Smt Translation

Robert Smith

August 18, 2020

Contents

1 Introduction

2 Preliminaries

2.1 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 labelled as 0, and the leftmost, most significant bit labelled 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 $\overleftarrow{x}_{[n]}^{[i]}$ to *extract* the ith bit from the BitVector x. It is also possible to extract a sub-vector with the notation $\overleftarrow{x}_{[n]}^{[j:i]}$, 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 bitwise operators not $\neg$, disjuction $\vee$, conjunction $\wedge$, 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.

2.2 Timed Automata

Let AP be a set of atomic propositions, and let Act be a set of synchronization events. 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 contraints, where a clock constraint γ is a relation $x \sim c \mid \neg\gamma \mid \gamma \wedge \gamma$, where $x \in X$, $\sim \in \{<, =\}$, and $c \in \mathbb{N}$. $Assign(Int)$ is a set of variable assignments of the form $y := exp$, where $exp := exp{+}exp \mid exp{-}exp \mid exp{\times}exp \mid exp{\div}exp \mid n \mid c$, $y \in Int$ 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 $\mathcal{A} =< AP, X, Act_\tau, Int, Q, q_0, v^0_{var}, Inv, L, T >$. Q is the finite set of states of the timed automaton, and $q_0 \in Q$ is the initial state. $Inv : Q \to \Gamma(X)$ is a function assigning each state to a (possibly empty) set of clock constraints. The labelling 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 $T \subset Q \times Q \times Act_\tau \times \Gamma(X) \times \Gamma(Int) \times \mathcal{P}(X) \times \mathcal{P}(Assign(Int))$, consisting of a source and destination state, an action, a set of clock and variable guards, and a set of clocks to be reset when the transition fires. 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 $\mathcal{A} = [\mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_k]$. Timed Automata in the same network can refer to common clocks, variables, and syncronization channels to coordinate their actions. To simplify the notation we will use the symbols T, X, Int, and Act/Act_t 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 a specific timed automaton.

2.3 Bounded Satisfiability Checking

(here sketch both ta->CLTLoc encoding and sbvzot translation)

3 TA Encoding

Using BitVector logic, we have the ability to group logically connected propositions into a Vector, granting significant speedups on operations performed on every element of the vector.

When encoding the constraints of the system, it is convienent to write that a constraint will hold over every discrete time position in the trace. As an example, consider a transition with an guard $x_i < 5$. When formalizing the constraints, it would be simpler to have a formula of the type *transition* $\rightarrow$ *constraint* that we can assert over every time position at once. Therefore we will use BitVectors of length $k+2$, where each position in the BitVector represents the formula at a different moment in time. This allows us to use the BitVector implication operator to assert that the transition BitVector implies a given constraint at every time instance.

This encoding, while convienent, is not very efficient. Using one BitVector per each transition yields a space complexity of $O(|T|k)$. Since only one transition is active at a time, it is more compact to store the currently active transition as a binary number over $\lceil \log_2 |T| \rceil$ bits, where T is the set of transitions. Therefore we will create $\lceil \log_2 |T| \rceil$ BitVectors of length $k+2$ to represent the active transition of the TA over time. In order to be able to convienently refer to individual elements of the set, we will define aliases which refer to unique combinations of the BitVectors. This will give us the convience 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 instances as follows:

	k+1, ..., 1, 0
0	$\overleftarrow{sb_{i,0[k+2]}}$
1	$\overleftarrow{sb_{i,1[k+2]}}$
...	...
$\lceil \log_2 n_i \rceil - 1$	$\overleftarrow{sb_{i,\lceil \log_2 n_i \rceil - 1[k+2]}}$

3.1 Transitions

In the traditional description of Timed Automata, a TA that does not perform a discrete transition at a given time instance is said to perform a *null transition*, i.e. staying in the same state without firing any transition in the set T. In our encoding it is convienent to explicitly add a null transition for each state $q \in Q$ to the set of transitions. $\forall_{q \in Q} trans_{null_q} :=< q, q, \tau, \varnothing, \varnothing, \varnothing, \varnothing >$, and $\mathcal{T} = T \cup \{\cup_{q \in Q} trans_{null_q}\}$ $trans_{null} := |_{q \in Q} trans_{null_q}$

