Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Master Degree in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

POLITECNICO
MILANO 1863

Improved Verification of Networks of Timed Automata

Supervisor: Prof. Pierluigi San Pietro
Co-supervisors: Prof. Matteo Rossi
Prof. Marcello Bersani

Tesi di laurea di:
Robert Lawrence Smith Matr. 914634

Academic Year: 2019-2020

Many thanks to my supervisors for their guidance and
insightful comments, and a special thank you to my family
for all of their love and support.

And to my friends, thank you for keeping me company
online during the long months of quarantine.

Contents

List of Figures

List of Tables

Abstract

Timed Automata (TA) are a de facto modeling standard for systems with time-sensitive properties. A common task is to verify if a given network of TA satisfy a given property for every possible execution of the network. These questions lend themselves to being expressed in Linear Temporal Logics (LTLs), which allows for expressions over atomic propositions that are defined over time positions in $\mathbb{N}$. We build upon the TA solver TACK, which supports properties expressed in the rich Metric Interval Temporal Logic (MITL), and encodes both the TA network and property to be verified into a variant of Linear Temporal Logic, Constraint LTL over clocks (CLTLoc). The produced CLTLoc formula can then be solved by tools such as Zot, which transform CLTLoc properties into SMT-LIB, a standardized SMT solver language with support for BitVector and real-valued logics.

We present a novel method that preserves TACK's encoding of MITL properties while encoding the Timed Automata network directly into SMT-LIB, making use of both the BitVector logic and the logic of real-valued functions. Our primary targeted SMT solver is Microsoft's Z3, which supports many standardized formats, including SMT-LIB and has strong support for several SMT logics. We introduce several optimizations that allow us to significantly outperform the CLTLoc encoding, and correct deficiencies in the original encoding.

Sommario

Gli Automi temporizzati (TA) sono de facto lo standard di modellazione per sistemi con proprietà dipendenti dal tempo. Un'importante attività è quella di verificare se una data rete di TA soddisfa una data proprietà, per ogni possibile esecuzione della rete. Molte proprietà interassanti si possono esprimere in modo sintetico e efficace in Logica Temporale Lineare (LTL), che permette di predicare su proposizioni atomiche variabili nel tempo. In questa tesi ci basiamo sul TA solver TACK, che supporta proprietà espresse nella logica temporale Metric Interval Temporal Logic (MITL), più espressiva di LTL, e codifica sia la rete TA che la proprietà da verificare in una variante di LTL, Constraint LTL over clocks (CLTLoc). La formula CLTLoc risultante può quindi essere risolta con strumenti di verifica come Zot, che trasformano CLTLoc in SMT-LIB, un linguaggio standardizzato per solutori SMT con supporto per BitVector e logiche reali.

In questa tesi presentiamo un nuovo metodo di verifica che conserva la codifica TACK delle proprietà MITL ma che codifica la rete di TA direttamente in SMT-LIB, utilizzando sia la logica BitVector che la logica delle funzioni a valore reale. Il nostro principale risolutore SMT mirato è lo Z3 di Microsoft, che supporta molti formati standardizzati, compreso SMT-LIB e ha un forte supporto per diverse logiche SMT. Abbiamo introdotto varie ottimizzazioni che ci hanno permesso di migliorare in modo significativo le prestazioni di verifica rispetto alla codifica CLTLoc e di correggere alcune carenze nella codifica originale.

Chapter 1

Introduction

Timed Automata [1] (TA) are a popular tool for modeling time-sensitive systems. By combining the transition semantics of finite 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 encoding languages exist for a variety of applications and use cases. These include the current de facto standard Uppaal [2], as well as NuSMV [3] and MITL$_{0,\infty}$BMC [4].

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. For *invariant* properties, the property holds if every reachable state in the system satisfies the property, and thus the model checker attempts to find a counterexample to falsify the property. Conversely, a *reachability* property asserts that there exists at least one reachable state in which the given property holds, and the model checker proves the property by finding that such a reachable state exists. A reachability property can be solved by asserting its negation as an invariant property (and vice versa).

TACK is a bounded model checker for networks of timed automata developed by Menghi et al [5]. Properties to be verified are specified in Metric Interval Temporal Logic (MITL), and are converted along with the TA network into CLTLoc, a variant of Constraint Linear Temporal Logic supporting real-valued clocks. MITL and CLTLoc are rich logics that allow for continuous-time semantics using traces, a form of timed words, to represent the execution of the networks of TA.

Our contribution is 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. 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 added additional features to make TACK more useful for users. In addition we have 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 will 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 (Chapter 2). We will then present our novel contribution to the problem of encoding the TA network into a form suitable for an SMT bounded model checker (Chapter 3). Afterwords we will present our experimental results (Chapter 4) as well as summary remarks (Chapter 5).

Chapter 2

Preliminaries

In this chapter we present the current TACK bounded model checking procedure, along with the required prerequisite knowledge. We begin with a discussion on the current state-of-the-art in model checking, followed by a discussion on Timed Automata, giving an intuitive introduction to the topic before formally defining the TA used in the rest of the paper. We then discuss Bounded Model Checking, formally defining a *trace* of a TA network and presenting an illustrative example trace. We then discuss the two temporal logics used in the TACK encoding, CLTLoc and MITL. With the theoretical background complete, we move on to discussing the TACK encoding itself, followed by an summary of BitVector logic and ae2sbvzot, the tool used by TACK to convert CLTLoc formulas into BitVector form.

2.1 State of the Art

For many years, model checking was performed using Binary Decision Diagrams (BDDs) [6], which offer many time- and space-complexity advantages over explicit state enumeration [7]. BDDs are symbolic representations of functions over boolean variables that exploit symmetries to avoid explicitly representing the entire state space. However to efficiently handle larger state spaces, BDDs have been largely abandoned in favor of bounded model checking techniques. A representative example is NuSMV [3], which was originally designed to use BDDs but has since been rewritten to use SAT-based bounded model checking. Using the strength of modern-day SAT and SMT solvers, bounded model checking encodes the semantics of the state transition system into SAT or SMT form, and then tasks the solver with finding a valid assignment of states to time positions starting from a given initial state such that the desired reachability property is true (resp. false for invariant properties). 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.

In addition to finding finite traces, bounded model checking has also been applied to finding 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.

To represent state systems with real-time properties, Timed Automata have emerged as a de facto standard [1]. At their core, timed automata are finite state machines that are enriched with real-valued clocks that track the passage of time. These clocks can then be incorporated into transition guards and state invariants to control and synchronize automata. Unfortunately while Timed Automata have had great success at modeling real-time systems, they have not been useful for representing the properties to be verified, and in fact it has been shown that verification using TA to represent properties is undecidable [1]. As a result various temporal logics have been developed by different solvers to represent the properties to be evaluated.

Uppaal [2] is a de facto standard for model checking systems of timed automata. Uppaal supports a subset of Timed Computation Tree Logic (TCTL), an extension of the widely known Computation Tree Logic (CTL) with real-time properties. The CTL family of logics uses the expressive tree-view of future positions, and as such can represent a wide range of universality and existence statements not possible in Linear Temporal Logics. However due to the difficulty of encoding the full semantics of this logic, Uppaal and similar implementations often restrict themselves to a subset of TCTL.

In addition to the work done with branching-time logics, there has been interest in the expressive power of Metric Temporal Logic (MTL), a form of Linear Temporal Logic (LTL) with Interval constraints on the 'until' operator. While powerful, standard MTL properties are undecidable in general for infinite traces [8]. To make MTL tractable, the subset MITL was proposed in 1991 [9], which offers decidable continuous semantics for TAs. MITL is valuable both for its *continuous* time semantics, where the value of a property can be known at every position $p \in \mathbb{R}^+$, and for its ability to capture expressive and complex properties. Examples of MITL solvers include a tool developed by Kindermann et al. [4], which relies on a further subset $\text{MITL}_{0,\infty}$ for property validation, and currently supports both finite and lasso-shaped bounded traces. Another is TACK, which was developed by Menghi et al. [5] and has implemented a bounded model checker for lasso-shaped infinite traces over the full MITL logic. To improve the speed of the verification process, TACK encodes the TA and property to be verified into BitVector logic [10, 11], which is supported by the standardized SMT-LIB language [12]. To avoid the incompleteness of the naive lasso encoding shown by Kindermann [13], both tools use a region-based encoding of the clock values. This is paired with a "non-Zeno" requirement, so that the infinite traces span an infinite amount of time. In the following chapter we will summarize the background and work accomplished in the TACK solver, before then presenting our contribution to the TACK verification tool.

2.2 Timed Automata

Timed Automata are a useful model for many interactions that require precise timing mechanisms [1]. Each Timed Automaton has a set of states, one of which is active at any given moment, much like a Finite State Machine (FSM). 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.

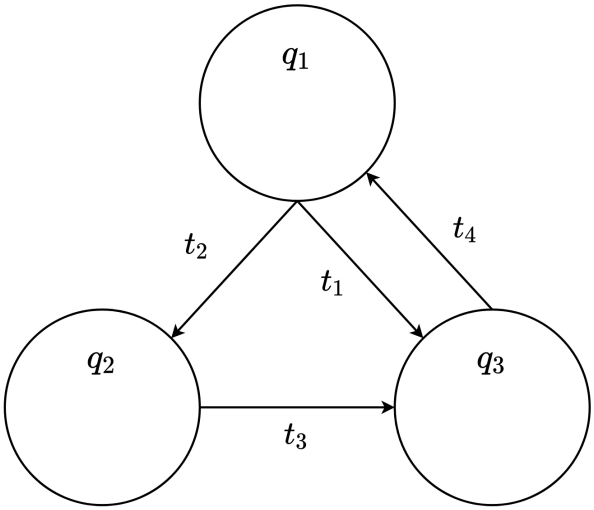


Figure 2.1: A basic Timed Automaton

As we can see in Figure 2.1, Timed Automata have many similarities with Finite State Machines and other automata. This example timed automaton 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-sensitive 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.

