{[1]}$ in formula ϕ_1. This requires the first bit of the state vector to be set to 1, signifying that the state is active in time position 0. Because the state BitVector is an alias, what this constraint is requiring is that the active transition for each TA i in position 0 must have its source state be equal to the initial state for the TA, $t_- = q_i^0$. This transition can be either a discrete transition beginning in q_i^0 or the null transition for state q_i^0. For variables, we assert that the provided initial starting value, var_n^0 is equal to the value of the variable at time position 0. For clocks, we assert that the clock function at the initial time position is equal to 0 in formula ϕ_3.

Property ϕ_4 Each time position in the range $[0, k+1]$ represents an instant of time in which timed automaton may perform discrete (non-null) transitions. In between these positions, all timed automata remain stationary, and only the clocks progress. To capture this progression, we use the special clock δ. Formula ϕ_4 captures that δ is defined as a function over integers in the range $[0, k+1]$ that returns positive real numbers. The value of $\delta(i)$ at position i refers to the amount of time between position i and position $i+1$.

Property ϕ_5 Another aspect of progression is ensuring that the active state of a timed automaton correctly reflects the transitions being taken. To that effect, formula ϕ_5 asserts that when a transition is taken at time position i, the destination state is active at position i+1. Because the state BitVectors are just aliases defined over the transition BitVectors, we do not need to explicitly constrain the TA to be in state t_{-} at time position i, since this is true by definition.

Property ϕ_6 Unique among the progression contraints is ϕ_6, which encodes the desired *liveness property* of the network, discussed in **??**. This parameter can be chosen by the user, and has three possible values. ϕ_{stl} encodes *Strong Transition Liveness*, which states that at every moment in time, each TA will eventually in the future perform a discrete (non-null) transition. Since our traces are lasso-shaped, this is equivalent to requiring that each TA takes a discrete transition somewhere in the loop. Similarly, ϕ_{wtl} encodes *Weak Transition Liveness*, which states that at every instant of time *at least one* TA will eventually perform a discrete transition in the future. The key to these encodings is the expression $!(\underset{q \in Q_i}{|} \overleftarrow{trans_{null_q}})$, which for a given TA combines each of the null transitions with the bit-wise disjuction operator. This value is then negated, and consequently a bit of this expression is set to 1 iff the TA has a non-null transition active at that position. We can then compute the desired value by using $\overleftarrow{inloop}$. This BitVector has bits which are set to 1 iff the corresponding position is inside the loop portion of the trace. By requiring that a discrete transition occurs during the loop, we ensure that at every instant of the infinite trace, a TA will eventually take a discrete transition. The third liveness option is to have no liveness constraint at all, in which case ϕ_6 trivially evaluates to true ($\top$).

Property $\phi_7 - \phi_8$ Formula ϕ_7 connects δ to the other clocks. At each time position i, a clock is either reset by a transition, or its value increments by $\delta(i)$. To do this we define the set $\mathcal{R}_x$ for every clock x, which is defined as the set of all transitions t that reset the value of clock x. When no transition in $\mathcal{R}_x$ is active, the clock must progress according to the value of δ. Similarly for variables, we define the set $assign(n)$ for every variable n containing all transitions that assign a value to the variable. When none of these transitions are active, formula ϕ_8 ensures that the value of n remains unchanged.

4.2 Transitions

As a quick review, transitions consist of a source and destination state, a synchronization action, as well as (possibly empty) sets of clock constraints, variable constraints, clock assignments, and variable assignments. In section 4.1 on initialization and progression, ϕ_6 was defined to ensure that the source and destination states were implemented correctly - that the destination of one transition is the source of the next. We now encode the

Transition Constraints, Assignments, and Invariants

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

guard and assignment constraints of every transition in the TA network.

