Search strategies for resolution in CTL-type logics: extension and complexity.

Artie Basukoski
Alexander Bolotov

Harrow School of Computer Science

Search Strategies for Resolution in CTL-type Logics: Extension and Complexity.

Artie Basukoski and Alexander Bolotov
Harrow School of Computer Science,
University of Westminster, HA1 3TP, UK
{A.Bolotov,A.Basukoski}@wmin.ac.uk

Abstract

A clausal resolution approach originally developed for the branching logic CTL has recently been extended to the logics ECTL and ECTL$^+$. In the application of the resolution rules searching for a loop is essential. In this paper we define a Depth-First technique to complement the existing Breadth-First Search and provide the complexity analysis of the developed methods. Additionally, it contains a correction in our previous presentation of loops.

1. Introduction

In [1, 7] a clausal resolution method was developed for the basic logic CTL with the two main stages of the method, translation to the normal form, called Separated Normal Form for CTL, SNF$_{\rm CTL}$, and resolution rules. It has been shown that SNF$_{\rm CTL}$ can serve as a normal form for more expressive logics, ECTL ([2, 3]) and ECTL$^+$ ([5]), and the corresponding procedures for translating ECTL and ECTL$^+$ formulae into SNF$_{\rm CTL}$ were defined. This enables us to apply the resolution method defined over a set of SNF$_{\rm CTL}$ clauses as a verification procedure for specifications written in languages of these CTL-type branching-time logics. The core procedure for the application of the resolution method is the discovery of loops. Formally loops are defined as follows:

Definition 1 (Loop in SNF$_{\rm CTL}$**)** *A loop in l is a set of merged clauses (possibly labelled) of the form*

$B_0 \Rightarrow \mathbf{P}_0 \bigcirc C_{0_{\langle \mathsf{ind_0} \rangle}}, \ldots, B_n \Rightarrow \mathbf{P}_n \bigcirc C_{n_{\langle \mathsf{ind_n} \rangle}}$

where **P** *is any of path quantifiers and the following conditions hold* $\models C_i \Rightarrow l$ *and* $\models C_i \Rightarrow \bigvee_{j=0}^{n} B_j$*, for all* $0 \leq i \leq n$*.*

We will abbreviate a loop introduced in Definintion 1 by $(B_0 \vee \ldots \vee B_n) \Rightarrow \mathbf{P} \bigcirc \mathbf{P} \Box l_{\langle \mathsf{ind} \rangle}$, where

(i) if for all i, $(0 \leq i \leq n)$, $\mathbf{P}_i$ is the '**A**' path quantifier then $\mathbf{P} = \mathbf{A}$, $\langle \mathsf{ind} \rangle$ is empty, and we have an **A**-loop in l,

(ii) if for all i $(0 \leq i \leq n)$, $\mathbf{P}_i$ there is only one '**E**' quantifier or every $\mathbf{P}_i$ is the '**E**' quantifier with the same label $\langle \mathsf{ind_i} \rangle$ then $\langle \mathsf{ind} \rangle = \langle \mathsf{LC}(\mathsf{ind_i}) \rangle$ and we have an **E**-loop in l on the path $\langle \mathsf{LC}(\mathsf{ind_i}) \rangle$, otherwise

(iii) we have indicated a hidden **E**-loop in l on an infinite path, $\langle \mathsf{ind} \rangle$, combined from $\langle \mathsf{ind_1} \rangle \ldots \langle \mathsf{ind_n} \rangle$. [1]

2 Depth First Loop search algorithms

We define a *self loop* in l as a loop of the form $B_i \Rightarrow \mathbf{P} \bigcirc (l \wedge B_i)$ for some i. A *partial loop* is given as $B_i \Rightarrow \mathbf{P} \bigcirc (l \wedge (B_i \vee Y_1 \vee \ldots \vee Y_n))$, for some n, and for each $Y_i (0 \leq i \leq n)$, Y_i is a conjunction of literals. A partial loop becomes a loop once we have established that each Y_i is also part of a loop in l. Finally, a *leading loop* in l is a sequence of m clauses of the form $B_i \Rightarrow \mathbf{P} \bigcirc (B_{i+1} \wedge l)_{<ind_s>}$ for $0 \leq i < m$, and for m, $B_m \Rightarrow \mathbf{P} \bigcirc \mathbf{P} \Box l$.