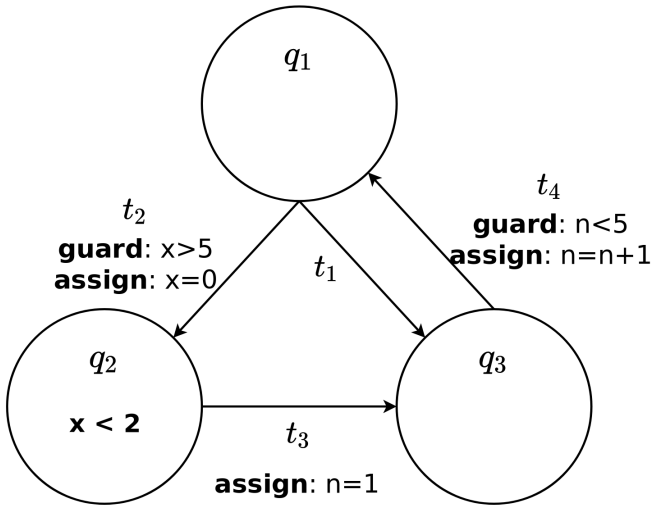


Figure 2.2: A Timed Automaton with clock x and variable n.

We can now show how these values are used and manipulated by a timed automaton. Figure 2.2 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 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 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, which is smaller than the value accepted by the transition guard $x > 5$. To be clear, the value is updated in the same instant as the transition, however the guards only consider the pre-transition value of the clock when determining if the transition is valid. 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 depart state q_2 before clock x reaches a value of 2. In this example the only transition that leads to state q_2 resets the value of clock x, but in general it is illegal to transition into a state if the invariant would be violated upon entry.

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 labeling them with the same synchronization channel, and using the actions to describe the type of synchronization desired.

As shown in Table 2.1, two types of synchronization are defined. The first type is one-to-one synchronization, which has two associated operators, one signifying one-to-one sending (!), and

Type	Synchronization Semantics
One-to-one	Whenever a TA $\mathcal{A}_i$ takes a transition with a synchronization event $\alpha!$, there exists exactly 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 labeled 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 2.1: Supported Synchronization Events

the second for one-to-one receiving (?). A transition labeled with $\alpha!$, for some channel α, can only be fired if at the same moment in time, another TA takes a transition labeled 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 $x \sim c \mid \gamma \wedge \gamma$, 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 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 $\mathcal{A}_i \in \mathcal{A}$.

2.3 Bounded Model Checking

Bounded Model Checking [8] 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 introduce '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 positions, in which case the trace would be illegal, as TA cannot perform multiple simultaneous transitions in our model.

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

Figure 2.3: An example trace through 4 positions.

In Figure 2.3 we can see a trace of the Timed Automaton defined in Figure 2.2, 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 $t[1]$ and δ have been shifted to the right by half of one position, so that the active transition is placed in between its source and destination states. 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, then after 6 time units in q_1 the TA transitions to q_2, remains in q_2 using a null transition, 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 captures the values of clocks and variables *after* any assignments have been applied.

Figure 2.4 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 $x > 5$.

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

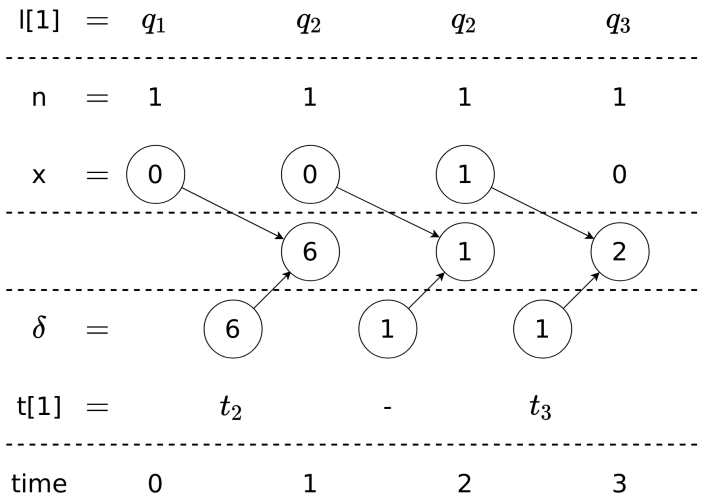


Figure 2.4: A trace highlighting the evaluation of a clock guard.

could include the source state, the destination, both, or neither. Different models of Timed Automata implement this differently. Some, like Uppaal [2] implement so called "super-dense" time semantics, in which the TA may be in multiple states and perform multiple transitions in the same instant of time. 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 interval 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.

Figure 2.5 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 half of 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 2.3.1. A trace η is an infinite sequence

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

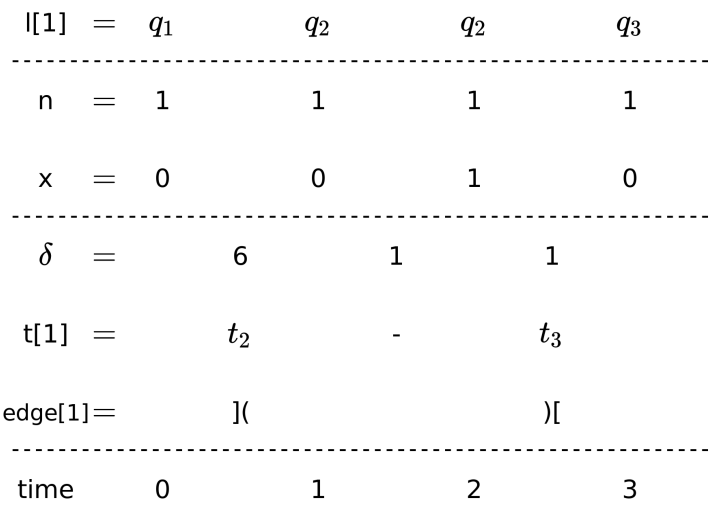


Figure 2.5: An example trace with edge variable.

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 $s = (l, v, v_{var}, \delta, t, edge)$. 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. Although restricting ourselves to only lasso-shaped traces may seem to be a severe limitation, it has been shown that for a given TA network $\mathcal{A}$, there exists a limit K such that if a counterexample for a given property exists, there exists a counterexample of length no more than K [14].

2.4 Constraint LTL Over Clocks

Constraint LTL is an extension of linear temporal logic allowing formulas over a given constraint system [15]. CLTLoc is a constraint LTL where the constraint system consists of clocks defined over the positive real numbers. This allows for the construction of formulas defined over atomic propositions and clocks. 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. In addition CLTLoc has been extended to support expressions over arithmetical variables [16].

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:

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

Where $p \in AP$, $x \in X$, $c \in \mathbb{N}$, $n \in Int$, $\sim \in \{<, =\}$ and exp are arithmetic formulas over integer variables and integers (defined in Section 2.2).

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 $\delta : \mathbb{N} \to \mathbb{R}_{>0}$, 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. $\sigma(l + 1, x) = 0$. 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.

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 $\iota(l, n)$.

A CLTLoc interpretation is the triple (π, σ, ι), where $\pi : \mathbb{N} \to \wp(AP)$ maps time positions to the set of atomic propositions that evaluate to true, and σ and ι are the clock and variable

valuations. A CLTLoc formula ϕ evaluated at time position l is defined as follows:

$$
\begin{aligned}
(\pi,\sigma,\iota),l \vDash x \sim c \quad &\Leftrightarrow \quad \sigma(l,x) \sim c \\
(\pi,\sigma,\iota),l \vDash \exp_1 \sim \exp_2 \quad &\Leftrightarrow \quad \exp_1(\iota,l) \sim \exp_2(\iota,l) \\
(\pi,\sigma,\iota),l \vDash \mathcal{X}(n) \sim \exp \quad &\Leftrightarrow \quad \iota(l{+}1,n) \sim \exp(\iota,l) \\
(\pi,\sigma,\iota),l \vDash p \quad &\Leftrightarrow \quad p \in \pi(l) \\
(\pi,\sigma,\iota),l \vDash \neg\phi \quad &\Leftrightarrow \quad \neg((\pi,\sigma,\iota),l \vDash \phi) \\
(\pi,\sigma,\iota),l \vDash \phi_1 \wedge \phi_2 \quad &\Leftrightarrow \quad ((\pi,\sigma,\iota),l \vDash \phi_1) \wedge ((\pi,\sigma,\iota),l \vDash \phi_2) \\
(\pi,\sigma,\iota),l \vDash \mathcal{X}(\phi) \quad &\Leftrightarrow \quad (\pi,\sigma,\iota),l{+}1 \vDash \phi \\
(\pi,\sigma,\iota),l \vDash \phi_1 \mathcal{U} \phi_2 \quad &\Leftrightarrow \quad ((\pi,\sigma,\iota),l \vDash \phi_2) \vee \\
&\qquad \Big(((\pi,\sigma,\iota),l \vDash \phi_1) \wedge ((\pi,\sigma,\iota),l{+}1 \vDash \phi_1 \mathcal{U} \phi_2)\Big)
\end{aligned}
$$

A CLTLoc formula ϕ is said to be *satisfiable* if an interpretation (π,σ,ι) exists such that $(\pi,\sigma,\iota),0 \vDash \phi$. This is often shortened to simply $(\pi,\sigma,\iota) \vDash \phi$.