Property ϕ_9 We will first consider the transition guards. Each transition can have multiple guards, which consist of two types, clock guards and variable guards. Clock guards have the form $x \sim c$, where $x \in X$, $c \in \mathbb{Z}$ and $\sim \in \{<, >, \leq, \geq\}$. 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. Recall that if a transition is active at time position l, the transition occurs in the instant between time position l and time position $l+1$. Therefore, at the instance of the transition, the clock does not have the value $x(l)$, but rather $x(l) + \delta(l)$, delta being the special clock that defines the amount of time spent in each time position. Note that we cannot simply use $x(l+1)$ as the value of the clock, because it is possible that during the transition between time position l and $l+1$, the value of the clock may be reset, which would set $x(l+1) = 0$. Our guard only sees the pre-transition value of the clock, and thus we must manually add $\delta(l)$ to the value. We use the function σ to encode the grammar of the clock constraints. In this case we use σ_δ, which differs only in the fact that δ is added to the clock value.

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

Property $\phi_{11} - \phi_{12}$ Clock assignments are more straightforward then 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 to 0. Variable assignments however, are more complex. Unlike clock assignments, variable assignments can access 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 $\lambda(n') \leq \lambda(n)$, just as in the variable guards. We can then cast all constants and variables to BitVectors of length $\lambda(v)$, sign-extending shorter values 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

$l{+}1$.

Property ϕ_{13} The last component of the transition to discuss is the state invariant. Although invariants are state-specific, not transition-specific, since states 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 state and at the moment it leaves, it is satisfied at all positions in the interval between them. We can see that the transition implies one of two statements, one for each possible value of $\overleftarrow{edge_i^{RC}}$. In addition the invariants of the source state are evaluated with $\delta(l)$, while the transitions of the state with the open-ended transition are evaluated with the weak satisfaction relation.

4.3 Sync

Sync Constraints

$$\phi_{14} := \mathop{\&}_{t\in T:t_\epsilon=\alpha!} \overleftarrow{trans_t} \Rightarrow (\neg \mathop{|}_{\substack{t'\in\{T\backslash t\}:\\ t'_\epsilon\in\{\alpha!,\alpha\#\}}} \overleftarrow{trans_{t'}}) \wedge (\mathop{|}_{t'\in T:t_\epsilon=\alpha?} \overleftarrow{trans_{t'}})$$

$$\phi_{15} := \mathop{\&}_{\substack{i\in[1,N]\\ t\in T_i:t_\epsilon=\alpha?}} \overleftarrow{trans_t} \Rightarrow (\neg \mathop{|}_{\substack{t'\in\{T\backslash t\}:\\ t_\epsilon=\alpha?}} \overleftarrow{trans_{t'}}) \wedge (\mathop{|}_{\substack{j\in[1,N]\\ t'\in T_j:\\ t_\epsilon=a!}} \overleftarrow{trans_{t'}} \,\&\, (\overleftarrow{edge_i^{RC}} \Leftrightarrow \overleftarrow{edge_j^{RC}}))$$

$$\phi_{16} := \mathop{\&}_{t\in T:t_\epsilon=\alpha\#} \overleftarrow{trans_t} \Rightarrow \neg(\mathop{|}_{t'\in T:t_\epsilon=\alpha\#\wedge t'\neq t} \overleftarrow{trans_{t'}})$$

$$\phi_{17} := \mathop{\&}_{\substack{i\in[1,N]\\ t\in T_i:t_\epsilon=\alpha@}} \overleftarrow{trans_t} \Rightarrow (\mathop{|}_{\substack{j\in[1,N]\\ t'\in T_j:t_\epsilon=\alpha\#}} \overleftarrow{trans_{t'}} \,\&\, (\overleftarrow{edge_i^{RC}} \Leftrightarrow \overleftarrow{edge_j^{RC}}))$$