The depth first search method we propose is an adaptation of the analogous method for PLTL by Dixon in [8]. As with the PLTL algorithm we construct a search graph in which edges represent SNFm clauses [6] and the nodes represent the left hand side of these clauses. However the set of SNFm clauses that are applicable is dependent on whether we are searching an **A**-loop or an **E**-loop. Nodes are added to the graph depth first if they satisfy the expansion criteria for either backward or forward search in order to find a subgraph where one of the nodes recurs. Backtracking is used if a particular path leads to a "dead-end". The rules governing expansion guarantee that the desired looping occurs.

Graphs in the algorithm are represented as nested lists

[1] Note that while working on this paper we have found a technical problem in our past presentations of the resolution method for CTL-type logics. Previously [5] we gave the wrong impression that a path on which a hidden loop has been found is a limit closure of some $\langle \mathsf{ind} \rangle$ while now we note that it is a limit closure of a combination of existing indices $\langle \mathsf{ind_1} \rangle \ldots \langle \mathsf{ind_n} \rangle$. However, the only way in which a hidden loop in l can be used in the resolution method is in combination with the $\mathbf{A} \Diamond \neg l$ clause, i.e. with the TRES 3 rule. However, the resolvent of TRES 3 is itself an $\mathbf{A} \mathcal{W}$ clause and, therefore, has no associated indices. This allows us to avoid unnecessary complication of the language of indices, formally preserving our old notation.

Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)
1530-1311/05 $20.00 © 2005 **IEEE**

in which successive entries represent the next node in the graph and where each additional nesting of a level indicates branching, e,g. $[n_0, n_1, [n_2, n_0], [n_4]]$ represents the two paths $[n_0, n_1, n_2, n_0]$ (which is a loop) and $[n_0, n_1, n_4]$. As each entry in this graph is guaranteed to also imply l in the next moment of time (by the expansion rules), this example represents a partial loop in l. It becomes loop in l if we successfully expand to another loop in l from n_4

Depth-First E-Search algorithm. Let R be a set of $\mathrm{SNF}_{\mathrm{CTL}}$ clauses.[2]

1. Search for all clauses of the form $B_k \Rightarrow \mathbf{A}\bigcirc l$, or $B_k \Rightarrow \mathbf{E}\bigcirc l_{<ind_n>}$, for $k = 0 \ldots b$, disjoin the left hand sides, set $toExpand = \{B_1 \ldots B_k\}$. If for any B we find $\neg B$ then terminate returning $\{true\}$. Otherwise use sets of literals in $toExpand$ as the start nodes.

2. Set the current node n_0 to the next element in $toExpand$ if one is available, and $path = [(n_0)]$. If no such node is available terminate - no new loops are found. Now, perform a backward search from n_0 (step 3) and mark this point as a continuation point for a possible forced forward search. When we return from the continuation - if we have not yet found a loop with n_0 as a disjunct we force a forward search on n_0 (to step 6). Otherwise goto step 7.

3. Perform a backward search from n_0 until either
   (a) no loop was detected from any of the successors to n_0 - goto continuation from step 2.
   (b) a self-contained loop was found - goto continuation from step 2.
   (c) a partial-loop was found, i.e. we have used a clause that has two or more disjuncts on its right hand side - remove any nodes that do not form part of the partial loop (the prefix to the loop), store the disjunct list obtained from this clause after removing the disjunct already used in the partial loop and continue processing with step 4.

4. Set n_0, the current node, equal to a new disjunct from the disjunct list if one is available and goto step 5. Otherwise we have detected a loop and return to the continuation from step 2.

5. Perform a forward search from n_0 until either
   (a) no loop was detected from any of the successors to n_0, backtrack to where the disjunctive clause was used (step 3, 5 or 6) and continue processing.
   (b) a self-contained loop was found - goto step 2 continuation.
   (c) a partial loop was found - goto step 4.

6. Perform a forced forward search on n_0 until either
   (a) no loop was detected from the successors to n_0, return to continuation Step 2.
   (b) a self contained loop was found - goto step 7.
   (c) a partial loop was found - goto step 4.

7. Remove any nodes from the path that do not form part of the loop, called the *prefix to the loop*. Extract the set of nodes from the path constructed (noting indices where appropriate) and add them to $loopsFound$ variable. Start the next search from step 2.