We define $O : \mathcal{T} \rightarrow \mathbb{N}$ be a bijective function mapping each transition to a natural number less than $|\mathcal{T}|$. We define BitVectors $\{\overleftarrow{tb_1}, \overleftarrow{tb_2}, \ldots, \overleftarrow{tb_{\lceil \log_2 |\mathcal{T}| \rceil}}\}$ of size $k+2$. The BitVector for each individual transition is defined as $\overleftarrow{trans_{t[k+2]}} := \&_{i=1}^{\lceil \log_2 |\mathcal{T}| \rceil} N_t(tb_i)$, where $N_t(tb_i)$ returns tb_i if the ith bit in the base two representation of $O(t)$ is 1, and returns $\neg tb_i$ otherwise.

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

3.2 States

For each TA $\mathcal{A}_l \in \mathcal{A}$, we define a BitVector to represent each state of the timed automaton. To do this we define each state as the disjunction of all the transitions whose source is that state.

$$state_s := |\{trans_t : source(t) = s\} \quad \forall_{s \in S}$$

For each TA $\mathcal{A}_l \in \mathcal{A}$, let $O : Q \to \mathbb{N}$ be a bijective function mapping each state to a natural number less than $|Q|$. We define BitVectors $\{\overleftarrow{sb_1}, \overleftarrow{sb_2}, \ldots, \overleftarrow{sb_{\lceil \log_2 |Q| \rceil}}\}$, each of length $k+2$. The BitVector for the individual state is then defined as $\overleftarrow{state}_{q[k+2]} := \&_{i=1}^{\lceil \log_2 |Q| \rceil} N_q(sb_i)$, where $N_q(sb_i)$ returns sb_i if the ith bit in the base two representation of $O(q)$ is 1, and returns $\neg sb_i$ otherwise.

3.3 Variables

Bounded integer variables are treated slightly differently, because unlike states and transitions, the possible values of a bounded integer variable are not unrelated objects in a set, but integers that must respect the operations of addition and subtraction. For each variable $v_i \in Int$ we still construct a bit representation $\overleftarrow{vb}_{i,j[k+2]}$, where each BitVector has length $k+2$. However the difference is that the values are encoded in 2s 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 $\lambda(v_i)$ as the number of bits needed.

However sometimes it is more convienent to refer to the complete value of a variable at a particular time instance, rather than a particular bit of the variable over every time instance. We make use of SMT-LIB2's 'extract' and 'concat' operators to define a second set of BitVectors that are defined over the first set. $\overleftarrow{var}_{v,j[\lambda(v_i)]}$, $0 \leq j \leq k+1$ is a vector of $\lambda(v_i)$ bits that represents the value of variable v_i at time instance j.

3.4 Clocks

Each clock $c \in \mathcal{C}$ is represented by a function c 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 instance.

4 Constraints

4.1 Initialization & Progression

Initialization and Progression Constraints

$$\phi_1 := \bigwedge_{i\in[1,|\mathcal{A}|]} \overleftarrow{1}_{[1]} = \overleftarrow{state_{init(i)}}^{[0]} \quad \Big| \quad \phi_2 := \bigwedge_{v\in Int} \overleftarrow{init(v)} = \overleftarrow{v[0]} \quad \Big| \quad \phi_3 := \bigwedge_{c\in C} init(c) = c(0)$$

$$\phi_4 := \bigwedge_{i\in[0,k+1]} \delta(i) > 0 \quad \Big| \quad \phi_5 := \overleftarrow{0}_{[k+2]} = \bigwedge_{i\in[1,|\mathcal{A}|]} \overleftarrow{trans_{null_i}}$$

$$\phi_6 := \bigwedge_{t\in\mathcal{T}} \overleftarrow{trans_t}^{[k:0]} \Rightarrow \overleftarrow{state_{t_-}}^{[k:0]} \;\&\; \overleftarrow{state_{t_+}}^{[k+1:1]}$$

$$\phi_7 := \bigwedge_{c\in C} \bigwedge_{j\in[0,k]} \bigwedge_{t\in\mathcal{R}(c)} (\neg \overleftarrow{t})^{[j]} \Rightarrow c(j+1) = c(j) + \delta(j)$$

$$\phi_8 := \bigwedge_{v\in Int} \bigwedge_{t\in assign(v)} (\neg\overleftarrow{trans_t}^{[k:0]}) \Rightarrow \bigwedge_{j\in[1,\lambda(v)]} (tb_j^{[k:0]} = tb_j^{[k+1:1]})$$

The initialization constraints are similar for states, clocks, and bounded variables. For states, we assert that the initial state holds in the first time instance by comparing the vector for the initial state $state_{init(i)}$ 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 instance 0. For variables, we assert that the provided intial starting value, $init(v)$ is equal to the value of the variable at time instance 0. For clocks, we assert that the clock function at time instance 0 is equal to its provided initial value in formula ϕ_3.