$$\phi_{18} := \bigwedge_{\alpha\in Act} \bigwedge_{\substack{i\in[1,|\mathcal{A}|]\\ l\in[0,k+2]}} (\bigvee_{\substack{t\in T_i:\\ t_\epsilon=\alpha\#}} \overleftarrow{trans_t}^{[l]}) \rightarrow$$
$$(\bigwedge_{\substack{j\in[1,k]:\\ j\neq i}} (\bigvee_{\substack{t'\in T_j:\\ t'_\epsilon=\alpha@}} \overleftarrow{state_{t'\_}}^{[l]} \wedge \sigma_\delta(l, t'_{\gamma_c}) \wedge \mu(l, t_{\gamma_v})) \rightarrow \bigvee_{\substack{t'\in T_j:\\ t'_\epsilon=\alpha@}} \overleftarrow{trans_{t'}}^{[l]})$$

$$\phi_{sync} := (\phi_{14} \,\&\, \phi_{15} \,\&\, \phi_{16} \,\&\, \phi_{17} = !\overleftarrow{0}) \wedge \phi_{18}$$

The synchronization constraints capture the semantics defined in table 1. Formula ϕ_{14} and ϕ_{15} capture the contraints for one-to-one synchronization, while the others cover broadcast synchronization.

Properties $\phi_{14} - \phi_{15}$ The first formula in this group requires firstly that when a transition marked with $\alpha!$ (one-to-one send) is taken, no other transition with the $\alpha!$ or $\alpha\#$ event may be active, and secondly that there exists at least one active transition with the event $\alpha?$ (one-to-one receive). The second formula is very similar, requiring that when a transition with the event $\alpha?$ is active, no other transition with the event $\alpha?$ may be active, and there must be at least one active transition with the $\alpha!$ event that has the same edge.

Properties $\phi_{16} - \phi_{18}$ Formulas ϕ_{16} and ϕ_{17} are similar to those for one-to-one communication, with modifications for the different semantics of broadcast synchronization. ϕ_{16} requires that when a transition with the event $\alpha\#$ (broadcast send) is active, no other transition with the same event may be active. The difference from ϕ_{14} is that there is no requirement for an active transition labelled with the $\alpha@$ (broadcast receive) event. ϕ_{17} requres that when a transition with the event $\alpha@$ is active, there exists at least one active transition with the event $\alpha\#$ and the same edge. Note that multiple transitions are allowed to receive the same broadcast signal. The final formula ϕ_{18} handles the "compulsive" nature of broadcast synchronization. When there exists an active transition with the broadcast send event on a channel α, for all other TA in the network we require that if there exists a transition

- that has the broadcast receive event on channel α
- whose source state equal to the current state of the TA
- whose guards are all satisfied

then the TA is required to take (have active) a transition with the event $\alpha@$.

4.4 Loop Constraints

Loop Constraints

$$\phi_{19} := \bigwedge_{i\in[1,N]} \; \bigwedge_{j\in[1,\lceil\log_2|\mathcal{T}_i|\rceil]} \overleftarrow{tb_{i,j}}^{[k+1]} = \overleftarrow{tb_{i,j}}^{[loop]}$$

$$\phi_{20} := \bigwedge_{n\in Int} \; \bigwedge_{j\in[1,\lambda(n)]} \overleftarrow{vb_{n,j}}^{[k+1]} = \overleftarrow{vb_{n,j}}^{[loop]}$$

$$\phi_{21} := \bigwedge_{x\in X} (\lfloor x(k{+}1)\rfloor \; = \lfloor x(loop)\rfloor) \vee (\lfloor x(k{+}1)\rfloor \; > \max(x) \wedge \lfloor x(loop)\rfloor > \max(x))$$

$$\phi_{22} := \bigwedge_{x\in X} \lfloor x(loop)\rfloor < \max(x) \Rightarrow (frac(x(k{+}1)) = 0) \Leftrightarrow (frac(x(loop)) = 0)$$

$$\phi_{23} := \bigwedge_{x,x'\in X} frac(x(k{+}1)) < frac(x'(k{+}1)) \Leftrightarrow frac(x(loop)) < frac(x'(loop))$$

$$\phi_{24} := \bigwedge_{x\in X} x(k) > \max(x) \vee ((\; \Big|_{t:x\in t_{ac}} \overleftarrow{trans_t}) \,\&\, \overleftarrow{inloop} \neq \overleftarrow{0}\,)$$

$$\phi_{loop} := \phi_{19} \wedge \phi_{20} \wedge \phi_{21} \wedge \phi_{22} \wedge \phi_{23} \wedge \phi_{24}$$