2.5 MITL

Metric Interval Temporal Logic is a restriction of Metric Temporal Logic (MTL) such that subscripts must be intervals [9] of non-zero length. An interval I is a convex region of $\mathbb{R}_{\geq 0}$. The bounds of this region must be in the set $\{\mathbb{N} \cup \infty\}$. We will represent an interval as $\langle a,b\rangle$ or $\langle a,\infty)$, where $a,b \in \mathbb{N}$, $\langle \in \{\ (,[\ \}$ and $\rangle \in \{\ ),]\ \}$.

The following grammar describes the set of MITL formulas:

$$\phi := \alpha \mid \phi \vee \phi \mid \neg\phi \mid \phi\, \mathcal{U}_I \phi$$

Where α represents the set of atomic propositions. The set of propositions currently supported by TACK consists of the set AP (atomic propositions of the network of TA) and propositions of the form $n = c$, where n is an integer variable and c is a constant value.

The semantics of MITL are defined as follows. An MITL signal is a function $M : \mathbb{R}_{\geq 0} \to \mathcal{P}(AP) \times \mathbb{Z}^{Int}$. At a given time t, $M(t) = P, v_{var}$ gives us the set of AP that evaluate to true P, as well as a valuation for the integer variables v_{var}. For a given signal M, $M,t \vDash \phi$ is defined

as follows:

$$
\begin{aligned}
M, t \vDash \alpha \quad &\Leftrightarrow \quad M(t) = (\psi, v_{var}) \wedge \alpha \in \psi \\
M, t \vDash n = c \quad &\Leftrightarrow \quad M(t) = (\psi, v_{var}) \wedge v_{var}(v) = c \\
M, t \vDash \phi_1 \wedge \phi_2 \quad &\Leftrightarrow \quad (M, t \vDash \phi_1) \wedge (M, t \vDash \phi_2) \\
M, t \vDash \neg\phi \quad &\Leftrightarrow \quad \neg(M, t \vDash \phi) \\
M, t \vDash \phi_1 \mathcal{U}_I \phi_2 \quad &\Leftrightarrow \quad \exists t' > t, t' - t \in I, (M, t \vDash \phi_2) \wedge \forall t'' \in (t, t'), (M, t'' \vDash \phi_1)
\end{aligned}
$$

An MITL formula ϕ is said to be *satisfiable* if an interpretation M exists such that $M, 0 \vDash \phi$. We then say that M models ϕ.

2.6 TACK CLTLoc Translation

The TACK [5] 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-LIB 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 $l[i]$ and $t[i]$ return the active state and active transition 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 and variables can be represented directly as CLTLoc clocks and variables.

Table 2.2 contains the formulas used to encode the Timed Automata into CLTLoc. To accomplish this encoding, several auxiliary 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 $i + 1$.

Table 2.2: TACK Encoding of an Automaton in CLTLoc

$\varphi_1 := \bigwedge_{i\in[1,N]} (l[i] = 0)$	$\varphi_2 := \bigwedge_{n\in Int} n = v^0_{var}(n)$	$\varphi_3 := \bigwedge_{i\in[1,N]} Inv(l[i])$
$\varphi_4 := \bigwedge_{x\in X} (x_0{=}0 \wedge x_1{>}0 \wedge x_v{=}0)$	$\varphi_5(j) := \bigwedge_{x\in X} (x_j{=}0) \rightarrow \mathcal{X}\left((x_{(j+1) \bmod 2} = 0)\mathcal{R}\left((x_v{=}j)\wedge(x_j{>}0)\right)\right)$	

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

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

$$\varphi_{edge}(a, b, i) := \varphi_{\alpha RC}(a, b, i) \vee \varphi_{\alpha LC}(a, b, i)$$
$$\varphi_{\alpha RC}(a, b, i) := Inv(a) \wedge r_2(Inv_w(b)) \wedge edge^{RC}[i]$$
$$\varphi_{\alpha LC}(a, b, i) := Inv_w(a) \wedge r_2(Inv(b)) \wedge \neg edge^{RC}[i]$$

$$\varphi_8 := \bigwedge_{i\in[1,N];q,q'\in\mathcal{Q}_i|q\neq q'} \left(\left((l[i] = q) \wedge \mathcal{X}(l[i] = q')\right) \rightarrow \bigvee_{t\in T_i|t_-=q,t_+=q'} (t[i] = t)\right)$$

$$\varphi_9 := \bigwedge_{x\in X} \left(\mathcal{X}(x_0 = 0 \vee x_1 = 0) \rightarrow \bigvee_{\substack{i\in[1,N]\\ t\in T_i|x\in t_{a_c}}} t[i] = t\right)$$

$$\varphi_{10} := \bigwedge_{n\in Int} \left((\neg(n = \mathcal{X}(n))) \rightarrow \bigvee_{\substack{i\in[1,N]\\ t\in T_i|n\in t_{a_v}}} t[i] = t\right)$$

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.

Formulas 4 and 5 are the clock constraints, and describe how the active value of the clock evolves throughout the trace. To both test the value of a clock and simultaneously reset the value during a transition, TACK has used two clock variables to encode a single clock value. There is currently a work in progress to extent CLTLoc to support simultaneous test and reset, however at the time of writing this is not published. For a clock x, x_v holds the index of the active clock value, and references to the clock value elsewhere are syntactic sugar for evaluating this variable and then choosing the appropriate clock value. The active value is never zero, and a clock reset at position i is not reflected in the formula until position $i{+}1$.

Formula φ_6 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. The 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 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. Function r_2 again ensures that in the event of a clock reset, the the invariants of the destination state are evaluated against the correct clock values.

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.

Type		**Synchronization Encoding**
One-to-one	$v_1 :=$	$\bigwedge_{\substack{i\in[1,N]\\ t\in T_i \mid t_\epsilon=\alpha!}} \left(t[i]=t \rightarrow \bigvee_{\substack{j\in[1,N],\\ j\neq i}} \left(\begin{array}{c} \varphi_{\textit{sync-on}}(j,\alpha?) \wedge \neg\varphi_{\textit{sync-on-but}}(\{i,j\},\alpha?) \\ \wedge \\ \varphi_{\textit{same-edge}}(i,j) \end{array} \right) \right)$
	$v_2 :=$	$\bigwedge_{\substack{i\in[1,N]\\ t\in T_i \mid t_\epsilon=\alpha?}} \left(t[i]=t \rightarrow \bigvee_{\substack{j\in[1,N],\\ j\neq i}} (\varphi_{\textit{sync-on}}(j,\alpha!) \wedge \neg\varphi_{\textit{sync-on-but}}(\{i,j\},\alpha!)) \right)$
Broadcast	$v_1 :=$	$\bigwedge_{\substack{i\in[1,N]\\ t\in T_i \mid t_\epsilon=\alpha\#}} (t[i]=t \rightarrow (\neg\varphi_{\textit{sync-on-but}}(\{i\},\alpha\#)))$ $\bigwedge_{\substack{i\in[1,N]\\ t\in T_i \mid t_\epsilon=\alpha\#}} \left(t[i]=t \rightarrow \left(\bigwedge_{\substack{h\in[1,N]\\ j\neq i}} \left(\begin{array}{c} \wedge \\ \varphi_{\textit{sync-on}}(j,\alpha@) \wedge \varphi_{\textit{same-edge}}(i,j) \\ \vee \\ \left(\bigwedge_{\substack{t'\in T_{i'},\\ t'_\epsilon=\alpha@}} (\mathcal{X}(\neg\phi_{t'_{\gamma_c}}) \vee \neg\phi_{t'_{\gamma_v}} \vee l[j]\neq t'_-) \right) \end{array} \right) \right) \right)$
	$v_2 :=$	$\bigwedge_{\substack{i\in[1,N]\\ t\in T_i \mid t_\epsilon=\alpha@}} (t[i]=t \rightarrow \varphi_{\textit{sync-on-but}}(\{i\},\alpha\#))$
$\varphi_{\textit{sync-on}}$	$(j,\alpha) :=$	$\bigvee_{t\in T_i \mid t_\epsilon=\alpha} (t[i]=t)$
$\varphi_{\textit{sync-on-but}}$	$(S,\alpha) :=$	$\bigvee_{g\in\{i \mid i\in[1,N]\}\setminus S} \varphi_{\textit{sync-on}}(h,\alpha)$
$\varphi_{\textit{same-edge}}$	$(i,j) :=$	$\mathcal{X}(edge^{RC}[i] \leftrightarrow edge^{RC}[j])$

Table 2.3: Encoding of Different Synchronization Types

Table 2.3 contains the encodings for the two supported synchronization types. These synchronization semantics are defined in Table 2.1. 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 two automatons have the same edge type.