Each time instance in the range $[0, k+1]$ represents an instant of time in which at least one timed automaton makes a discrete (non-null) transition. In between these instances, all timed automata remain stationary, and only the clocks progress. To capture this progression, we introduce a new clock, δ. Formula ϕ_4 captures that δ is defined as a function over integers in the range $[0, k+1]$ that returns positive integers. The value of $delta(i)$ at instance i refers to the amount of time between instance i and instance $i+1$. To ensure that each time instance contains a discrete transition, we assert with formula ϕ_5 that at every instance, at least one timed automaton i has $\overleftarrow{trans_{null_i}}$ set to 0, meaning that it is not taking a null transition. This guarantees that at least one timed automaton has an active non-null transition at each time instance. 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_6 asserts that when a transition is taken at time instance i, the source state of the transition is active at instance i, and the destination state is active at instance $i+1$.

We must next discuss the progession of the clocks and integer variables. In formula ϕ_4 we discussed the special clock δ, and how it represents the passing of time between the discete time instances. Formula ϕ_7 connects δ to the other clocks. At each time instance i, a clock is either reset by a transition, or its value increments by $\delta(i)$. To do this we define the set $\mathcal{R}_c$ for every clock c, which is defined as the set of all transitions t that reset the value of clock c. When no transition in $\mathcal{R}_c$ is active, the clock must progress according to the value of δ. Similarly for variables, we define the set $assign(v)$ for every variable v containing all transitions that assign a value to the variable. When none of these transitions are active, formula ϕ_8 ensures that the value of v remains unchanged.

4.2 Transitions

Transition Constraints

$\models \ := \sigma_c(l) + d \models \Gamma(X) \iff \bigwedge_{c,\sim,val \in \Gamma(X)} c(l) + d \sim val$
$\models \ := \sigma_v(l) \models \Gamma(X) \iff \bigwedge_{v,\sim,val \in \Gamma(X)} \overleftarrow{var_v(l)} \sim \overleftarrow{val}$
$\models_w \ := \sigma_c(l) + d \models_w \Gamma(X) \iff \bigwedge_{c,\sim,val \in \Gamma(X)} c(l) + d \sim_w val$
$\phi_9 := \bigwedge_{t \in T} \bigwedge_{l \in [0,k]} \overleftarrow{trans_t}^{[l]} \Rightarrow \sigma_c(l) + \delta(l) \models t_{\gamma_c}$
$\phi_{10} := \bigwedge_{t \in T} \bigwedge_{l \in [0,k]} \overleftarrow{trans_t}^{[l]} \Rightarrow \sigma_v(l) \models t_{\gamma_v}$
$\phi_{11} := \bigwedge_{t \in T} \bigwedge_{c,val \in t_{a_c}} \bigwedge_{l \in [0,k]} \overleftarrow{trans_t}^{[l]} \Rightarrow c(l{+}1) = val$
$\phi_{12} := \bigwedge_{t \in T} \bigwedge_{v,expr \in t_{a_v}} \bigwedge_{l \in [0,k]} \overleftarrow{trans_t}^{[l]} \Rightarrow \overleftarrow{var(l{+}1)} = \overleftarrow{expr(l{+}1)}$
$\phi_{13} := \bigwedge_{t \in T} \overleftarrow{trans_t}^{[l]} \Rightarrow (\sigma_c(l) \models Inv(t_-) \wedge \sigma_c(l{+}1) \models_w Inv(t_+) \vee (\sigma_c(l) \models_w Int(t_-) \wedge \sigma_c(l{+}1) \models Inv(t_+))$