As mentioned previously, we are only interested in lasso-shaped runs that end in a loop. To keep track of the initial position of the loop, we have defined the BitVector $\overleftarrow{loop}$, and constrained it to have a value in the range $[1, k]$.

Intuitively, the time position $k{+}1$ represents the first time position in the next iteration of the loop. It is effectively a 'copy' of the position *loop*, however we add it as a distinct position so that we may capture the semantics of the transition between time position k and time position *loop*. We therefore must introduce constraints to ensure that these two positions are in fact equivalent. This requires that the active state and transition of each timed automata at position $k + 1$ be equal to that at position *loop*. Formula ϕ_{19} captures this by requiring that for each Timed Automaton, each transition bit $tb_{i,j}$ contains the same value at time positions $k + 1$ and *loop*. Similarly, formula ϕ_{20} enforces the same requirement for each bounded integer variable.

It is tempting to encode the clock constraints in a similar manner, requiring that $x(k + 1) = x(loop)$ for each clock. However prior work by Kindermann[ref] has shown that this constraint is not complete, as it excludes valid lasso-shaped runs. To remedy this problem we use the requirements suggested by Kindermann. To begin, for each clock x we define the non-negative integer $\max(x)$, which is equal to the maximum value either assigned to the clock in a clock assignment or compared against the clock in a clock guard. We also define $frac(x(l))$, which is equal to the fractional part of x at time position l, or

$frac(x(l)) = x(l) - \lfloor x(l) \rfloor$. Formulas ϕ_{21}, ϕ_{22}, and ϕ_{23} encode the desired requirements. ϕ_{21} encodes the first part of the relationship between $x(\mathit{loop})$ and $x(k+1)$. It states that either both values are greater than $\max(x)$, or both have the same floor. This is the first part of the region encoding. ϕ_{22} handles the special case where the fractional part of the value is equal to zero. Since clock guards can test for equality, if the clock value is less than $\max(x)$, either the clock value at both time positions has a fractional value of 0 or neither do. Finally, ϕ_{23} completes the region encoding by considering the relationship between values of different clocks, asserting that the relationship between two clock values $\{<, >, =\}$ is preserved.

Unfortunately, there is one more consideration we must make in this section. The culprit are so-called "Zeno traces", named because while they are lasso-shaped runs with an infinite number of transitions, their execution happens in finite time. Time in these traces is said to "slow down", because often each successive loop of the lasso executes in a smaller amount of time than the loop before. Because these represent unrealistic scenarios, they are often excluded from consideration in many TA models. It is sufficient to require that every clock is either reset within the loop, or has a value greater than $max(x)$ at position k, which is shown in ϕ_{24}. The vector $\overleftarrow{inloop}$ has length $k+2$, and each bit i is 1 iff $i \geq \mathit{loop}$. Using this vector, we can determine if a given clock is reset within the loop portion of the trace.

4.5 Equivalence

Using the definitions presented above, we are now ready to define our TA network in BitVector logic as follows:

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

We argue that this is a correct and complete representation of all lasso-shaped, non-Zeno runs given the bound of k.

Both encodings constrain the clocks, variables, and timed automata to their respective initial positions at time position 0. For variables and clocks these constraints are identical, as both assign the desired value at time position 0. For states our encoding uses the

$\overleftarrow{state_q}$ aliases to require that the TA begins in the initial state, despite not having state BitVectors. Because the state alias is only true when one of the transitions whose source is that state is true (including the state-specific null transitions), the constraint is valid. Our clock $\delta(l)$ 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 into several pieces, the functionality remains the same. We ensure that in order for a transition to be valid, its destination state 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 state must be true at the moment of transition. Like TACK, we allow 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.