Liveness Constraints When we introduced the concept of traces, we gave examples in which at least one Timed Automata performed a non-null (or discrete) transition at every position in the trace. It seems natural to require at least one discrete transition in each position, because otherwise the state of the network would not change, and we could simply remove the unnecessary duplicate position from the trace. Unfortunately the addition of MITL properties makes this a bit more complex. Consider the MITL property $\mathcal{G}_{[0,\infty)}\neg(\mathcal{G}_{[0,10]}\ \alpha)$, where α is an atomic proposition which only evaluates to true in state q. This property states that a TA is never in state q for 10 or more time units before leaving. Now let us consider a hypothetical counterexample. In

trace η, the TA transitions into state q at position i, and then transitions to another state in position $i{+}i$, with $\delta(i) = 20$. This clearly violates the property, as the TA remained in state q for 20 seconds. However in order to detect this violation, the MITL encoding requires there to be a specific position in which the property is violated. Unfortunately position $i{+}1$ does not suffice, as the TA is no longer in the critical state q. This counterexample can only be detected if the solver has the ability to add another time position in-between i and $i{+}1$, where at least 10 units of time has passed **and** the TA is still located in state q. Thus we must tolerate positions in which every TA performs a null transition. That being said, we often do not wish to accept traces in which discrete transitions **never** occur. These requirements are termed *liveness constraints*.

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 2.4: Possible Liveness Constraints

In addition to the TA and synchronization constraints, TACK includes support for optional liveness constraints. The first is termed Strong Transition Liveness, which is shown in Table 2.4. This guarantees that every time position, eventually in the future all timed automaton in the network will take a discrete transition. Although a weaker version termed Weak Transition Liveness is defined in the TACK paper, it is not implemented in the TACK tool. It requires that at every position *at least one* timed automaton will eventually perform a discrete transition. Finally the Unrestricted Liveness option places no restrictions on TA liveness.

Type	**Edge Semantics**
Right-closed	$\bigwedge_{i\in[1,N]} \mathcal{G}(edge^{RC}[i])$
Open	$\top$

Table 2.5: Possible Edge Constraints

Edge Constraints In addition to liveness, TACK also allows for customization of its edge constraints. As we have seen in Section 2.3, 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 the option to set all edge variables to right-closed in every time position. As seen in Table 2.5, this is contrasted with the 'open' edge semantics, which do not place any additional restrictions on the edge variables.

2.7 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 $(\overleftarrow{1100} \Rightarrow \overleftarrow{1010})$ would evaluate to $\overleftarrow{1101}$, since $a \rightarrow b$ is equivalent to $a \vee \neg b$.

2.8 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 2.6: 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 2.6 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 are constrained to 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 2.7: ae2sbvzot definition of a proposition φ

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

Given a formula with a bound of k, ae2sbvzot constructs a BitVector for the formula and for each sub-formula 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 sub-formulas based on the rules in Table 2.7. 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. Note the use of the *reduction or* operator $\Uparrow$, which has a value of 0 if the BitVector argument is exactly zero, 1 otherwise.

Atomic propositions (including the active states, transitions, edges 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.

To reduce the number of variables used, rather than store each transition (respectively state, variable value) as a separate BitVector, ae2sbvzot uses a more compact bit-based representation. We will use transitions to illustrate this encoding, which is identical for states and variables as well. Since only one transition is active at a time, it is more compact to store the currently

active transition as a binary number over $\lceil \log_2 |T| \rceil$ bits. Therefore ae2sbvzot creates $\lceil \log_2 |T| \rceil$ BitVectors of length $k+2$ to represent the active transition of the TA over time. Table 2.8 shows how these BitVectors are constructed to represent the active transition of a single timed automaton. To reconstruct a BitVector whose bits have the value of 1 iff a given transition t is active, ae2sbvzot combines the multiple transition BitVectors in a unique fashion to represent a specific transition. As an example, consider a transition $t \in T_i$ that has been encoded as the number 1. Since the binary representation of 1 is $\overleftarrow{00\ldots}001$, to create the BitVector for transition t ae2sbvzot would construct the following expression:

$$!\overleftarrow{tb_{i,\lceil \log_2 \mathcal{T}_i \rceil - 1}} \& \; !\overleftarrow{tb_{i,\lceil \log_2 \mathcal{T}_i \rceil - 2}} \; \& \ldots \& \; !\overleftarrow{tb_{i,1}} \; \& \; \overleftarrow{tb_{i,0}}$$

In this expression every BitVector is negated except for the last, which corresponds to the number one. Each transition (resp. state, variable value) is assigned a combination of the corresponding BitVectors in this fashion.

Clocks, being real-valued functions, cannot be easily represented in BitVector logic. Instead ae2sbvzot takes advantage of SMT-LIB'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 construct larger expressions using the rules shown in Table 2.7.

Unlike the other terms, clocks cannot simply be constrained to hold the same value in the loopback position. As shown by Kindermann [13] this excludes certain valid lasso shaped traces. To avoid this ae2sbvzot adopts the region-based clock constraints suggested by Kindermann. Two clock valuations are said to be in the same region if at both positions *loop* and $k{+}1$:

- Each clock x is either greater than the maximum value compared against x in a clock guard in both positions ($\lfloor x \rfloor > \max(x)$) or $\lfloor x(loop) \rfloor = \lfloor x(k{+}1) \rfloor$
- If $\lfloor x \rfloor \leq \max(x)$, the formula $x(l) - \lfloor x(l) \rfloor = 0$ must have the same value in both *loop* and $k{+}1$
- For every pair of clocks x, x', if the floor of both clocks are less than their maximum values, then the formula $frac(x(l)) < frac(x'(l))$ must have the same value in positions *loop* and $k{+}1$, where $frac$ is the fractional part of the clock's value.

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

Table 2.8: Construction of the Transition BitVectors for $\mathcal{A}_i \in \mathcal{A}$

These conditions are sufficient to ensure that the clock valuation in each position belongs to the same region, a concept expanded on in Kindermann's work [13]. This allows ae2sbvzot to represent loops where the elapsed time of each loop iteration is constrained to be less than the elapsed time of the previous iteration. Although a lasso with a fixed amount of time per-loop iteration does not exist, there exists infinite series of loop iterations, each with a decreasing elapsed time, that nonetheless have a diverging sum, and thus represent infinite traces.

Since the time per-loop iteration is no longer fixed, another constraint is required to ensure that no "Zeno-shaped" traces are allowed. In Zeno traces, the time elapsed in the infinite trace sums to a finite number. These cases are considered pathological and to exclude them, it has been proven [13] that it is sufficient to require that in the loop section of the trace, that every clock is either

- Reset at least once during the loop
- or has a value greater than $max(x)$ in the final loop position.

This concludes the ae2sbvzot encoding of a CLTLoc formula.

Chapter 3

Novel Encoding of Timed Automata Networks

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, named ta2smt, 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. This chapter is organized into three parts. In the first, we describe the various terms that make up our TA network, and discuss how they are encoded into BitVector logic. In the second, we present the constraints, defined over the terms from the first section, that capture the TA semantics. In the final section we argue for the correctness of our model, and highlight improvements made over the original encoding. For ease of reading we will refer to the old encoding as ae2sbvzot when contrasting it with ta2smt, even though ae2sbvzot is only one part of the CLTLoc-based encoding of the network.

3.1 BitVectors

Like ae2sbvzot, our novel encoding (ta2smt) is based on encoding the terms of the Timed Automata into BitVectors. 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

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 3.1: Construction of the Transition Aliases

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.

3.1.1 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 state $q \in Q_i$. We define these null transitions as follows:

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

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 $\mathcal{T}_i = T_i \cup \{ \underset{q\in Q_i}{\cup} t_{null_q} \}$ for each TA $\mathcal{A}_i$. As before $\mathcal{T}$ is the union of the $\mathcal{T}_i$ sets. The motivation for this redefinition will become clear when we discuss the encoding of the active states of the TA.

To encode our expanded set of transitions, we adopt the more compact bit-based representation used in ae2sbvzot and shown in Table 2.8. In order to be able to conveniently refer to individual elements of the set, we will also 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 3.1. 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.

3.1.2 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.

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

This is made possible by our addition of $|Q_i|$ null transitions for each TA i. This allows us to assign each null transition a source and destination state, which in turn allows us define the current state as the source of the active transition. 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 was active, it was not possible to determine the active states without referring to the state BitVector.

3.1.3 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 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.

3.1.4 Clocks

Our encoding of the clocks does not differ from ae2sbvzot. 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.

3.1.5 Complete Encoding

Table 3.2: Terms and Aliases used in BV encoding of TA

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	$\underset{x\in X}{\forall}$	$x : [0, k+1] \to \mathbb{R}_{\geq 0}$
δ		$\delta : [0, k+1] \to \mathbb{R}_{>0}$
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	$\underset{i\in[1,N]}{\forall}\ \underset{t\in[1,\mathcal{T}_i]}{\forall}$	$\overleftarrow{trans_t} = \underset{j\in[0,\lceil\lvert\log_2\mathcal{T}_i\rvert\rceil-1]}{\&} !^{(t\setminus 2^j) mod\ 2}(!(\overleftarrow{tb_{i,j}}))$
States	$\underset{i\in[1,N]}{\forall}\ \underset{q\in Q_i}{\forall}$	$\overleftarrow{state_q} = \underset{t\in\mathcal{T}_i}{\mid} \overleftarrow{trans_t}$
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	$\underset{l\in[0,k+1]}{\forall}$	$\left(\overleftarrow{inloop}^{[l]}_{[k+2]} = \overleftarrow{1}_{[1]}\right) \leftrightarrow (\overleftarrow{loop} \leq \overleftarrow{l})$

Note: the operator \ is used to represent integer division.

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 3.2. 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.

3.2 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.

3.2.1 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 constraints over the BitVector terms that make up our encoding. Also included are constraints on how these three groups of values (transitions, variables, and clocks) evolve throughout the trace.