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 the earlier chapter 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 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 $c \ \sim \ val$, where $c \in X$, $val \in \mathbb{Z}$ and $\sim \in \{<, >, \leq, \geq\}$. Formula ϕ_9 asserts that for every clock guard, its transition being active at time instance 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 instance l, the transition occurs in the instant between time instance l and time instance $l + 1$. Therefore, at the instance of the transition, the clock does not have the value $c(l)$, but rather $c(l) + \delta(l)$, delta being the special clock that defines the amount of time spent in each time instance. Note that we cannot simply use $c(l + 1)$ as the value of the clock, because it is possible that during the transition between time instance l and $l + 1$, the value of the clock may be reset, which would set $c(l + 1) = 0$. Our guard only sees the pre-transition value of the clock, and thus we must manually add $\delta(l)$ to the value. ϕ_{10} captures the same semantics for variable guards, asserting that an active transition with a guard implies that the guard is true at that time instance. Because variables, unlike clocks, do not progress with time, it is sufficient to simply use the value $var(l)$ to determine if the guard is satisfied.

Clock assignments are more straightforward then the clock guards. It is enough to require that if a transition is taken at time instance l, then in the following time instance the clock is reset to the desired value. Variable assignments however, are more complex. Unlike clock assignments, which reset clocks to a constant number in $\mathbb{Z}^+$, 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 bvlogic,

we first require that if a variable v' appears in the assignment expression of variable v, then the possible values of v' must be a subset of the possible values of v. Recall that $\overleftarrow{var_v}(l)$ is a bit vector of $\lambda(v)$ bits that contains the value of v at time instance l in two's-complement form. By constraining $v' \subseteq v$, we prevent v' from having a BitVector of greater length than that of v. We can then cast all constants and variables to BitVectors of length $\lambda(v)$, sign-extending shorter variables if necessary. This allows us to use the standard BitVector addition and substraction operators to compute the final value, which is assigned to v at time instance l+1.

The last component of a 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 a transition never leads to a state whose invariant would be unsatisfied. ϕ_{13} accomplishes this using the notions of strong and weak satisfaction. Explained in the original TACK paper, weak satisfaction is a relation where the invariants are relaxed so that the relations $<, >$ are also satisfied with equality. This was done to model the fact that at the instant of transition, the transition is located in exactly one of the two states, source or destination. This choice can be different for each timed automaton and each time instance. If the timed automaton is not in a state in the instance of transition, then a strict inequality can be satisfied with equality at the instance of transition, since the automaton is not actually in that state at that instant.

4.3 Sync

Sync Constraints

$$\phi_{14} := \bigwedge_{t \in T: t_\epsilon = \alpha!} \overleftarrow{trans_t} \Rightarrow (\neg \bigvee_{t' \in T: t'_\epsilon = \alpha! \wedge t' \neq t} \overleftarrow{trans_{t'}}) \wedge (\bigvee_{t' \in T: t_\epsilon = \alpha?} \overleftarrow{trans_{t'}})$$

$$\phi_{15} := \bigwedge_{t \in T: t_\epsilon = \alpha?} \overleftarrow{trans_t} \Rightarrow (\neg \bigvee_{t' \in T: t_\epsilon = \alpha?' \neq t} \overleftarrow{trans_{t'}}) \wedge (\bigvee_{t' \in T: t_\epsilon = a!} \overleftarrow{trans_{t'}})$$

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

$$\phi_{17} := \bigwedge_{\alpha \in Act} \bigwedge_{\substack{i \in [1,|\mathcal{A}|] \\ l \in [0,k+2]}} (\bigvee_{\substack{t \in T_i: \\ t_\epsilon = \alpha\#}} \overleftarrow{trans_t}^{[l]}) \Rightarrow$$
$$\bigwedge_{j \in [1,k]:} (\bigvee_{\substack{t' \in T_j: \\ t'_\epsilon = \alpha@}} (\overleftarrow{state_{t'_-}}^{[l]} \wedge \sigma_c(l) + \delta(l) \vDash t'_{\gamma_c} \wedge \sigma_v(l) \vDash t'_{\gamma_v}) \Rightarrow \bigvee_{\substack{t' \in T_j: \\ t'_\epsilon = \alpha@}} \overleftarrow{trans_{t'}}^{[l]})$$

$$\phi_{18} := \bigwedge_{t \in T: t_\epsilon = \alpha@} \overleftarrow{trans_t} \Rightarrow (\bigvee_{t' \in T: t_\epsilon = \alpha\#} \overleftarrow{trans_{t'}})$$