Backwards Search Algorithm During the backwards search we seek clauses in SNFm whose right hand side contains a conjunct with l, and also implies the current node.

1. Given the current node n_i, expand the next node n_{i+1} in the search tree by looking for clauses or combinations of clauses of the form $\bigwedge_{a=0}^{g} k_a \Rightarrow \mathbf{A}\bigcirc(\bigvee_{b=0}^{r} C_b \wedge l)$ or $\bigwedge_{a=0}^{g} k_a \Rightarrow \mathbf{E}\bigcirc(\bigvee_{b=0}^{r} C_b \wedge l)_{<ind_s>}$,
   where $\vdash C_b \Rightarrow n_i$.

2. If such a clause exists
   (a) set the current node n_{i+1} to be $\bigwedge_{a=0}^{g} k_a$ and label $_{<ind_s>}$ if looking for an **E** loop;
   (b) if $r > 1$ structure the search path to represent this and store the disjuncts that have not been matched to the current node in a list for future processing; and
   (c) goto step 3;
   otherwise, if no such a clause exists
   (a) if $i > 0$ backtrack setting the current node to n_{i-1} and repeat step 1; or
   (b) if $i = 0$ terminate backwards search and return to the main algorithm.

3. (a) if $n_{i+1<ind_s>}$ is already in the search path return to the main algorithm - a loop or partial loop was detected on $_{<ind_s>}$; otherwise
   (b) increment i and continue at step 1.

Forwards Search Algorithm The forward search algorithm is invoked after a partial loop has been detected using Backwards Search but disjuncts remain to be processed. We also force the forward search for each element in the initial expansion list if the backward search has not returned the most general left hand side for a loop condition for this node.[3] The algorithm works by finding clauses in the set SNFm such that the current node implies the left hand side of the next node, and the right hand side of the next node also contains l. Here the expansion criteria for the node n_i are $\bigwedge_{a=0}^{g} k_a \Rightarrow \mathbf{A}\bigcirc(\bigvee_{b=0}^{r} C_b \wedge l)$ or $\bigwedge_{a=0}^{g} k_a \Rightarrow \mathbf{E}\bigcirc(\bigvee_{b=0}^{r} C_b \wedge l)_{<ind_s>}$, where $n_i \Rightarrow \bigwedge_{b=0}^{r} k_a$. Otherwise the algorithm follows the basic principles of the Backwards search algorithm.

[2] In the Depth-First **A**-Search algorithm which we do not describe here we only consider **A** clauses.

[3] By "most general" loop we mean the current node as a disjunct, e.g. if searching for a loop starting with node $n_0 = p$ and we return a loop $p \wedge q \Rightarrow \mathbf{A}\bigcirc\mathbf{A}\,\Box l$, then we force forward search on p to see if we can discover a loop condition with p as a disjunct.