In addition, our encoding contains improvements over the original TACK+ae2sbvzot encoding. As mentioned previously, ae2sbvzot contains a limitation regarding integer variables - because they are represented as elements of a set, ae2sbvzot can only test for equality. This means that contraints of the form $n \sim c$ or $n \sim n'$, where $\sim \neq =$ are not supported. Our encoding 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.

Although the TACK paper describes right-closed, left-closed, and unrestricted traces, the implementation currently only supports right-closed traces. We have implemented all three options in our encoding, allowing the user to choose which one they will use to evaluate their traces.

5 Evaluation

In this chapter we present the results of several experimental evaluations of the SMT conversion process. These tests cover several different benchmarks common for bounded model checking programs. For comparison, results are presented alongside those of the previous iteration of the TACK program, to better judge the improvements made using the new process.

In all of the following tests, the time provided measures the combined time taken by both the TACK program to parse the problem and convert it to SMT form and the z3 program to decide the satisfiability of the SMT problem. In practice, the time taken by z3 dwarfs the time used by the TACK translation, and for the problem sizes encountered below the time taken by the TACK translation was always negligible. For every test below, the evaluation proceeded in several rounds, each with a larger bound on the length of traces considered by TACK. The data obtained demonstrates how the running time of each program scales with the size of the search space.

All tests were performed on a server equipped with an Intel(R) Core(TM) i7-4770 CPU (3.40 GHz) with 8 cores, 16 GB of RAM and Debian Linux (version 4.19).

5.1 Fischer Mutual Exclusion Protocol

The Fischer benchmark models a protocol for ensuring exclusive access to a shared common resource that can be requested by multiple processes. The protocol uses global variables, integrated into the guards and assignment statements of the timed automata, to control access. Each timed automaton in the network has a 'critical state', and the protocol guarantees that only one timed automaton can be in its critical state at a time.

To be more specific, there is a shared variable id, which can take any integer value in the range $[0, n]$, where n is the number of processes in the protocol. Each process begins in an 'idle' state a, and in order to reach the critical section cs, a process must first check to see that the critical section is unoccupied ($id = 0$), at which point the process writes its own id to the shared variable (while entering state b) and then performs a second transition to state c within 2 seconds of entering state b. The process is then required to wait at least 2 seconds in state c. If after that interval the value of the shared variable is still equal to its id, the process may access the critical section, otherwise it must wait for the value of id to return to 0 before trying again (returning to b). Once access to the critical section is granted, the process may remain for an unlimited amount of time before returning to state a.

To measure the scalability of our program, in addition to modifying the bound k, we performed multiple test runs while modifying the number of timed automata in the network that are attempting to execute their critical region. Aside from a numerical id, these processes are identical in their behavior. In addition to testing the scalability of the program, we have also run the Fischer protocol through several different MITL properties for verification. These properties will be explained below.

$$
\begin{array}{lll}
Live1 & := & \mathcal{G}_{[0,\infty)}\ (p_1.b \Rightarrow \mathcal{F}_{[0,\infty)} p_1.c) \\
Live2 & := & \mathcal{G}_{[0,\infty)}\ (p_1.b \Rightarrow \mathcal{F}_{[0,3]} p_1.c) \\
Live3 & := & \mathcal{G}_{[0,\infty)}\ (p_1.b \Rightarrow \mathcal{F}_{(0,3)} p_1.cs) \\
Live4 & := & \mathcal{G}_{[0,\infty)}\ (p_1.b \Rightarrow \mathcal{F}_{(0,3)} p_1.c) \\
Live5 & := & \mathcal{G}_{[0,\infty)}\ (p_1.b \Rightarrow \mathcal{F}_{[0,3]} p_1.cs) \\
Live6 & := & \mathcal{G}_{[0,\infty)}\ \neg(\bigvee_{i=1:n-1} (p_i.cs \wedge \bigvee_{j=i+1:n} p_j.cs))
\end{array}
$$

