Smt Translation

Robert Smith

August 10, 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 $x \sim c \mid \neg\gamma \mid \gamma \wedge \gamma$, 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 $t \in \mathcal{T}$ has the form $\mathcal{T} \subset Q \times Q \times Act_\tau \times \Gamma(X) \times \Gamma(Int) \times \mathcal{P}(X) \times \mathcal{P}(Assign(Int))$, 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 state with an invariant $x_i < 5$. When formalizing the constraints, it would be simpler to have a formula of the type $state \rightarrow constraint$ 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 state is active at a time, it is more compact to store the currently active state as a binary number over $\lceil \log_2 |Q| \rceil$ bits, where Q is the set of states. Therefore we will create $\lceil \log_2 |Q| \rceil$ BitVectors of length $k + 2$ to represent the state 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 formallized 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 states, we represent the state 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
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 States

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.

For clarity, let us consider an example TA with $\lceil \log_2 |Q| \rceil = 5$ and a state $q \in Q$ with $O(q) = 5$. The base two representation of 5 is 00101, and therefore $\overleftarrow{state_q}_{[k+2]}$ is equivalent to $(\neg sb_5 \,\&\, \neg sb_4 \,\&\, sb_3 \,\&\, \neg sb_2 \,\&\, sb_1)$.

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

Similarly we define $P : T \rightarrow \mathbb{N}$ be a bijective function mapping each transition to a natural number less than $|T|$. We define BitVectors $\{\overleftarrow{tb_1}, \overleftarrow{tb_2}, \ldots, \overleftarrow{tb_{\lceil \log_2 |T| \rceil}}\}$. The BitVector for each

individual transition is defined as $\overleftarrow{trans}_{t[k+2]} := \&_{i=1}^{\lceil \log_2 |Q| \rceil} N_t(tb_i)$, where $N_t(tb_i)$ returns tb_i if the ith bit in the base two representation of $P(t)$ is 1, and returns $\neg tb_i$ 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

4.1 Initializations

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 $\overleftarrow{1}_{[k+2]}$ as follows:

$\overleftarrow{1}_{[1]} = \overleftarrow{state_{i,init}}^{[0]}$

For clocks, we assert the at time instance 0, the clock is equal to the initial value.

$init(c) = c(0), \forall c \in \mathcal{C}$

For variables, we equate $var_{i,0}$ to the initial value of the variable.

$\overleftarrow{init(v_i)} = \overleftarrow{var_{i,0}}, \forall v_i \in Int$

4.2 Progression

4.2.1 Timed Automata

A simple constraint to ensure progression is the constraint that at each time instance, at least one Timed Automaton must undergo a discrete transition. We assert that at least one of the null transitions must not hold at each time instance.

$\neg\overleftarrow{trans_{1,null}}_{[k+2]} \vee \neg\overleftarrow{trans_{2,null}}_{[k+2]} \vee \ldots \vee \neg\overleftarrow{trans_{|\mathcal{A}|,null}}_{[k+2]} \sim$

In addition, we must ensure that if a transition is fired, the state of the TA has the appropriate values. We assert that if a transition from state a to state b is active at time instance i, then state a is active at time instance i and state b is active at time instance $i+1$.

$\overleftarrow{trans_{i,j}}^{[k:0]} \Rightarrow \overleftarrow{state_{i,source_i(j)}}^{[k:0]} \ \& \ \overleftarrow{state_{i,dest_i(j)}}^{[k+1:1]}$

4.2.2 Clocks

Each clock $c \in \mathcal{C}$ is represented by a function 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. To formalize this, we introduce a new clock delta, which is initialized to 0 and is constrained to only have positive values.