Initialization and Progression Constraints

$\phi_1 := \bigwedge_{i\in[1,N]} \overleftarrow{1}_{[1]} = \overleftarrow{state_{q_i^0}}^{[0]}$	$\phi_2 := \bigwedge_{n\in Int} \overleftarrow{v^0_{var}(n)} = \overleftarrow{var_n(0)}$	$\phi_3 := \bigwedge_{x\in X} x(0) = 0$
$\phi_4 := \bigwedge_{i\in[0,k+1]} \delta(i) > 0$	$\phi_5 := \bigwedge_{t\in\mathcal{T}} (\overleftarrow{trans_t}^{[k:0]} \Rightarrow \overleftarrow{state_{t_+}}^{[k+1:1]})$	$\phi_6 := \{\phi_{wtl} \mid \phi_{stl} \mid \top\}$
$\phi_7 := \bigwedge_{x\in X} \bigwedge_{j\in[0,k]} \left(\&_{t\in\mathcal{R}(x)} (!\overleftarrow{trans_t})^{[j]} \right) \rightarrow x(j+1) = x(j) + \delta(j)$		
$\phi_8 := \&_{n\in Int} \&_{t\in assign(n)} (!\ \overleftarrow{trans_t}^{[k:0]}) \Rightarrow \&_{j\in[1,\lambda(n)]} (\overleftarrow{v_n b_j}^{[k:0]} \Leftrightarrow \overleftarrow{v_n b_j}^{[k+1:1]})$		
$\phi_{wtl} := \overleftarrow{0}_{[k+2]} \neq \left(\overleftarrow{inloop} \ \& \ (\ \mid_{i\in[1,N]} ! (\mid_{q\in Q_i} \overleftarrow{trans_{null_q}})) \right)$		
$\phi_{stl} := \bigwedge_{i\in[1,N]} \overleftarrow{0}_{[k+2]} \neq \left(\overleftarrow{inloop} \ \& \ ! (\mid_{q\in Q_i} \overleftarrow{trans_{null_q}}) \right)$		
$\phi_{Init} := \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \wedge \phi_4 \wedge \phi_5 \wedge \phi_6 \wedge \phi_7 \wedge (\phi_8 =! \overleftarrow{0}_{[k+2]})$		

Table 3.3: Initialization and Progression Constraints for a network of timed automata

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 constraints is ϕ_6, which encodes the desired *liveness property* of the network, discussed in 2.4. 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 $!(\bigm|_{q \in Q_i} \overleftarrow{trans_{null_q}})$, 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

Transition Constraints, Assignments, and Invariants

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

Table 3.4: Transition Constraints for a network of timed automata

value of n remains unchanged.

3.2.2 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 3.2.1 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 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.

3.2.3 Sync

The synchronization constraints capture the semantics defined in Table 2.1. Formula ϕ_{14} and ϕ_{15} capture the constraints 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

Sync Constraints

$\phi_{14} :=$	$\underset{t\in T:t_\epsilon=\alpha!}{\&} \overleftarrow{trans_t} \Rightarrow (\neg \underset{\substack{t'\in\{T\setminus t\}:\\ t'_\epsilon\in\{\alpha!,\alpha\#\}}}{\mid} \overleftarrow{trans_{t'}}) \wedge (\underset{t'\in T:t_\epsilon=\alpha?}{\mid} \overleftarrow{trans_{t'}})$
$\phi_{15} :=$	$\underset{\substack{i\in[1,N]\\ t\in T_i:t_\epsilon=\alpha?}}{\&} \overleftarrow{trans_t} \Rightarrow (\neg \underset{\substack{t'\in\{T\setminus t\}:\\ t_\epsilon=\alpha?}}{\mid} \overleftarrow{trans_{t'}}) \wedge (\underset{\substack{j\in[1,N]\\ t'\in T_j:\\ t_\epsilon=a!}}{\mid} \overleftarrow{trans_{t'}} \,\&\, (\overleftarrow{edge_i^{RC}} \Leftrightarrow \overleftarrow{edge_j^{RC}}))$
$\phi_{16} :=$	$\underset{t\in T:t_\epsilon=\alpha\#}{\&} \overleftarrow{trans_t} \Rightarrow \neg(\underset{t'\in T:t_\epsilon=\alpha\#\wedge t'\neq t}{\mid} \overleftarrow{trans_{t'}})$
$\phi_{17} :=$	$\underset{\substack{i\in[1,N]\\ t\in T_i:t_\epsilon=\alpha@}}{\&} \overleftarrow{trans_t} \Rightarrow (\underset{\substack{j\in[1,N]\\ t'\in T_j:t_\epsilon=\alpha\#}}{\mid} \overleftarrow{trans_{t'}} \,\&\, (\overleftarrow{edge_i^{RC}} \Leftrightarrow \overleftarrow{edge_j^{RC}}))$
$\phi_{18} :=$	$\bigwedge_{\alpha\in Act} \bigwedge_{\substack{i\in[1,\lvert\mathcal{A}\rvert]\\ l\in[0,k+2]}} (\bigvee_{\substack{t\in T_i:\\ t_\epsilon=\alpha\#}} \overleftarrow{trans_t}^{[l]}) \rightarrow$ $(\bigwedge_{\substack{j\in[1,k]:\\ j\neq i}} (\bigvee_{\substack{t'\in T_j:\\ t'_\epsilon=\alpha@}} \overleftarrow{state_{t'_-}}^{[l]} \wedge \sigma_\delta(l,t'_{\gamma_c}) \wedge \mu(l,t_{\gamma_v})) \rightarrow \bigvee_{\substack{t'\in T_j:\\ t'_\epsilon=\alpha@}} \overleftarrow{trans_{t'}}^{[l]})$
$\phi_{sync} :=$	$(\phi_{14} \,\&\, \phi_{15} \,\&\, \phi_{16} \,\&\, \phi_{17} = !\overleftarrow{0}) \wedge \phi_{18}$

Table 3.5: Synchronization Constraints for a network of timed automata

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 labeled with the $\alpha@$ (broadcast receive) event. ϕ_{17} requires 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 $\alpha@$.

Loop Constraints

$\phi_{19} :=$	$\bigwedge_{i\in[1,N]} (\overleftarrow{edge_i^{RC}}[k+1] = \overleftarrow{edge_i^{RC}}[loop]) \wedge \bigwedge_{j\in[1,\lceil\log_2\lvert\mathcal{T}_i\rvert\rceil]} \overleftarrow{tb_{i,j}}[k+1] = \overleftarrow{tb_{i,j}}[loop]$
$\phi_{20} :=$	$\bigwedge_{n\in Int} \bigwedge_{j\in[1,\lambda(n)]} \overleftarrow{vb_{n,j}}[k+1] = \overleftarrow{vb_{n,j}}[loop]$
$\phi_{21} :=$	$\bigwedge_{x\in X} (\lfloor x(k+1)\rfloor = \lfloor x(loop)\rfloor) \vee (\lfloor x(k+1)\rfloor > \max(x) \wedge \lfloor x(loop)\rfloor > \max(x))$
$\phi_{22} :=$	$\bigwedge_{x\in X} \lfloor x(loop)\rfloor \leq \max(x) \Rightarrow (frac(x(k+1)) = 0) \Leftrightarrow (frac(x(loop)) = 0)$
$\phi_{23} :=$	$\bigwedge_{x,x'\in X} (\lfloor x(loop)\rfloor \leq \max(x) \wedge \lfloor x'(loop)\rfloor \leq \max(x')) \rightarrow \Big(frac(x(k+1)) \leq frac(x'(k+1)) \leftrightarrow frac(x(loop)) \leq frac(x'(loop))\Big)$
$\phi_{24} :=$	$\bigwedge_{x\in X} x(k) > \max(x) \vee ((\bigvee_{t:x\in t_{a_c}} \overleftarrow{trans_t}) \,\&\, \overleftarrow{inloop} \neq \overleftarrow{0})$
$\phi_{loop} :=$	$\phi_{19} \wedge \phi_{20} \wedge \phi_{21} \wedge \phi_{22} \wedge \phi_{23} \wedge \phi_{24}$

Table 3.6: Loop Constraints for a network of timed automata

3.2.4 Loop Constraints

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.

Propositions $\phi_{19} - \phi_{20}$ For transitions, edges and variables this is very straightforward. We simply require that in the *loop* and $k{+}1$ positions the active transitions, edge types, and variables have identical values.

Propositions $\phi_{21} - \phi_{23}$ It is tempting to encode the clock constraints in a similar manner, requiring that $x(k+1) = x(loop)$ for each clock. However, as discussed in Section 2.8, this encoding is not complete as it fails to capture certain runs. Instead we use the same region-based clock constraints used in ae2sbvzot.

To begin, for each clock x we define the non-negative integer $\max(x)$, which is equal to the maximum value 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 $x(loop)$ 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.

Property ϕ_{24} As mentioned in Section 2.8, we must exclude the "Zeno traces" from consideration because while they are infinite traces, they execute in a finite amount of time. We encode the same constraint used in ae2sbvzot. 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 $i \geq loop$. Using this vector, we can determine if a given clock is reset within the loop portion of the trace.

3.3 Equivalence and Improvements

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.

Similarly, the original TACK encoding contains three constraints that assert that the values of the active states, as well as the values of the variables and clocks, can only be changed if there is an active transition that modifies them. For states, this is accomplished with ϕ_5, which requires that the active state in the following position be equal to the value of the destination

state 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 the for the state to change, there must be a non-null transition to enable the state change. For variables, ϕ_8 asserts that when no transition explicitly changes the value of a variable, its value must remain the same. ϕ_7 asserts the same for the clocks. Our new encoding also respects the same loop constraints as ae2sbvzot including the extended clock constraints described in Section 3.2.4 necessary to represent all possible lasso-shaped traces.

