asCTL Model Checker

Matrics: 150003282 and 150009974

November 2018

1 Introduction

This practical consisted of the implementation and testing of a asCTL model checker. All requirements of the basic specifications (Main Task) were completed. Some parts were done in such a way as to improve efficiency over a naive implementation.

All asCTL formula in this discussion follow the syntax accepted by the provided parser.

2 Logic and Model Specifications

2.1 Model

2.2 asCTL Formula

2.2.1 Action Sets

Where action sets were provided, an empty list was interpreted as the set of no transitions. If no list was given, any transition was allowed, as there were no restraints on the operation.

Action sets were treated as constraints on which paths to consider, rather than further logical conditions to check. For example, $\forall G_A a$ would return true for a state from which all onwards paths of transitions from A gave states where a was true, and would not be violated if there was a transition from the state that was not included in A. Only the satisfiability of the state formula were verified.

To verify that no transitions other than those in A were possible, the condition $\exists_A F_B \texttt{True}$ would be verified, where B is the set of all actions not in A (or the set of all undesirable actions).

2.2.2 Strong Until

$$a_A \bigcup{}_B b$$

a holds until b. If a transition from B is possible and leads to a state where b holds, the expression is true. If not, transitions from A to states where a is true are made until it is possible. If such a path exists, the expression holds. If a final transition cannot occur, and no transitions from A are possible, the expression is false. A path of A transitions resulting to a cycle also fails to satisfy the expression.

This implements strong until, as the only accepted paths are those that end in a transition from B to a state where b holds, and all other transitions are from A and end in states where a holds.

If B is not specified, the expression can be true before the first transition, if the initial state satisfies b. If B is given, at least one transition must occur.

2.2.3 Eventually

$${}_A F_B a \equiv \texttt{True}_A \bigcup{}_B a$$

Eventually is true if a transition from B ends in a state where a is true, and all other transitions are from B. This is logically equivalent to the given strong until clause.

As with strong until, eventually can be true before the first transition if and only if B is not specified.

2.2.4 Weak Until

$$a_A W_B b$$

Weak until is true for paths where a transition from B leads to a state where b is satisfied, and all transitions until then are from A and end in states satisfying a. It also accepts cyclic paths of transitions from A in which every state satisfied a, and any path leading to a state from which no transition in B gives a state where b is true and no transitions from A are possible.

As with strong until, if B is not specified the expression can be true before the first transition, but if it is, at least one transition is needed.

2.2.5 Always

$$G_A a \equiv a_A W_{[]} \texttt{False}$$

Always holds if all transitions from A lead to states where a is true, and all onwards transitions from A continue to lead to states where a is satisfied. It fails only for paths of transitions from A containing a state the does not satisfy a.

It is equivalent to the given weak until clause.

2.2.6 Next

$$X_A a$$

Next is true for a path if there is a transition in A from the current state to a state where a is true.

2.2.7 Existential and Universal Qualifiers

2.3 Constraints

Constrains narrow down the search space by forbidding certain transitions. In each node of the search tree, the constraint must hold. A child constraint can then be derived that applies to branches from that node.

A child constraint is created for an onwards transition from the current state. If the condition is satisfied or violated by the current state, the child constraint will be $True$ or $False$, as the constraint becomes stable. For example, the constraint p will be satisfied if p is true in the initial state, so would become $True$ in child constraints.

Boolean constants and atomic propositions always become stable in their children, as they either are or are not satisfied in the current state.

An $\exists$ constraint also becomes stable, as it only excludes states from which a certain path does not exist, and is unconcerned with whether or not it is taken.

Recursive constraints (and, or, and not) get the child constraints of their sub expressions, and perform the same operation on them.

$\forall$ constraints do not always become stable. They constrain the search to paths fitting their path formula, but unlike normal formula, do not fail if they find a path that breaches them, so long as it is not taken. The final path that passes or fails a formula must not break the path formula. If it exceeds it (the path formula is fulfilled before the end of the search), the $\forall$ will become $True$. This happens for $\forall Fp$ if p is true at some point along the path. The constraint has been satisfied, so becomes stable. Always constraints on the other hand never become stable before the end of that search path. This generalises to the untils: strong until becomes stable once the right hand side is reached, while weak until may never become stable, as it does not need to reach the right hand side.

The child constraints of a $\forall$ are therefore more complex. If there does not exist a path from the current state that satisfies the path formula, the child constraint is false. If the onwards transition in in the right hand action set and the right hand condition is fulfilled in the next state, the child is $True$. If the transition is in the left action set, the child constraint is the same as the parent. Otherwise, the child is $False$ (there are no left transitions, and all right transitions lead to an unsatisfactory state). This allows constraints to enforce action set limits.

3 Verification Process

A depth-first search (DFS) was used to find a satisfying or unsatisfying path. An asCTL formula was converted to an abstract syntax tree (AST). Each node was a state formula.

A model passed verification if each initial state satisfied the formula. For multiple initial states, each was tested separately, and the model failed if any failed.

A state was verified by passing it to a node of the AST. For simple state formula (without quantifiers), the condition was checked for that state. For logical operators, this required recursive calls to the branches of the tree.

Quantified state formula contained path formula. Path formula were checked by traversing the path until the formula was either satisfied or breached (according to the operators described above).

∀ formula searched for paths where the path formula did not hold, and returned true only if none were found, while ∃ formula searched for a single path that satisfied it.

3.0.1 Constraint Enforcement

Constraints limited how the search tree grows, in particular for quantified formula.

For simple formula nodes, the constraint was checked for the given state. If it did not hold, the node failed, as only states fitting the constraint were acceptable.

For quantified formula, the constraint limited both the transitions that can be taken, and the states to which they can move.

∃ checked the constraint held in the current state, then produced a child constraint for each branch it searched (as described under Constraints).

∀ performed a similar process, except that it ignored states that breached constraints, rather than considering them path formula failures. This prevented a path the breached the constraint from causing the ∀ to fail.

This method prunes the search tree as it grows. An alternative approach could be to produce a tree of all paths satisfying the constraint, the search it for a path meeting the formula. This would involve extra work, as the first tree generated would be unconstrained by the formula, so would perform constrain checks on branches that would breach the formula anyway. This approach narrows down the search space, so improves efficiency.

3.0.2 Trace Generation

Traces were generated from a stack of state transitions. The DFS pushed and poped transitions as it made recursive calls. If it failed, the stack contained all transitions needed to reach the failing state. Otherwise, the stack would be empty.

For simple state formula, the traces contain the transition after which they do not hold. The same is true for ∃ formula (as there is no trace, as a path does not exist). ∀ formula produce a trace of a path for which the checked path formula is not true.

All stack manipulation was performed by quantified formula, as they used the stack for cycle checks. If model checking failed, either the path would be empty (for simple state formula and ∃), in which case the initial transition would be pushed on, or it would contain a trace that breached a ∀.

4 Checker Verification

5 Efficiency Improvement

Efficiency can be improved by narrowing the search space. As discuss, the constraints mechanism does this by limiting tree growth, allowing constraints to provide an improvement in efficiency.