CTL satisfability via tableau
A C++ implementation

Nicola Prezza

April 30, 2015

Outline

1. Introduction
2. Parsing
3. Data structures
4. Initial sets of labels and edges
5. Cull
6. Examples and benchmarks

The problem

CTL satisfability

Given a CTL formula ϕ, tell wether there exists a model satisfying it or not

Approach

Tableau-based: a graph is built having as states sets of ϕ-subformulas (and their negations) and as edges all feasible transitions between states. After an iterative culling process, it is verified if ϕ is still present in the tableau.

References

The implementation follows the description of prof. Mark Reynolds, lectures `La verifica del software: temporal logic` — `CTL Satisfiability via tableau`, university of Udine, April 29 - May 20, 2014

Code

Language

CTL-SAT has been implemented in C++ and compiled/tested under gcc 4.8.2+.

link

The **code** with a `readme` and examples can be found at `https://github.com/nicolaprezza/CTLSAT`

Input: a CTL formula ϕ

Pre-processing

1. Blank spaces are removed
2. Substitutions (for grammar simplification):
   - $E\neg\alpha \Rightarrow \neg A\alpha$, $A\neg\alpha \Rightarrow \neg E\alpha$
   - $T \Rightarrow p \vee \neg p$
   - $p \vee q \Rightarrow \neg(\neg p \wedge \neg q)$
   - $EF\alpha \Rightarrow E(T \cup \alpha)$, $AG\alpha \Rightarrow \neg EF\neg\alpha$, $AF\alpha \Rightarrow A(T \cup \alpha)$, ...

Parser type

The grammar is modified in order to remove left-recursion, so that it can be recognized by a top-down parser.

From the user-level, this is the accepted grammar:

Accepted grammar:

S $\rightarrow$

```
S^S | SvS | S->S | (S) | ATOM | ~S | AXS |
EXS | A~XS | E~XS | A(SUS) | A~(SUS) | E(SUS) |
E~(SUS) | AFS | A~FS | EFS | E~FS | AGS |
A~GS | EGS | E~GS | T
```

Where `ATOM` is any symbol other than

```
^, v, ~, ->, (, ), T, A, E, U, F, X
```

and $\sim$ is the negation ($\neg$),

Normalization

Formulas are normalized:

- Double negations removed
- arguments of $\wedge$ are sorted lexicographically

in this way, some easy-to-check logically equivalent sub-formulas are identified (smaller models).

Formula representation

Efficient representation of sub-formulas:

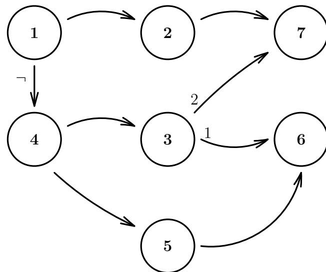
1. *Positive* sub-formulas (i.e. formulas that are not a negation) are sorted lexicographically $\rightarrow$ **positive closure set**: $PC = \{\phi_1, ..., \phi_n\}$
2. The input formula is represented as a *labeled DAG* having as nodes $\{1, ..., n\}$. Edge labels indicate order and polarity of the arguments.
3. Each sub-formula in the closure set is then represented simply as an integer: $\phi_i \leftrightarrow i$, $\neg\phi_i \leftrightarrow -i$
4. Using the DAG, $\mathcal{O}(1)$ access to the sub-formulas of any formula.

Formula representation - example

$$\phi \equiv (AXq) \vee \neg(E(p \cup q) \wedge EXp)$$

Sorted positive closure

1. $\phi_1 = (AXq) \vee \neg(E(p \cup q) \wedge EXp)$
2. $\phi_2 = AXq$
3. $\phi_3 = E(p \cup q)$
4. $\phi_4 = E(p \cup q) \wedge EXp$
5. $\phi_5 = EXp$
6. $\phi_6 = p$
7. $\phi_7 = q$



Sets of formulas

Sets of formulas

Letting n be the positive closure's size, a set of formulas can be represented with a bitvector $B[1, ..., n]$.

Example

$$B = \langle 0, 1, 1, 0, 1, 0, 0 \rangle \equiv$$

$$\{\neg((AXq) \vee \neg(E(p \cup q) \wedge EXp)), AXq, E(p \cup q), \neg(E(p \cup q) \wedge EXp), EXp, \neg p, \neg q\}$$

Advantages

- implicit: only one between ϕ and $\neg\phi$ is in each set
- $\mathcal{O}(1)$ memberships, insert, set copy

Computing the initial set of states

solution 1 - inefficient

generate all 2^n possible sets (bitvectors) and then discard infeasible sets

Adopted solution - efficient

generate on-the-fly only the feasible sets using a tree of choices:

- n levels (one per formula in the positive closure)
- In level i, branch and copy current state k times, one for each set of formulas consistent with ϕ_i or $\neg\phi_i$. Merge set with the copy.
- Always check consistency of a new node. If node is not consistent, then it is discarded $\Rightarrow$ the whole subtree rooted in it is never processed.