Liveness property 1 requires that once process one enters state b, it always transitions to state c. Property 2 is similar, however it contains the additional constraint that process 1 must complete the transition to state c in at most 3 seconds. Property 3 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 4 and 5 are copies of properties 2 and 3 respectively with the sole difference of inclusion vs exclusion at the boundaries of the interval. Property 6 seeks to prove the “safety” of the protocol, namely that two distinct processes are never in the critical section at the same time.

5.2 CSMA/CD

The CSMA/CD (Carrier Sense Multiple Access/Collision Detection) protocol 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 1 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

Table 11: Time required to solve the Fischer Benchmark Properties

TACK ae2sbvzot

		n								
	k	2	3	4	5	6	7	8	9	10
live-one	10	0.9	0.9	1.0	1.0	1.2	1.2	1.5	6.0	1.7
	15	0.9	0.9	1.1	1.4	1.6	1.7	1.5	14.2	2.1
	20	1.0	1.1	1.3	2.2	1.9	2.6	2.4	3.6	3.9
	25	1.1	1.6	1.5	2.1	189.8	3.4	4.9	4.5	4.6
	30	1.1	1.5	2.0	3.1	6.4	4.5	4.9	4.7	4.8
live-two	10	1.4	1.6	1.7	2.0	2.0	2.4	2.5	2.7	3.1
	15	4.5	6.3	6.0	9.7	15.0	27.0	48.4	67.4	146.5
	20	10.8	13.0	37.5	22.1	56.6	36.8	56.8	567.9	1589.1
	25	32.6	39.3	57.3	131.8	102.4	2369.2	1614.3	1197.9	6311.7
	30	42.4	80.7	127.7	63.8	1149.1	125.9	–	–	–
live-three	10	1.0	1.1	1.6	3.6	8.0	13.2	19.5	17.6	26.7
	15	1.2	1.3	2.5	2.0	18.5	27.6	30.6	59.4	41.2
	20	1.2	1.8	2.1	4.2	8.0	10.4	23.9	33.1	64.7
	25	2.0	2.3	3.7	5.9	9.5	24.1	681.5	753.3	826.9
	30	1.5	4.6	4.6	8.8	32.7	24.4	30.7	102.8	150.1
live-four	10	1.2	1.3	1.5	1.7	1.9	2.1	2.1	2.4	2.3
	15	2.9	3.6	5.0	8.0	11.2	19.6	33.9	74.8	130.9
	20	6.6	15.5	14.4	17.7	77.7	86.2	183.5	504.3	1516.6
	25	10.3	17.8	22.9	45.5	151.4	2482.6	2406.0	6578.4	–
	30	27.5	26.2	33.9	56.0	67.0	217.1	1717.2	291.6	462.5
live-five	10	1.1	1.3	1.9	1.8	5.4	13.7	24.0	22.9	23.7
	15	1.3	1.7	2.5	2.3	5.3	5.7	20.3	–	36.8
	20	1.7	2.2	2.6	5.2	8.6	15.2	29.8	71.2	–
	25	2.1	3.6	4.4	13.5	10.6	17.2	781.3	816.1	569.4
	30	2.2	6.9	5.6	14.9	21.1	26.7	28.8	140.6	238.6
live-six	10	0.9	1.1	1.9	2.5	3.1	4.7	10.1	7.9	16.2
	15	1.0	1.8	4.4	10.0	33.9	51.2	84.0	110.9	191.5
	20	1.5	3.9	10.2	21.6	93.3	234.8	670.4	1000.4	–
	25	1.7	8.9	27.6	83.5	188.1	596.2	2469.2	5365.0	–
	30	2.5	16.3	54.4	263.4	993.9	3354.1	–	–	–