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 constraints of the form $n \sim c$ or $n \sim n'$, where $\sim \notin \{=\}$ 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. Another deficiency in the original TACK implementation was a previously unknown error in the encoding of broadcast synchronization symbols, which were added to the TACK CLTLoc encoding recently. While the CLTLoc encoding of broadcast signals is still a work in progress, our experimental evaluation adds confidence that our ta2smt encoding correctly handles these signals.

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.

Chapter 4

Evaluation

The modular design of TACK aided us greatly in adding our novel 'ta2smt' encoding to the existing TACK codebase. However due to the removal of CLTLoc as an intermediate language, some adjustments were required to integrate our solution with the existing solvers. Previously TACK assumed that an execution would countain five main steps:

1. The TA Network file and MITL property file are read from the input files and parsed.
2. The TA Network is converted into CLTLoc.
3. The MITL property is converted into CLTLoc.
4. The two CLTLoc formulas are combined using a CLTLoc 'and' expression.
5. The combined CLTLoc property is written to disk and the external converter (ae2sbvzot) is invoked, which in turn invokes an SMT solver (Z3).

While TACK was designed so that the converters used in steps 2 and 3 (and the external program used in step 5) could be easily modified, it was not anticipated that this general execution plan would need to change. However our implementation modifies this plan slightly as follows:

1. The TA Network file and MITL property file are read from the input files and parsed.
2. The TA Network is converted into SMT-LIB.
3. The MITL property is converted into CLTLoc.
4. The CLTLoc formula representing the MITL property is written to disk, and ae2sbvzot is invoked to convert the formula into SMT-LIB (without invoking Z3).
5. The two SMT-LIB files are combined, and Z3 is invoked to check for satisfiabiltiy.

We have therefore modified TACK to first determine which solution method is desired, and to then execute one of the two possible 'solution plans' outlined above. Although currently ta2smt

is the only algorithm available to execute step 2 in the new plan, we have continued in the spirit of the original TACK design to make it simple for this algorithm to be modified or replaced, so that future improvements are encouraged.

In this chapter we present the results of several experimental evaluations of the bounded model checking 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. For both ae2sbvzot and ta2smt, strong transition liveness was used in all of the tests. In addition 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 time provided measures the combined time taken by both the TACK program to parse the problem and convert it to SMT form and for 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 less than a second. 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 AMD EPYC 7551 CPU (2.5 GHz) with 2 32-core sockets, 500 GB of RAM and Debian Linux (version 4.19). It should be noted that the SMT solver is unusual among modern programs in that it is a highly CPU-bound application. 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 we believe that very similar results could be obtained on a machine with more reasonable resources. In addition, 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 Z3 process terminates when either thread terminates, which means that the times reported here are in fact the shortest time of two runs. This was done to reduce instabilities in the solver and to present a clearer comparison between the two different algorithms.

4.1 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.

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

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.

As we can see, the results printed in Tables 4.1 and 4.2 show the difficulty curves for each problem, as both solvers take considerably more time to solve problems with larger bounds and with a larger number of TA in the network. To better visualize these results, we have condensed the tables above into Figure 4.1, which computes the average (geometric mean) speedup for a given property and bound. The speedup is defined as the time taken by ae2sbvzot divided by the time taken by ta2smt. A speedup of 1 indicates that both times are equivalent, values larger than one (resp. less than one) represent a faster time by ta2smt (resp. ae2sbvzot). The speedup

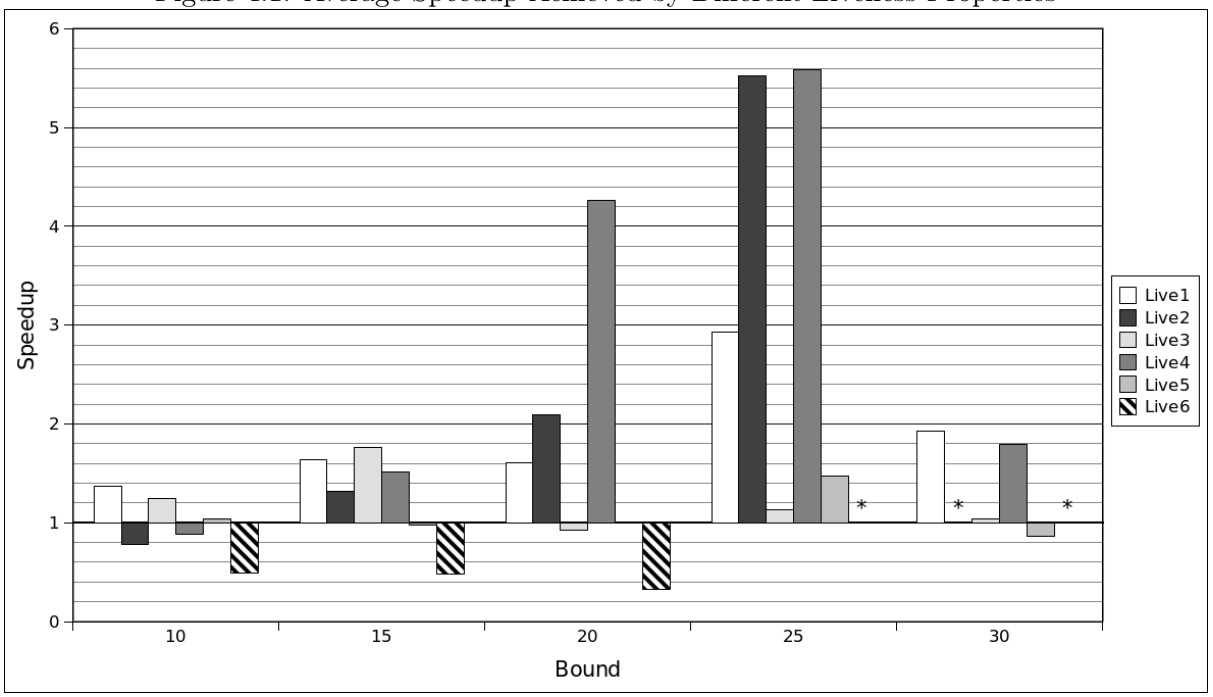
Table 4.1: Time required to solve the Fischer Properties (ae2sbvzot). A ✓ indicates the property is satisfied, a ✗ indicates that a counterexample was found. Blank entries indicate no result after two hours.

		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✓	–	–	–

Table 4.2: Time required to solve the Fischer Properties (ta2smt). A ✓ indicates the property is satisfied, a ✗ indicates that a counterexample was found. Blank entries indicate no result after two hours.

		TACK ta2smt								
		n								
	k	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
live-one	**10**	0.8✓	0.9✓	0.9✓	1.0✓	1.0✓	1.2✓	1.1✓	1.0✓	1.2✓
	15	0.8✓	0.9✓	1.0✓	1.0✓	1.1✓	1.2✓	1.2✓	1.2✓	1.3✓
	20	0.9✓	1.0✓	1.1✓	1.2✓	1.2✓	1.5✓	1.6✓	1.5✓	1.4✓
	25	1.0✓	1.0✓	1.2✓	1.2✓	1.3✓	1.5✓	1.8✓	1.7✓	2.7✓
	30	1.1✓	1.1✓	1.3✓	1.3✓	2.2✓	2.3✓	1.8✓	2.1✓	2.3✓
live-two	**10**	1.6✓	1.8✓	1.8✓	2.5✓	3.0✓	3.5✓	3.0✓	4.3✓	4.0✓
	15	4.8✓	5.3✓	6.6✓	6.0✓	9.1✓	12.4✓	34.5✓	28.6✓	229.7✓
	20	10.2✓	13.2✓	14.4✓	21.6✓	27.4✓	28.8✓	61.2✓	72.3✓	108.6✓
	25	20.9✓	28.9✓	25.5✓	47.6✓	52.7✓	50.6✓	122.3✓	235.3✓	106.6✓
	30	36.1✓	37.9✓	56.9✓	79.4✓	87.2✓	123.7✓	198.4✓	146.9✓	283.4✓
live-three	**10**	0.9✗	1.0✗	1.4✗	1.7✗	2.4✗	24.5✓	26.0✓	16.1✓	16.1✓
	15	1.0✗	1.3✗	2.2✗	2.4✗	3.6✗	11.9✗	10.0✗	23.0✗	26.4✗
	20	1.1✗	2.1✗	3.3✗	5.4✗	9.0✗	13.7✗	23.5✗	24.6✗	53.9✗
	25	2.3✗	2.3✗	8.3✗	7.1✗	14.7✗	26.1✗	90.8✗	647.5✗	467.8✗
	30	1.9✗	4.6✗	5.9✗	9.9✗	17.3✗	32.3✗	37.4✗	73.2✗	93.2✗
live-four	**10**	1.2✓	1.4✓	1.6✓	2.2✓	2.1✓	2.4✓	2.9✓	2.5✓	2.7✓
	15	2.5✓	3.2✓	4.2✓	4.9✓	6.1✓	9.0✓	10.6✓	26.7✓	287.2✓
	20	6.1✓	6.1✓	7.3✓	12.9✓	13.8✓	20.7✓	30.0✓	30.9✓	56.9✓
	25	17.8✓	15.4✓	16.7✓	19.8✓	24.1✓	39.1✓	60.6✓	232.2✓	108.8✓
	30	41.8✓	21.8✓	37.7✓	44.2✓	69.9✓	67.7✓	133.5✓	156.4✓	161.0✓
live-five	**10**	1.0✗	1.3✗	1.5✗	4.2✗	4.2✗	17.2✓	14.1✓	19.1✓	21.0✓
	15	1.1✗	1.7✗	2.5✗	3.3✗	5.2✗	8.1✗	18.0✗	20.6✗	29.1✗
	20	1.4✗	3.1✗	4.6✗	7.4✗	10.8✗	16.9✗	21.6✗	22.0✗	56.9✗
	25	2.1✗	3.6✗	4.5✗	10.0✗	18.4✗	28.2✗	44.0✗	259.8✗	445.3✗
	30	2.5✗	11.1✗	8.9✗	20.0✗	22.8✗	39.4✗	66.0✗	80.0✗	116.7✗
live-six	**10**	1.0✓	1.4✓	2.2✓	6.4✓	13.1✓	32.9✓	27.1✓	14.2✓	15.0✓
	15	1.3✓	3.0✓	9.5✓	24.4✓	61.4✓	95.7✓	170.8✓	346.7✓	543.9✓
	20	1.5✓	8.2✓	53.6✓	141.8✓	368.0✓	756.6✓	1896.7✓	2808.3✓	6662.7✓
	25	2.3✓	45.2✓	122.6✓	719.1✓	1597.1✓	3368.9✓	–	–	–
	30	3.1✓	99.1✓	440.3✓	823.2✓	2724.4✓	–	–	–	–