3 Complexity Analysis

Here the upper bounds on the performance of the algorithms are proposed. Let $|SNF|$ stand for the number of these clauses, and Let $|Prop|$ stand for the number of literals in SNF. We will further distinguish the following subsets, $|\mathbf{E}|$, $|\mathbf{A}|$ and $|\mathbf{E}_l|$, $|\mathbf{A}_l|$, meaning the number of **E** and **A** clauses in SNF respectively, and the subsets of these that also imply $\bigcirc l$. Also let $||P_l||$ stand for $|P| - |P_l|$, where P is either of the path quantifiers. We will follow the convention where $T(n) = O(f(n))$ is the upper bound on the time it takes the algorithm to find *all* loops and terminate for the set SNF. The algorithm begins by extracting all the clauses which imply $\bigcirc l$ in SNF. This number is $|A_l|$ for the **A**-search and $|A_l| + |E_{l_{<ind>}}|$ (the number of **E** clauses implying $\bigcirc l$ along a path labelled by $< ind >$) for the **E**-search. Since each such clause is used as the root of a tree this value is a linear multiplier of the complexity for the depth first tree expansion. The tree expansion begins with the backwards search. To conduct a backwards search, from the current node, n_i, we create a new node in the search tree for each $SNFm$ clause that fulfills the expansion criteria, i.e. the RHS of the matching clause implies the current node and also implies $\bigcirc l$. Let $|A_k|$ be the number of clauses which satisfy these criteria. Hence, the maximum branching factor is $2^{|A|} - 2^{||A_l||}$, with an average branching factor of $2^{|A_k|}$ for some $k \leq |A_l|$. We now look at the depth of the search tree. Consider the base case, that of a self loop, $n_i \Rightarrow \bigcirc(n_i \wedge l)$ which is the shortest loop possible. If all the loops are of this kind then the maximum depth of the search tree is 0. Hence to increase the depth of the search tree requires a chain of the form $n_i \Rightarrow \bigcirc(n_{i+1} \wedge l) \Rightarrow \bigcirc(n_{i+2} \wedge l) \ldots$ for unique $n \in Prop$. The maximum such chain possible is $|Prop|$, hence, the maximum depth of the search tree is $|Prop|$, and we have an upper bound for our algorithm of $O((2^{|A|} - 2^{||A_l||})^{|Prop|})$. A similar measure is obtained for a **E** search where we need to consider all the **E** clauses with the *same index* in addition to the **A** clauses. So, instead of $|A|$ we need to consider $|A| + |E_{<Ind>}|$, where $|E_{<Ind>}|$ is the number of clauses with the same label. The algorithm is very sensitive to the proportion of **A** and **E** clauses in the set SNF. The worst case performance arises when all are **A** clauses, in which case it reduces to the PLTL complexity given by $O(2^{|SNF|*|Prop|})$. The best case performance arises when there are no **A** clauses. Here the determining factor becomes the number of clauses with the same label. In this case $|A| + |E_{<Ind>}|$ becomes $(0 + |E_{<Ind>}|)$ and the complexity is $O((2^{|E_{<Ind>}|} - 2^{||E_{<Ind>_l||}})^{|Prop|})$, where if there is only one clause for each index then the complexity is $|E_l| * (2^1 - 2^0)^{|Prop|}$ which is linear in the number of clauses that imply $\bigcirc l$.

4 Conclusions and Future Work

We have extended the Depth-First Loop search algorithm originally developed for PLTL to branching time logic and presented complexity results. We have also highlighted the cases where the algorithm performs better than the PLTL resolution in cases where a choice between the two methods is appropriate. This algorithm significantly enhances our ability to incorporate different resolution strategies for the clausal resolution method developed for a number of branching time logics (CTL, ECTL and ECTL+). These insights will be useful in developing guided searches in order to improve the performance of the clausal resolution method with a view towards implementation in the near future.

In our previous work [4] we have shown that the complexity of the transformation procedure ECTL$^+ \longrightarrow$ SNF$_{\mathrm{CTL}}$ is polynomial in the length of the input ECTL$^+$ formula. This result combined with the complexity of the loop searching method brings us one step further to establishing the overall complexity of the resolution technique for ECTL$^+$.

References

[1] A. Bolotov. *Clausal Resolution for Branching-Time Temporal Logic*. PhD thesis, Department of Computing and Mathematics, The Manchester Metropolitan University, 2000.

[2] A. Bolotov. Clausal resolution for extended computation tree logic ECTL. In *Proceedings of the Time-2003/International Conference on Temporal Logic 2003*, pages 107–117, Cairns, July 2003. IEEE.

[3] A. Bolotov and A. Basukoski. Clausal resolution for extended computation tree logic ECTL. *To be published in the Journal of Applied Logic*. Extended version of [2].

[4] A. Bolotov and A. Basukoski. Clausal resolution for extended computation tree logic ECTL$^+$. Extended version of [5], submitted for a journal piblication.

[5] A. Bolotov and A. Basukoski. Clausal resolution for extended computation tree logic ECTL$^+$. In *Proceedings of the Time-2004*, pages 140–147. IEEE, July 2004.

[6] A. Bolotov and C. Dixon. Resolution for Branching Time Temporal Logics: Applying the Temporal Resolution Rule. In *Proceedings of the 7th International Conference on Temporal Representation and Reasoning (TIME2000)*, pages 163–172, Cape Breton, Nova Scotia, Canada, 2000. IEEE Computer Society.

[7] A. Bolotov and M. Fisher. A Clausal Resolution Method for CTL Branching Time Temporal Logic. In *Proceedings of the 7th International Conference on Temporal Representation and Reasoning (TIME97)*, pages 20–27. IEEE Computer Society, 1997.

[8] C. Dixon. Temporal resolution using a breadth-first search algorithm. *Annals of Mathematics and Artificial Intelligence*, 22:87–115, 1998.