TACK ta2smt

		n								
	k	2	3	4	5	6	7	8	9	10
live-one	20	0.9	1.0	1.1	1.1	1.5	1.4	1.3	1.5	1.4
	25	1.0	1.1	1.2	1.2	1.4	1.5	1.6	1.5	1.6
	30	1.0	1.1	1.2	1.3	2.1	1.8	1.7	3.0	1.8
	35	1.1	1.1	1.3	1.6	1.7	1.8	1.8	1.8	2.9
	40	1.1	1.2	1.8	1.8	1.8	2.8	2.3	2.8	2.4
live-six	20	1.2	3.7	14.0	34.1	62.7	101.4	145.5	290.7	454.0
	25	1.4	6.7	21.1	44.8	83.9	144.3	511.3	673.0	847.3
	30	1.5	12.0	35.2	63.6	141.3	191.8	311.6	449.4	803.2
	35	2.0	19.0	40.2	117.3	173.4	281.0	450.3	599.4	1070.6
	40	2.6	24.2	58.4	119.5	246.3	410.3	539.8	1040.9	2924.0

of time before attempting to communicate again. While an agent is sending over the bus, the bus process can send a 'busy' signal to the other agents, in order to simulate the agents listening to the bus and detecting a communication in-progress.

$$\begin{aligned}
\textbf{live-csma} :=& \mathcal{G}_{(0,\infty)}(P1.begin\_send \rightarrow (\neg collision\_after\_deadline)) \\
P1.begin\_send :=& (\neg P1.send) \wedge (P1.send\, \mathcal{U}_{(0,inf)} \top) \\
P1.collision\_after\_deadline :=& \mathcal{G}_{(0,52]}(P1.send \wedge (P1.send\, \mathcal{U}_{[0,inf)}(P1.send \wedge P2.send)))
\end{aligned}$$

This protocol was tested to ensure that a collision is always detected before a certain amount of time has passed. The property **live-csma** asserts that after process 1 has been sending for 52 units of time, process 2 cannot begin sending until process 1 has finished.

5.3 Token Ring

The Token Ring protocol 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 is used to coordinate ownership of the token.

$$\begin{aligned}
\textbf{live-token} :=& \mathcal{G}_{0,\infty}(\neg((ST_1.zsync \vee ST_1.zasync \vee ST_1.ysync \vee ST_1.yasync) \wedge \\
& (ST_2.zsync \vee ST_2.zasync \vee ST_2.ysync \vee ST_2.yasync)))
\end{aligned}$$

This property asserts that agents 1 and 2 never simultaneously synchronize with the token.

6 Conclusion

We have presented a novel approach for encoding networks of Timed Automata directly into BitVector logic, suitable for solving by state of the art SMT solvers. Building off of the work begun with the development of the TACK solver, we retain TACK's approach of encoding MITL properties to be verified over the network, while creating a more efficient model for the encoding of the network itself. Rather than encode the TA network into CLTLoc, we directly translate the network terms and constraints into a combination of BitVector and real-valued logic, written in the standard SMT-LIB2 language.

In addition to re-implementing the translation of timed automata networks, we have corrected several difficiencies in the original TACK implementation, and the tool now supports the full variable constraint and variable assignment grammars. In addition our implementation allows users to choose the desired liveness and edge semantics at runtime. We are proud to present the implementation of this algorithm, which is available for download at `https://github.com/fm-polimi/TACK` alongside the original TACK encoding, as free software.

Empirical testing has revealed that our approach exibits significant speedups across several benchmarks when compared to the previous encoding. The results seem to indicate that our approach is better suited to exploring models with larger bounds, as the time necessary to solver larger and larger bounds grows more slowly when comparised with ae2sbvzot.

References

[1] C. Menghi, M. M. Bersani, M. Rossi, and P. S. Pietro, “Model checking mitl formulae on timed automata: A logic-based approach,” *ACM Trans. Comput. Logic*, vol. 21, Apr. 2020.