Figure 4.1: Average Speedup Achieved by Different Liveness Properties



For a given liveness property and bound, the geometric mean of the speedup is computed over the nine different model sizes. An asterisk (*) indicates that at least one solver did not terminate within the time limit for two or more of the nine runs, and so an accurate mean cannot be computed.

can only be calculated when both algorithms terminate within the given time limit of two hours. When at least one algorithm does not terminate for a given number of TA, that run is excluded from the average. While this usually does not significantly impact the average, property live-two with bound thirty, as well as live-six with bounds of twenty-five and thirty have three or more runs excluded from the average, and as a result we have omitted these averages from the chart. This is because there is a clear trend, at least across the first five properties, where the ta2smt algorithm clearly scales better to handle both more agents and larger bounds. This means that when multiple runs with large numbers of agents are excluded, as is the case in the three examples mentioned, the average is heavily weighted towards runs with fewer agents, favoring ae2sbvzot. The most striking example of this trend is Live-two, where with two agents, both algorithms' performances are roughly equivalent, whereas with ten agents the ta2smt algorithm quickly becomes over an order of magnitude faster than the ae2sbvzot encoding.

The averages shown above are an average of the performance over the nine model sizes, and we believe it is in fact a rather pessimistic assessment of the ta2smt algorithm. Because ta2smt exhibits superior scaling to larger models and bounds, if larger models and higher bounds were to be added to the results, we expect that the results would be even more favorable to ta2smt. An interesting exception to this trend seems to be the bound of thirty, where ta2smt exhibits a lower speedup than with a bound of twenty-five. After consulting the tables we highly suspect that this is due to a quirk of ae2sbvzot, where the results with a bound of thirty are slightly better than with a bound of twenty-five for several problems. We are not sure as to the cause

of this strange phenomenon, but we do not expect this to repeat at higher bounds.

The one property that does not fit the general trend is Live-six, where the ta2smt algorithm's performance ranges from equivalent to worse than the ae2sbvzot algorithm. This property is unique in that the MITL property to be verified grows in size with the number of TA in the network. It is possible that at larger sizes, the MITL encoding becomes a bottleneck that limits the utility of further TA optimizations. It is also possibile that due to the additional complexity of live-six, higher bounds are needed to offset the possible higher initial costs of ta2smt and reap the benefits of ta2smt's superior scaling. We believe that further tests with a longer timeout are needed to understand these results.

Another result that may seem unusual is the behavior seen in live-three and live-five with a bound of ten. Both algorithms are unable to find a counterexample for networks with seven or more processes. This is in fact an artifact of the strong transition liveness. Because each TA must transition during the loop section of the trace, they must all move to state b from the initial state a. While all processes can make this transition from state a to state b simultaneously, the invariant in state b requires a transition to state c, and unlike the previous transition, each TA must transition individually, as the transition from b to c assigns the process id to the variable id. Thus with a large number of processes and a small bound, it is impossible for every process to complete this path from b to c and then back to b, and thus no valid lasso-shaped traces (with strong transition liveness) exist. Although not shown in the table, with a bound of eleven there exists a valid trace for a seven-process network, with a bound of twelve there exists a valid trace for eight processes, and so on.

4.2 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.

Table 4.3 contains the results of the token ring tests. Although the time taken is not as large as the Fischer tests, we can still see a clear pattern emerge. We used this test as an opportunity to extend the maximum time bound used, and as shown above the general trend of ta2smt scaling more efficiently than ae2sbvzot continues past a bound of thirty. Although the initial

		TACK ae2sbvzot								
		n								
	k	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
live-token	**10**	1.0✓	1.2✓	1.3✓	1.4✓	1.5✓	1.8✓	1.9✓	2.1✓	2.2✓
	15	1.6✓	1.5✓	1.6✓	1.8✓	1.9✓	1.9✓	2.3✓	2.5✓	2.9✓
	20	2.8✓	2.5✓	2.4✓	2.0✓	2.5✓	2.7✓	2.9✓	3.0✓	3.5✓
	25	3.3✓	4.7✓	4.5✓	3.4✓	3.5✓	3.3✓	3.9✓	4.2✓	4.4✓
	30	5.0✓	7.0✓	7.4✓	8.1✓	7.2✓	5.3✓	4.1✓	5.9✓	6.1✓
	35	6.0✓	9.0✓	11.2✓	10.0✓	17.3✓	10.2✓	9.4✓	5.7✓	6.5✓
	40	9.9✓	10.5✓	14.2✓	15.8✓	18.8✓	12.1✓	11.7✓	12.7✓	12.2✓
		TACK ta2smt								
		n								
	k	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
live-token	**10**	0.9✓	1.0✓	1.0✓	1.0✓	1.2✓	1.2✓	1.3✓	1.4✓	1.5✓
	15	1.0✓	1.0✓	1.1✓	1.2✓	1.4✓	1.4✓	1.5✓	1.7✓	1.7✓
	20	1.0✓	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✓	2.0✓	2.4✓	2.6✓	3.0✓
	35	1.3✓	1.4✓	1.6✓	2.1✓	2.2✓	2.1✓	3.0✓	3.0✓	3.2✓
	40	1.4✓	1.5✓	2.1✓	2.6✓	2.4✓	2.4✓	2.9✓	3.4✓	4.4✓

Table 4.3: Time required to check the property of the Token Ring. A ✓ indicates the property is satisfied, a ✗ indicates that a counterexample was found. Blank entries indicate no result after two hours.

results with a bound of ten are practically equivalent, by the time we reach a bound of forty ta2smt starts to perform at least three times faster than ae2sbvzot.

4.3 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 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. 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.

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

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.

The version of the CSMA/CD benchmark presented here is slightly different from the benchmark that has been used in benchmarks of Uppaal and other TA solvers. We noticed that the common benchmark lacked the ability for processes to correctly recover from a collision. In essence, once a collision occurs, one process begins transmitting, while the other must wait in the retry state until the transmission is complete. However a full transmission takes 808 time units to complete, and the retry state has an invariant requiring that processes in the state transition after at most 52 seconds. The only way for a process to remain in the retry state longer is if another collision occurs, which of course also forces the transmitting process to restart. Because there were no other options for the process stuck in the retry state, the transmitting process could never finish a transmission. We modified the benchmark by adding a new transition whose source and destination state are the retry state. It can only be taken if the bus process sends a 'busy' signal (one-to-one), which the bus processes is only allowed to do if a process has been transmitting for over 26 units of time. This transition resets the clock that would require the process to leave the retry state, thus allowing it to remain in the retry state for the full 808 time units required for a successful transition. Although our benchmark is different from those previously used, and therefore our results cannot be compared with the CSMA/CD benchmark for Uppaal, we have created a more accurate model of a real-world transmission protocol, and thus a more useful tool for assessing how our algorithm would perform with a real-world problem.

Figure 4.2 shows the TA diagrams used for the bus process. We have shown in bold the new transition that was added to the standard version to improve the quality of the benchmark. To simulate the ability of the sending processes to detect when the bus is already in use, we have added a new transition to the bus process that allows it to send a **busy** signal to another process when it is in the active state and at least 26 time units have passed since the beginning of the current transmission (26 is used to represent the delay for the signal to propagate to other processes).

Figure 4.3 shows the diagram for the sending process. Again the new additions are shown in bold. Note that each sending process has its own unique clock, so that each process can reset its clock independently. In the diagram for the sending processes we have added three new transitions. The first keeps the TA in the waiting state, and synchronizes using the **cd** signal (short for "collision detected"). This allows the process to choose whether to remain in the waiting state or move to the retry state when it receives this broadcast synchronization

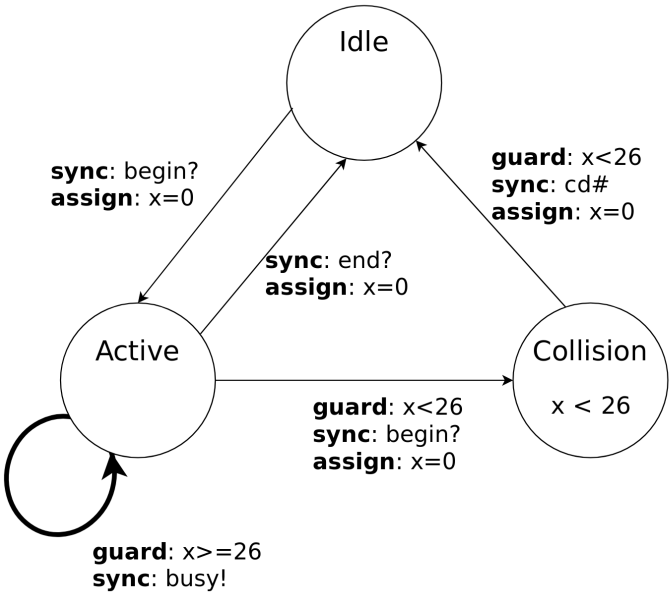


Figure 4.2: CSMA Bus Automaton

signal. The other two new transitions concern the added **busy** synchronization channel. The first allows a waiting process that wishes to transmit to detect a transmission in-progress and transition directly into the retry state. The final, and most crucial transition allows a process in the retry state to detect that the bus is still busy, and reset its clock. This allows it to remain in the retry state for the full 808 time units needed for a complete transmission.

