Smt Translation

Robert Smith

August 12, 2020

Contents

1 Introduction

Timed Automata are a commonly used representation for modelling the behavior of systems with real-time semantics.

Bounded Satisfiability Checking is a process in which timed automata can be verified against a property. The TA system along with the desired property are converted into a format suitible for parsing by a SAT or SMT solver. Then the solver is tasked with finding a counterexample to the property. Since TA traces are infinite in length, we restrict ourselves to traces of the form $s_1 s_2, \ldots s_{l-1}(s_l s_{l+1} \ldots s_{n-1} s_n)^\omega$. 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 inifinite 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 n. Bounded Satisfiability Checking refers to checking if a given property is satisfied over lasso-shaped traces of up to length n.

Current examples of TA bounded model checkers include XX, YY, and de-facto standard Uppaal. TACK is a tool focused on allowing for the expression of TA properties in Metric Interval Temporal Logic, a rich ...

TACK translates both the TA and the property to be verified into CLTLoc. Constraint Linear Temporal Logic (over clocks) is a variant of

Zot ..

Sbvzot is a very sucessful solver which takes advantage of bit vector logic...

To further improve the performance of TACK, we wish to directly translate the network of timed automata into the SMT-LIB format, skipping the intermediate CLTLoc representation. While CLTLoc is an elegant expressive language, there is a significant overhead in

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 $\overleftarrow{x}^{[i]}_{[n]}$ 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 bitwise operators not $\neg$, disjuction $\vee$, conjunction $\wedge$, equivalence $\iff$.

2.2 Timed Automata

Let AP be a set of atomic propositions, and let Act be a set of 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 $x \sim c | \neg\gamma | \gamma \wedge \gamma$, 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 $exp := exp + exp \,|\, exp - exp \,|\, exp \times exp \,|\, exp \div exp \,|\, n \,|\, c$, $y \in Int$ and $c \in \mathbb{Z}$. $\Gamma(Int)$ is the set of integer variable constraints, where a variable constraint γ is defined as $\gamma := n \sim c | n \sim n' | \neg\gamma | \gamma \wedge \gamma$, 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.

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. 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 encoding, while convienent, is not very efficient. 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:

//#+ATTR$_{\text{LATEX}}$: :caption Representation of n elements over time with $\log_2$ n BitVectors

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

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 *nulltransition*, 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. $\forall_{q \in Q} trans_{null_q} :=< q, q, \tau, \varnothing, \varnothing, \varnothing, \varnothing$ $\mathcal{T} = T \cup \{\cup_{q \in Q} trans_{null_q}\}$ $trans_{null} := |_{q \in Q} trans_{null_q}$

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 $O : Q \rightarrow \mathbb{N}$ 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

TODO: mention that the operators $\vee, \wedge, |, \&, \Rightarrow$ represent bvor, bvand, etc. (in background) - maybe explain how you are exploiting bvlogic to write constraints - quick comment

Initialization and Progression Constraints

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

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

$$\phi_6 := \bigwedge_{i\in[1,|\mathcal{A}|]} \bigwedge_{t\in\mathcal{T}_i} \overleftarrow{trans_t}^{[k:0]} \Rightarrow \overleftarrow{state_{source(j)}}^{[k:0]} \ \& \ \overleftarrow{state_{dest(j)}}^{[k+1:1]}$$

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

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

4.1 Initialization & Progression

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 $state_{i,init}$ 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 $\overleftarrow{trans_{i,null}}$ 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

$\phi_9 := \bigwedge_{t\in T} \bigwedge_{\gamma\in TODO_t} \bigwedge_{l\in[0,k]} \overleftarrow{trans_t}^{[l]} \Rightarrow (c_\gamma(l) + \delta(l)) \sim_\gamma val_\gamma$
$\phi_{10} := \bigwedge_{t\in T} \bigwedge_{\gamma\in TODO_t} \bigwedge_{l\in[0,k]} \overleftarrow{trans_t}^{[l]} \Rightarrow \overleftarrow{var_\gamma(l)} \sim_\gamma \overleftarrow{val_\gamma}$
$\phi_{11} := \bigwedge_{t\in T} \bigwedge_{\alpha\in TODO_t} \bigwedge_{l\in[0,k]} \overleftarrow{trans_t}^{[l]} \Rightarrow c_\alpha(l+1) = val_\alpha$
$\phi_{12} := \bigwedge_{t\in T} \bigwedge_{\alpha\in TODO_t} \bigwedge_{l\in[0,k]} \overleftarrow{trans_t}^{[l]} \Rightarrow \overleftarrow{var_\alpha(l+1)} = expr_\alpha$
$\phi_{13} := \bigwedge_{t\in T} \overleftarrow{trans_t} \Rightarrow (v \vDash Inv(source(t)) \wedge v' \vDash_w Inv(dest(t))) \vee (v \vDash_w Int(source(t)) \wedge v' \vDash Inv(dest(t)))$

Each transition can have multiple guards. The guards consist of two types, clock guards and variable guards. We will consider clock guards first. Clock guards have the form $c \sim val$, where $c \in \mathcal{C}$, $val \in \mathbb{Z}$ 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 (which is incremented by $\delta(l)$ to account for the amount of time spent in the state $source(t)$) and the value val. 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.

$\overleftarrow{trans_{i,j}}^{[l]} \Rightarrow (v \vDash Inv(source_i(j)) \wedge v' \vDash_w Inv(dest_i(j))) \vee (v \vDash_w Inv(source_i(j)) \wedge v' \vDash Inv(dest_i(j)))$

4.2.1 Sync

4.3 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 declare the variable $loop_init$, 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 $loop_init$, 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 $loop_init$. We therefore introduce constraints [], which require that the active state and transition of each timed automata at instance $k+1$ be equal to that at instance $loop_init$. The set of constraints [] enforces the same requirement for each bounded integer variable.

Naively we would enforce the same constraint on every clock, however [reference Kinderman paper]. Therefore for each clock