Computing the initial set of states

Branching example

Level i, $\phi_i = A(\alpha \cup \beta)$. Each state S at level i is copied (branched) 5 times $S = S_1 = S_2 = ... = S_5$, and the copies are updated as follows:

- $S_1 \leftarrow S_1 \cup \{A(\alpha \cup \beta), \alpha, \beta\}$
- $S_2 \leftarrow S_2 \cup \{A(\alpha \cup \beta), \neg\alpha, \beta\}$
- $S_3 \leftarrow S_3 \cup \{A(\alpha \cup \beta), \alpha, \neg\beta\}$
- $S_4 \leftarrow S_4 \cup \{\neg A(\alpha \cup \beta), \neg\alpha, \neg\beta\}$
- $S_5 \leftarrow S_5 \cup \{\neg A(\alpha \cup \beta), \alpha, \neg\beta\}$

Inconsistent branches are removed at this step. Other formulas processed analogously, as described in the slides.

Computing initial set of edges

- For each pair $\langle B, D \rangle$ check conditions as described in the slides. Add only edges that satisfy all conditions.
- Memorize edges as adjacency lists
- Also reversed edges are memorized in order to speed up cull (backtracking)

Cull heuristics

Cull rules are divided in two categories:

1. **Easy** (check only successors): EXT, EX, ENX
2. **Hard** (visit all the graph): EU, ENU, AU, ANU

idea: remove as many states as possible very quickly (easy rules), so that there are less states left for the hard rules.

heuristic

1. For each node, check easy conditions. When a node fails, backtrack recursively using reverse edges. Repeat until all nodes satisfy easy conditions.
2. For each node, check hard conditions.
   - If found node failing a hard condition: remove it, then jump back to step 1.
   - If all nodes satisfy also hard conditions: STOP.

EXT, EX, ENX implementation

EXT, EX, ENX

Just check if there are successors (EXT) or if there exists one successor satisfying the condition (EX, ENX).

EU, ENU, AU, ANU

Heuristic

We apply the following heuristic: we associate to each state s a set $SAT(s)$, containing all EU, ENU, AU, ANU formulas that we know to be satisfied in s. At the beginning, $SAT(s)$ is empty for each state. While checking validity of formulas, we update $SAT(s)$ by inserting verified formulas.

reason

If we are checking the validity of α in state s and $\alpha \in SAT(s)$, then we are done: no need to re-compute everything.

drawback

Now removing a state s is more difficult: we need to backtrack on its predecessors $s_1, ..., s_k$, remove elements from $SAT(s_1), ..., SAT(s_k)$ that are no more valid and repeat recursively on their predecessors.

EU rule implementation

check $\phi = E(\alpha \cup \beta)$

Find a path $s_0, ..., s_k$ from the current state s_0 such that α is in all $s_0, ..., s_{k-1}$ and β is in s_k

implementation

The solution is a DFS visit. Initially, all states are marked as `NOT_VISITED`:

1. If $\beta \in s_0$ or $\phi \in SAT(s_0)$, then mark s_0 as `SAT` and return.
2. Else if $\alpha \notin s_0$, then mark s_0 as `UNSAT` and return.
3. Else ($\alpha \in s_0 \wedge \beta \notin s_0$):
   - Mark s_0 as `VISITED`
   - If at least one successor v is such that $\phi \in SAT(v)$, then mark s_0 as `SAT`, insert ϕ in $SAT(s_0)$ and return.
   - Else, call recursively on each `NOT_VISITED` successor
   - If at least one successor is labeled `SAT`, then mark s_0 as `SAT`, insert ϕ in $SAT(s_0)$ and return.
   - else, mark s_0 as `UNSAT` and return.

ENU rule implementation

check $\phi = E\neg(\alpha \cup \beta)$

Find a path $s_0, ..., s_k$ from the current state s_0 such that $\neg\beta$ is in all $s_0, ..., s_{k-1}$ and either (i) $\neg\alpha$ and $\neg\beta$ are in s_k or (ii) the sequence loops (i.e. $s_k = s_i,\ 0 \leq i < k$)

implementation

DFS visit. Initially, all states are marked as `NOT_VISITED`.

1. If $\phi \in SAT(s_0)$ or $(\neg\beta \in s_0 \wedge \neg\alpha \in s_0)$, then mark s_0 as `NOT_VISITED` and return with *success*.
2. Else if $\beta \in s_0$, then mark s_0 as `NOT_VISITED` and return with *fail*.
3. Else $(\neg\beta \in s_0 \wedge \alpha \in s_0)$ mark s_0 as `VISITED`. If a successor v is marked as `VISITED` or is such that $\phi \in SAT(v)$, then insert ϕ in $SAT(s_0)$, mark s_0 as `NOT_VISITED` and return with *success*.
4. Else for each successor v, call recursively on v. If the call terminates with *success* then insert ϕ in $SAT(s_0)$, mark s_0 as `NOT_VISITED` and return with *success*.
5. If all successors returned with *fail*, then mark s_0 as `NOT_VISITED` and return with *fail*.