Different Timed Automata in our network use the synchronization channels in Act_τ to communicate and coordinate their transitions between states. Each element in Act consists of a channel, which we will represent with greek letters α, β, etc, and an action to be performed over the channel. Our implementation supports four actions which are represented using four punctuation symbols. The first two, $send(!)$ and $receive(?)$, capture one-to-one communication. For every channel α there can be at most one active transition with $\alpha!$, and similarly at most one active transition with $\alpha?$. Informally this means that only one timed automaton can send over the channel at a time, and only one can receive at a time. Furthermore each send must be matched by a receive and vice versa. Formula ϕ_{14} captures these semantics for a transition with action $\alpha!$ for some channel α. Such a transition implies that no other transition with the action $\alpha!$ can be active in the same time instance, and furthermore one of the transitions with the action

$\alpha?$ must be active. Formula ϕ_{15} captures the same constraints from the point of view of the receiving transition. A transition with action $\alpha?$ implies both that no other receiving transition is active, and also that there exists an active sending transition over the same channel.

The second pair of synchronization communication is termed 'broadcast synchronization'. Like the one-to-one communication, there is a broadcast send (#) and a broadcast receive (@). However there are several differences in the semantics of broadcast signals. To begin, while a broadcast receive signal must be matched with a broadcast send, the reverse is not true, and a broadcast send signal can be matched with any number of broadcast receives, including zero. While multiple broadcast receive signals on the channel can be fired at the same time, there can only be one broadcast send signal at a time per channel. The other important distinction is that the broadcast send signal 'compels' the other Timed Automata to respond with broadcast receive if they are able to. By this we mean that when a Timed Automaton fires a transition with a 'broadcast send' event, all other Timed Automata with an 'available' transition containing a 'broadcast receive' signal (on the same communication channel) must take the transition. By 'available' we mean that the Timed automaton is in the source state of the transition and all of the clock and variable guards are satisfied. Formulas ϕ_{16} and ϕ_{18} describe these constraints for broadcast send and receive, respectively, while formula ϕ_{17} describes the 'compulsive' nature of the 'broadcast send' transition.

4.4 Loop Constraints

Loop Constraints

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

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

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

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

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

$$\phi_{24} := \bigwedge_{c\in X} c(k) > c(max) \vee ((\ \bigm|_{t:c\in t_{\gamma_c}} \overleftarrow{trans_t}) \ \&\ \overleftarrow{inloop} \neq \overleftarrow{0}\ )$$

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 declare the variable *loop*, and constrain it to have a value in the range $[1, k]$.

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

bit tb_i contains the same value at time instances $k+1$ and *loop*. Similarly, formula ϕ_{20} enforces the same requirement for each bounded integer variable.

It is tempting to encode the clock constraints in a similar manner, requiring that $c(k+1) = c(loop)$ for each clock. However prior work by Kindermann[ref] has shown that this constraint is not complete, as it excludes valid lasso-shaped runs. To remedy this problem we use the requirements suggested by Kindermann. To begin, for each clock c we define the non-negative integer $max(c)$, which is equal to the maximum value either assigned to the clock in a clock assignment or compared against the clock in a clock guard. We also define $frac(c(l))$, which is equal to the fractional part of c at time instace l, or $frac(c(l)) = c(l) - \lfloor c(l) \rfloor$. Formulas ϕ_{21}, ϕ_{22}, and ϕ_{23} encode the desired requirements. ϕ_{21} encodes the first part of the relationship between $c(loop)$ and $c(k+1)$. It states that either both values are greater than $max(c)$, 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 $max(c)$, either the clock value at both time instances 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.

Unfortuantely, there is one more consideration we must make in this section. The culprit are so-called "Zeno traces", named because while they are lasso-shaped runs with an infinite number of transitions, their execution happens in finite time. Time in these traces is said to "slow down", because often each successive loop of the lasso executes in a smaller amount of time than the loop before. Because these represent unrealistic scenarios, they are often excluded from consideration in many TA models. It is sufficient to require that every clock is either reset within the loop, or has a value greater than $max(c)$ 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.

5 Verification

6 Evaluation

7 Conclusion