asCTL Model Checker

Matriculation Numbers: 150003282 and 150009974

22 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. Additionally, two extensions were implemented. The logical operator weak until (W) is supported by both the model checker and the formula parser. Query and constraint checking were implemented in a way as to improve efficiency over a naive implementation.

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

An exhaustive test suit was created which can be used to verify the correctness of the implemented checker. Test and coverage can be done as described in the readme file that accompanied the practical specification. Simply execute './gradlew clean build test coverage' to recreate the testing report. Beyond this report, the source code was documented via *Javadoc* comments.

This project (both code and report) were completed as a group project with close collaboration within the team. An individual report section is included at the end.

2 Logic and Model Specifications

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 $\neg\exists_A F_B \texttt{True}$ would be verified, where B is the set of all actions not in A (or the set of all undesirable actions). If an action in B is taken, the exists statement will return true, which will be negated to false. If an action in B is never possible, the eventually statement will return false because it will never get to its right side.

In the case of transitions associated with multiple actions, they were permitted if at least one action occurred in the action set. This was on the assumption that multiple labels represented different names for the same transition (or duplicate transitions between the same pair of states), and so only one had to match.

2.2 Strong Until

$$a_A \bigcup\nolimits_B b$$

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 transition from B cannot occur, and no transitions from A are possible, the expression is false. A path of A transitions resulting in a cycle also fails to satisfy the expression.

This implements strong until, as the only accepted paths are those that end with a transition from B in a state where b holds, and all previous 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.

2.3 Eventually

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

Eventually is true if a transition from B ends in a state where a is true, and all other transitions are from A. This is logically equivalent to the given strong until clause. States after transitions from A are accepted unconditionally, but the right side of the until must be satisfied at some point.

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

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 satisfies a, and it accepts 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.

2.5 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 to a state that does not satisfy a.

It is equivalent to the given weak until clause. No actions allow transitioning to the right side, so only actions in A are taken and a must be satisfied.

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.

2.7 Existential and Universal Quantifiers

$$\exists \phi$$

Exists holds if at least one path, starting from the current state, satisfies the path formula ϕ.

$$\forall \phi$$

For All holds if all paths, starting from the current state, satisfy the path formula ϕ.

In both instances, ϕ is a path formula as defined in the subsections above.

2.8 Constraints

Constraints 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 and applied 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 $False$, thus 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 the found path is taken.

Recursive constraints (and, or, and not) get the child constraints of their sub expressions, and perform the same operation on them. For example, in state with label p, the child constraint of $p \vee q$ becomes $True \vee False$.

$\forall$ 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 constraint path formula. If it exceeds it (the constraint 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.

2.9 Formula & Model Representation

The provided template has been used and expanded. As given, the query and constraint are state formula. By the defined grammar, a state formula is a logical expression or a quantified path formula. A path formula is one of the aforementioned expressions which contains one or two state formula (which may in turn contain a path formula and so forth).

Nesting of all these was implemented through abstract classes. This allows model checking while abstracting from the given formula. A Main class with a main method was provided, so that a user-defined query, constraint and model can be checked. These need to be supplied as command-line arguments.

To further optimize the model, a TransitionTo class was implemented to provide better execution times, along with a Path class, to represent a trace of execution for the model. An instance of TransitionTo knows the State objects at its two ends, which removes the need to iterate through the model when associating a state name with a state object. An instance of Path keeps both a list and a set of TransitionTo objects. This provides constant time *pushing* and *popping* operations, while also providing constant time *contains* operation. The benefit beyond a set is that an object can be stored multiple times in the case of execution cycles.

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).

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

3.1 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.

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

$\forall$ formula performed a similar process, except that it ignored states that breached constraints, rather than considering them path formula failures. This prevented a path that breached the constraint from causing the $\forall$ 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, then search it for a path meeting the query. This would involve extra work, as the first tree generated would be unconstrained by the query, so would perform constrain checks on branches that would breach the formula anyway. This approach narrows down the search space by checking the query and constraint simultaneously, so improves efficiency.

3.2 Trace Generation

Traces were generated from a stack of state transitions. The DFS pushed and popped 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 $\exists$ formula (if a path does not exist, then there is no further trace to return). $\forall$ 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 $\exists$), in which case the initial transition would be pushed on, or it would contain a trace that breached a $\forall$.

4 Checker Verification

4.1 Unit Tests

The modular structure of an AST lends itself well to unit testing. A comprehensive suite of JUnit tests are provided. The exhaustive testing shows that each part of the implemented model checker behaves as expected. Each test class covers methods that in the respective source code class.

4.2 Integration Tests

Integration testing was done to verify that the system can check complex logical expression in large models. The results can be seen in the gradle report under `all/modelChecker/MutexTests`. The tests are for a mutual exclusion model. The Standard Out view shows the results with a constraint that ensures mutual exclusion and weak fairness, under which the formula being checked is true, and without any constraints, for which the model fails, as mutual exclusion is not enforced.

5 Efficiency Improvement

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

At first, a variation of the current mechanism was attempted - a pruning algorithm that would remove transitions from the given model that do not satisfy the constraint. After an inefficient implementation and discussion of corner cases, it was decided to move on to a different approach. The transition removal would take a long time to prune the branches of a complex constraint and it will not consider the query whilst doing so. The current implementation catches cases when a small query is checked quickly even under a complex constraint.

6 Individual Comments