AU rule implementation

check $\phi = A(\alpha \cup \beta)$

Find a sub-structure such that:

- The structure is a DAG with one source in s_0 and many sinks $f_1, \ldots, f_k$.
- The sinks contain β.
- All paths from s_0 have α in their internal nodes.
- All *EX*, *ENX*, *EU*, *ENU* rules are locally (on the edges) verified on the non-sink nodes.

AU rule implementation

implementation

Initially, all states are marked as NOT_VISITED.

1. If $\phi \in SAT(s_0)$ or $\beta \in s_0$, then insert ϕ in $SAT(s_0)$, mark s_0 as SAT and return.
2. Else if $\alpha \notin s_0$, then mark s_0 as UNSAT and return.
3. Else $(\alpha \in s_0 \wedge \beta \notin s_0)$ mark s_0 as VISITED and call recursively on each NOT_VISITED successor.
4. If no successor v is marked as SAT, then mark s_0 as UNSAT and return.
5. Else, for each E rule in s_0 check that *at least one SAT successor* satisfies the condition.
6. If the previous step terminates with success, then insert ϕ in $SAT(s_0)$, mark s_0 as SAT and return, otherwise mark s_0 as UNSAT and return.

ANU rule implementation

check $\phi = A\neg(\alpha \cup \beta)$

Find a sub-structure (loops allowed) with source in s_0 and zero o more sinks $f_1, ..., f_k$ such that:

- All nodes contain $\neg\beta$
- The sinks contain $\neg\alpha$ and $\neg\beta$.
- All paths from s_0:
  - either have α in their internal nodes until a sink is reached
  - OR have α in their internal nodes and they loop.
- All *EX*, *ENX*, *EU*, *ENU* rules are locally (on the edges) verified on the non-sink nodes.

Initially, all states are marked as `NOT_VISITED`.

ANU rule implementation

implementation

use timestamps (DFS visit time):

1. If $\phi \in SAT(s_0)$ or $\neg\alpha \in s_0$, then insert ϕ in $SAT(s_0)$, mark s_0 as `SAT` and return.
2. Else ($\alpha \in s_0 \wedge \beta \notin s_0$) mark s_0 as `VISITED` and set s_0's timestamp* to current time. Call recursively on each `NOT_VISITED` successor.
3. For each E rule in s_0 check that *at least one* `SAT` *or* `VISITED` *successor* satisfies the condition.
4. If the previous step terminates with success, then insert ϕ in $SAT(s_0)$, mark s_0 as `SAT` and return
5. Otherwise, for each state v having a timestamp greater than that of s_0:
   1. Mark v as `NOT_VISITED`
   2. Remove ϕ from $SAT(v)$

   Then, mark s_0 as `UNSAT` and return.

* We need timestamps because when we accept a `VISITED` successor s_j (i.e. a loop), we assume that $A\neg(\alpha \cup \beta)$ is true in s_j, even if we didn't finish visiting its successors. With this strategy, we are sure that if s_j will turn out to not satisfy $A\neg(\alpha \cup \beta)$, we will erase all choices made after its visit. Equivalently, only loops going back to states that eventually will satisfy $A\neg(\alpha \cup \beta)$ will remain in the sub-structure.

Some Examples

"If all paths lead to some point where $A\neg(\alpha \cup \beta)$ is true, then for each path (from current state) finally we will have either $\neg\alpha \wedge \neg\beta$ or a α-loop"

$$\phi = AFA\neg(\alpha \cup \beta) \rightarrow AF\left((\neg\alpha \wedge \neg\beta) \vee EG\alpha\right)$$

results

$\neg\phi$ is not satisfiable, hence ϕ is valid.

Time (s)	0.01
Memory (MB)	1.72
Closure set size	22
Initial number of states	40
Initial number of edges	880

Some Examples

"If from every reachable point there exists a path that leads to a α-loop AND there exists a reachable point from where α always implies β, then a β-loop is reachable"

$$\phi = (AGEFEG\alpha \land EFAG(\alpha \to \beta)) \to EFEG\beta$$

results

$\neg\phi$ is not satisfiable, hence ϕ is valid.

Time (s)	0.11
Memory (MB)	2.69
Closure set size	28
Initial number of states	220
Initial number of edges	10516

Some Examples

"If every occurrence of α leads through some path to a β and vice-versa AND in the current state α holds, then there exists a path from the current state in which I will see (alternated) 5 times β and 4 times α"

$$\phi = (AG((\alpha \rightarrow EF\beta) \wedge (\beta \rightarrow EF\alpha)) \wedge \alpha) \rightarrow$$

$$EF(\beta \wedge EF(\alpha \wedge EF(\beta \wedge EF(\alpha \wedge EF(\beta \wedge EF(\alpha \wedge EF(\beta \wedge EF(\alpha \wedge EF\beta))))))))$$

results

$\neg\phi$ is not satisfiable, hence ϕ is valid.

Time (s)	16.2
Memory (MB)	113
Closure set size	56
Initial number of states	5018
Initial number of edges	1169432