While running this experiment we discovered a previously unknown flaw in the ae2sbvzot encoding of the broadcast synchronization constraints. Therefore the results are not directly comparable, as ta2smt must explore the entire search space before returning a result of **sat**, while the ae2sbvzot algorithm can terminate after finding one (erroneous) counterexample. Rather than present incomparable results, we have decided to instead present two runs of the ta2smt encoding with different liveness and edge constraints.

In Table 4.4 we see the first run, which follows our convention of using Strong Transition Liveness and Right-Closed edges. The second run, however uses Unrestricted Liveness and Open edges. This means that the second run has many more possibilities (valid traces) to consider, and we would expect it to take a longer amount of time to verify the property. Once again for small problem sizes the difference is minute, however although not as drastic as the difference between ae2sbvzot and ta2smt, there is still a significant difference at higher bounds and higher numbers of agents. For the problem sizes we have tested, the unrestricted runs can take up to twice as long as the runs restricted by strong transition liveness and right-closed edges. This is useful information for users weighing the benefits of a broader search against the extra time costs.

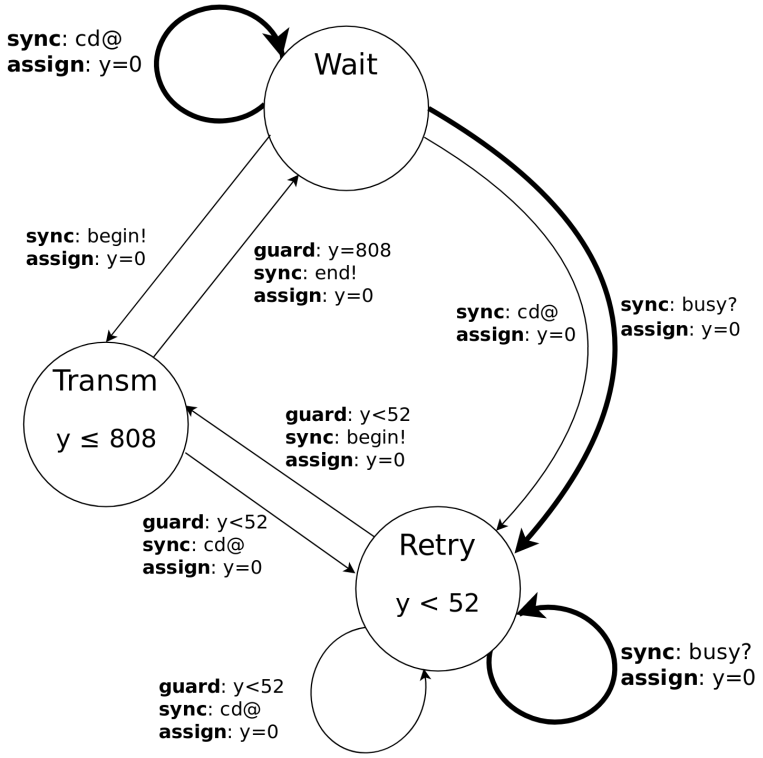


Figure 4.3: CSMA Sender Automaton

TACK ta2smt (Strong Transition Liveness, Right-Closed Edges)										
		n								
	k	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
live-csmacd	**10**	2.4✓	3.5✓	4.3✓	5.1✓	5.2✓	5.8✓	8.8✓	7.8✓	9.5✓
	15	11.0✓	31.2✓	37.2✓	52.7✓	93.6✓	121.7✓	208.7✓	275.4✓	324.0✓
	20	27.1✓	109.5✓	263.5✓	721.5✓	1778.3✓	2206.1✓	3616.7✓	5097.3✓	5068.9✓
	25	82.7✓	813.7✓	2325.2✓	4772.4✓	–	–	–	–	–
	30	296.5✓	2882.0✓	–	–	–	–	–	–	–
TACK ta2smt (Unrestricted Liveness, Open Edges)										
		n								
	k	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
live-csmacd	**10**	2.8✓	3.9✓	5.0✓	5.3✓	8.1✓	9.8✓	9.2✓	11.8✓	10.5✓
	15	14.0✓	44.2✓	55.6✓	72.7✓	198.2✓	219.4✓	209.7✓	469.9✓	367.2✓
	20	64.3✓	245.4✓	543.9✓	1188.7✓	2764.7✓	2037.6✓	4110.6✓	–	–
	25	205.9✓	2362.4✓	3753.4✓	6535.8✓	–	–	–	–	–
	30	559.7✓	5921.7✓	–	–	–	–	–	–	–

Table 4.4: Time required to check the property of the CSMA/CD Protocol. A ✓ indicates the property is satisfied, a ✗ indicates that a counterexample was found. Blank entries indicate no result after two hours.

Chapter 5

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-LIB language. One of the original motivations for using CLTLoc as an intermediate representation in TACK was to create a common language that was both feasible to translate into SMT form and expressive enough to easily support the encoding of additional features. We have strived to make our encoding modular and easily accessible so that future additions will not be hampered by the "low-level" nature of our encoding.

In addition to re-implementing the translation of timed automata networks, we have corrected several deficiencies 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 run-time, with a wider selection of values than the original CLTLoc encoding. Furthermore our encoding eliminates the unnecessary state BitVectors, creating a smaller search space for the SMT solver. 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 exhibits 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 solve larger and larger bounds grows more slowly when compared with the CLTLoc encoding. In addition, after discovering a flaw in the CLTLoc encoding that rendered it unable to solve the CSMA/CD property, we took the opportunity to compare the performance of our encoding with Unrestricted Liveness and open edges against the same encoding but with Stong Transition Liveness and right-closed edges only. Our results showed that the latter encoding achieved significant speedups by reducing the search space, with the speedup increasing with the problem

size.

5.1 Future Work

One potential future research direction is to similarly implement the encoding of the MITL property files directly into BitVector logic. As seen in the TACK results, the speedup achieved by our tool was weakest when the MITL property was the largest, namely during the tests of Fischer liveness property six. Another benefit to this re-implementation would be the potential inclusion of finite traces (albeit with restrictions on the properties supported). This can be supported in the TA encoding by disabling the loop constraints or making them conditional on a 'loop-exists' variable, however modifications to the MITL encoding would be necessary to make this extension useful.

In addition to future research, there are also opportunities to add additional ease-of-use features to TACK. One such feature that could be added is support for automatically verifying that bounded integer variables are never assigned a value outside of their bounds. Currently our BitVector implementation simply allows variables to overflow, considering the burden of verifying the bounds to fall on the user. However it would be relatively straightforward to increase the size of the variable BitVectors so that all possible illegal assignments are guaranteed to not overflow, and to then create a custom MITL property that asserts that the variable never leaves its given bounds. Users could then check to see if their bounds were in fact correct, and receive a trace showing the violation if one existed.

Bibliography

[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), pp. 359–364, Springer Berlin Heidelberg, 2002.

[4] 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.

[5] 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.

[6] R. Bryant, "Bryant, r.e.: Graph-based algorithms for boolean function manipulation. ieee trans. computers 35(8), 677-691," *Computers, IEEE Transactions on*, vol. C-35, pp. 677 – 691, 09 1986.

[7] 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.

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

[9] R. Alur, T. Feder, and T. A. Henzinger, "The benefits of relaxing punctuality," *J. ACM*, vol. 43, p. 116–146, Jan. 1996.

[10] L. Baresi, M. M. Pourhashem Kallehbasti, and M. Rossi, "Efficient scalable verification of ltl specifications," 05 2015.

[11] L. Baresi, M. M. Pourhashem Kallehbasti, and M. Rossi, "How bit-vector logic can help improve the verification of ltl specifications over infinite domains," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, (New York, NY, USA), p. 1666–1673, Association for Computing Machinery, 2016.

[12] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)." `www.SMT-LIB.org`, 2016.

[13] R. Kindermann, T. Junttila, and I. Niemelä, "Beyond lassos: Complete smt-based bounded model checking for timed automata," in *Formal Techniques for Distributed Systems* (H. Giese and G. Rosu, eds.), (Berlin, Heidelberg), pp. 84–100, Springer Berlin Heidelberg, 2012.

[14] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for the Construction and Analysis of Systems* (W. R. Cleaveland, ed.), (Berlin, Heidelberg), pp. 193–207, Springer Berlin Heidelberg, 1999.

[15] S. Demri and D. D'Souza, "An automata-theoretic approach to constraint ltl," *Information and Computation*, vol. 205, no. 3, pp. 380 – 415, 2007.

[16] 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), pp. 193–209, Springer International Publishing, 2016.