Faster Logic Satisfiability

Optimising Tableau Methods for Computation Tree Logic Satisfiability

Jimmy Thomson
Raj Goré

Motivation

Formally verifying something such as a program, circuit design, or protocol gives an assurance of correctness. In each of these cases, how things change over time is an important concern. In order to consider correctness, some concept of time is required.

Computation Tree Logic is a candidate logic for expressing these temporal properties. The problem of determining whether a given system satisfies a given property (known as Model Checking) has been investigated extensively, but the related problem of determining whether any system could satisfy a property (known as Satisfiability) has only recently been re-examined.

Several methods of determining satisfiability are known, but they can be impractically slow in some cases. My goal was to try to optimise one particular method and see how much it could be improved.

What is Computation Tree Logic?

Computation Tree Logic (CTL) is standard propositional logic, with a concept of infinite, branching, discrete time.

Because CTL is about time, a formula’s truth can depend on the “future”, so truth is in the context of kripke-models (like the one in figure 1). A kripke-model is a collection of states or worlds where some set of propositions are true or false, and also a relation between these states. For CTL this is the “next” relation, and it must be total meaning no dead-ends.

In order to reason about time or these models, CTL includes “Next” and “until”, allowing statements about a path in the model to be made. Because there are multiple paths, these statements are qualified with “all” or “some” path. For example, \( E(U)(q) \) states “There is some path where \( q \) is true at every time until \( k \) is true, and \( k \) is eventually true”. In the model above, this is true in all of the states. However, \( A(U)(q) \) which makes the same statement except about all paths is only true where \( k \) is true, because from all other times it is possible to get “stuck” in the looping node forever, never making \( k \) true. \( AXp \) and \( EXp \) state that \( p \) holds in all next times, or some next time respectively.

The problem examined here is whether any model exists where at some time a given formula is true, and one way this problem is solved is by attempting to make such a model.

What is Tableau?

A Tableau is a tree created by applying rules a formula. These rules attempt to construct a model which satisfies the formula. Because a finite CTL model will have cycles (as it represents infinite time), the tree of tableau nodes must contain loops, or it would be infinite.

A Tableau can be considered an And-Or tree, where at And nodes all successors have to work out, and at Or nodes at least one does. Figure 3 gives an idea of what kind of interaction can arise in a practical situation.

The specific rules vary between Tableaux, but currently there is no Tableau which is optimal. However, despite this they can perform comparably or better than optimal methods in some cases. One advantage of Tableau methods is that they potentially have a better memory footprint if a depth-first approach is used.

Semantic Branching

When attempting to construct a model using Tableau there are points where an or-choice is made. Much of the time the first option will not be enough to determine an answer one way or another, so the second option must be attempted as well.

Given a set of formulae \( ψ \) and a formula \( ϕ \), the Tableau procedure might try making \( ϕ \) true first. If this is inconclusive, then it must try making \( ψ \) true, but it can also make \( ϕ \) false to make sure that this branch is distinct from the first. This may lead to a contradiction that \( ψ \) on its own would not, but the case of \( ψ \land ϕ \) is covered by the first branch meaning that examining it is redundant. This practice of adding the negation of one or-branch to the other is known as “semantic branching”.

Backjumping

Sometimes when constructing a Tableau a combination of formulae force a contradiction, but not immediately. Figure 2 shows an example where the first branch of an or-choice immediately introduces both \( AXp \) and \( AX¬q \), which together will result in a contradiction while not being negations of each other. The Tableau procedure then decomposes the rest of the formula which does not effect this impending contradiction, before finally discovering the contradiction when it moves to the next timestep.

\[
(AXp \land AX¬q) \land (q \land ¬p) \Rightarrow AX¬p
\]

Figure 2: Simple Tableau where backjumping helps

A naive approach is to simply backtrack and try making other choices. However, in this case that can result in exponentially many Tableau nodes if the tree does not contain other contradictions. Instead, by examining the cause of the contradiction it is possible to ignore all those intermediate steps and immediately go back to the first choice, using a process known as backjumping.

This idea works well on propositional logic where all contradictions are local, but in CTL another cause of contradictions is an unfulfilled eventuality. This is where a loop exists which contains a formula \( A(U)(q) \) but never makes \( q \) true. Backjumping can be applied to this situation as well by determining a set of formulae which contribute to the so called “bad loop”. An overestimation of this is the \( AXp \) and \( EXp \) which created the first timestep in the loop.

Caching

Backjumping requires tracking sets of formulae which create a problem. By caching these sets it is possible to preempt unsatisfiability in some cases. If the set of formulae at a given Tableau node has a subset which is known to be unsatisfiable, then the whole set of formulae cannot be satisfiable and so that branch is known to be unsatisfiable. This can help offset the main downside of the Tableau methods by reducing repeated work. A similar idea can be applied to known satisfiable formulae, though these occur much less frequently.

Detecting when a cached formula is a subset of a Tableau node can be expensive, especially if every unsatisfiable formula is cached, so determining when to check and what to cache is an important concern.

Comparison

Unfortunately there is still no magic bullet. In most cases semantic branching gives a reasonable improvement.

Backjumping and caching together can have a considerable improvement in some cases as well.

but a noticeable detriment in others

None of this helps an important case where a formula is almost satisfiable, but not quite. This can create a large graph (like Figure 3) which must be exhaustively explored, which takes a long time. Other methods such as only creating the graph can perform much better in these cases, but not in all cases.

Figure 3: And/Or graph explored for the Alternating Bit Protocol. Dashed lines indicate Or, solid lines indicate And.