$0 < \delta(i), \forall i \in [0, b]$

We then add a constraint for each clock that informally says iff no transition assigns or resets a clock c_i at time instance j, then the clock value increments by the value of $(delta\ j)$. To formalize this we first compute for each clock a list of all transitions that reset the clock's value, $\mathcal{R}_c = \{\mathcal{R}_{c,1}, \mathcal{R}_{c,2}, \ldots\}$, where each $\mathcal{R}_{c,l}$ represents the BitVector $\overleftarrow{trans_{i,j}}_{[k+2]}$ for that transition. We can then express the desired constraint:

$(\neg\mathcal{R}_{c,1}\ \&\ \neg\mathcal{R}_{c,2}\ldots)^{[j]} \Rightarrow c(j+1) = c(j) + \delta(j), \forall 0 \leq j \leq k+1$

4.3 Transitions

4.3.1 Guards

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 $\sim \in \{<, >, \leq, \geq\}$. These guards neatly translate into:

$\overleftarrow{trans_{i,j}}^{[l]}_{[k+2]} \Rightarrow (c(l) + \delta(l))\ \sim\ val, \forall l \in \mathbb{Z}, 0 \leq l \leq k$

Variable guards are similar to the clock guards, with the exception that the value that the variable is being compared to does not have to be an integer, but can be an expression of variables and integers, related with the operators in the set $\sim$ mentioned above. This requires only minor changes to the format of the constraint. We will use the term *expr* to refer to the expression of variables and integers, where variables are represented using the $var(l)$ syntax and the integers and operators are represented as-is. TODO: mention conversion from BV to int of variables in expr.

$\overleftarrow{trans_{i,j}}^{[l]}_{[k+2]} \Rightarrow \overleftarrow{var_{v,l}}\ \sim\ expr, \forall l \in \mathbb{Z}, 0 \leq l \leq k$

4.3.2 Assignments

There are both clock and variable assignments.

Clock: $\overleftarrow{trans_{i,j}}^{[l]} \Rightarrow c(l+1) = val$

Variable: $\overleftarrow{trans_{i,j}}^{[l]} \Rightarrow \overleftarrow{var_{v,l+1}} = expr$

4.3.3 Invariants

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

4.3.4 Sync

4.4 Loop Constraints

5 Clipboard

The first problem is how to represent the currently active state of a given Timed Automaton at a given position in time. Each TA has a finite set of possible states, $S_i = \{S_{i,0}, S_{i,1}, \ldots S_{i,n_i}\}$. In a purely propositional logic the most straightforward approach would be to assign a proposition to each state, however it is possible to instead use the built-in BitVector arithmetic of SMT-LIB2 to instead represent the current state as BitVector. Rather than n_i propositions, a BitVector

of length $\lceil \log_2 n_i \rceil$ bits can represent the current state. $sb_{i,j}, 0 \leq j < \lceil \log_2 n_i \rceil$ refers to the individual bits that make up the state BitVector for TA $\mathcal{A}_i$.

In addition to multiple state bits being related by a single BitVector, we also wish to capture the relationships between the position of the TA at different time positions. Many TA constraints are expressed over all time instances, for example the guard of a transition expresses the constraint that the transition can only fire if a certain condition holds. This constraint will need to be expressed over every time instance. In order to easily express constraints that must hold over every time instance, we will create a BitVector for every bit of our state representation. However instead of having one vector per time instance, where each bit in the vector represents a different bit of the current state, we will have one vector per state bit, where each bit of the BitVector represents a different time instance.

Notation: There is a network (set) of timed automata, $\mathcal{A}$. Each $\mathcal{A}_i \in \mathcal{A}$ has a set of states $\mathcal{S}_i$ and a set of transitions $\mathcal{T}_i$. The functions $source_i$ and $dest_i$ accept a transition $j \in \mathcal{T}_i$ and return the source and destination state respectively. In addition to the timed automata, the network has a set of clocks X and the accompaning clock evaluation function $v : X \to \mathbb{R}_{\geq 0}$. Similarly there is the set of bounded integer variables Int and the variable evaluation function $v_{var} : Int \to \mathbb{Z}$.

Transitions and bounded variables are represented in a similar way. Let m_i be the number of transitions for $\mathcal{A}_i$, then we define a BitVector for each transition bit, $\overleftarrow{tb_{i,j[k+2]}}$, $0 \leq j < \lceil \log_2 m_i \rceil$. We also define BitVectors, defined over the $\overleftarrow{tb_{i,j}}$ BitVectors, for describing each individual transition, $trans_{i,j}$, $1 \leq j \leq m_i$ which has a length of $k + 2$.