

asCTL Model Checker

160002045, 160003289

November 2019

Contents

1	Introduction	3
1.1	Requirements	3
1.2	Compiling and Running	3
2	asCTL	3
2.1	Path Formulae Operators	3
2.2	State Formulae Operators and Propositions	4
3	Design and Implementation	4
3.1	Formula Evaluation	4
3.1.1	State Formula	5
3.1.2	Path Formula	5
3.2	Constraints	7
4	Testing	7
4.1	Model 1	8
4.1.1	Model	8
4.1.2	Formulae	8

4.1.3	Tests	9
4.1.4	Model 2	9
4.1.5	Model	9
4.1.6	Formulae	10
4.1.7	Tests	10
4.2	Model 3	11
4.2.1	Model	11
4.2.2	Formulae	11
4.2.3	Tests	12
4.3	Mutual Exclusion	13
4.3.1	Model	13
4.3.2	Formulae	13
4.3.3	Tests	14
5	Critical Evaluation	14
6	Extensions	14

1 Introduction

1.1 Requirements

The requirements of this practical are to implement a model checker for the asCTL logic as defined in the specification.

1.2 Compiling and Running

This project uses gradle to build and run. You can build and run this submission by running:

```
./gradlew clean build test coverage
```

2 asCTL

2.1 Path Formulae Operators

On path formula operators if no actions are specified any action is allowed. The operators consider all paths and produce a set of results, 1 for each path.

Each operator has 1 or 2 state formulae as conditions and the same number of sets of actions.

Always The condition must hold on every state on the path and that the transitions between those path contain one of the allowed actions.

Eventually The path must lead to a state where the condition is true. Before the condition is met one of the left actions must have occurred on a previous transition and one of the right actions must occur on the next transition in the path.

Next After 1 transition the condition must be true, this transition must contain one of the allowed actions.

Until Each eventually a state in the class must meet the right condition (by going through a transition that contains a right action) until then each state must meet the left condition and each transition on the path must contain a left action.

2.2 State Formulae Operators and Propositions

There are 2 types of operator in state formulae: state formulae as operands and those with a child path formula as operand. There are also propositions.

Propositions There are atomic and Boolean propositions. Boolean propositions have a set value that the always are. Atomic propositions are true if the state they are being evaluated against contains a label matching the proposition name.

State Formulae Operands The "and", "not" and "or" operators each take state formulae as operands. They combine or invert the result of their operands. "And" is the logical conjunction of the result of its operands. "Or" is the logical disjunction of its operands. "Not" is the negation of its operand.

Path Formula Operand There are 2 state formula operators that take a path formula as operand: "for all" and "there exists". "For all" is true if all possible paths meet the conditions of the path formula. "There exists" if any of the possible paths meet the conditions of the path formula.

3 Design and Implementation

3.1 Formula Evaluation

Each state and path formula class has a `checkFormula(Model model, State currentState)` method that checks if the formula is valid in a given state (or for any paths in the case of a path formula). These methods return `Result` objects that have a holds flag, trace and path. The trace property are used to display a failure trace if a formula doesn't hold. The path property is used to remove paths that don't fit the constraint from the model.

Formulae are evaluated recursively with each formula evaluating itself on the result of the evaluation of any child formulae. In the case of failure the path

and trace properties are built up while returning up the call stack. This allows formula evaluation to happen in a modular way where each operator is independent from the others.

3.1.1 State Formula

There are 3 general types of **StateFormula**: those with no child formulae, those with child state formulae and those with a child path formula.

No Child Formulae The **BoolProp** and **AtomicProp** have no child formulae and act as the base case for the evaluation recursion. The **AtomicProp** formulae evaluate to true if the current state contains the correct label and the **BoolProp** formulae are evaluated based on the internal value property.

State Formulae Children The **And**, **Or** and **Not** formulae each have other state formulae as their children. These formulae evaluate their child formulae by recursively calling their **checkFormula(..)** method and using the relevant logical operator on the result.

Path Formula Child The **ForAll** and **ThereExists** formulae each have a path formula as their child. The **checkFormula(..)** method for path formula differs from the state formula in that it returns a set of results instead of a single result. These state formula call the **checkFomrula(..)** method on their child formula and then return a result based on the values in the result set. In the case of **ForAll** the formula is true if all the elements in the result set hold and in the case of **ThereExists** the formula is true if at least one element of the result set is true.

3.1.2 Path Formula

Path formulae differ from state form state formulae in that they have to consider multiple paths instead of a single state (this is why they return a set of results instead of just one). They all have a single child state formula.

Next The next formula is the simplest path formula. It applies it considers each transition from the current state and applies its child formula to the target state of the transition and compares the transitions actions to its own action set. If the actions and formula both match then a true result is added to the

result set otherwise a false. The method returns a result set with one result per transition from the target state.

Always The always formula has a recursive `checkPath(...)` method. This method recurs through all possible path checking that the state formula holds for each state and that the actions are present on each transition. While traversing a path if the state formula doesn't hold or the actions on a transition don't match then a negative result is added to the result set. As the method recurs it keeps a set of visited states, if a transition will take the traversal into a previously visited state (or there are no transitions out of a state) then a positive result is added to the result set (as the entire path was traversed without the state formula failing or the actions not matching). If a state has been visited for, or there are no transitions out of a state acts as a base case for the recursion.

Eventually The eventually formula has a similar method structure to always in how it uses recursion to traverse the possible paths. The key difference is that instead of checking that the conditions hold at each step and failing if they don't it continues searching until they do hold. At each step the child formula is evaluated to see if it holds, additionally checks are made to see if the left actions have been seen on any previous transition (a list of previous transitions is passed down the call stack) and if the right actions match the last transition. If these conditions don't hold it recurs down through the possible paths. As in always a set of visited states is kept, if the next state has been visited before then recursion stops. However, when the target of a transition has been seen before a look ahead is done as this transition may not have been considered before.

Until Until uses a similar recursive algorithm to the other path formulae. At each stage the right actions and conditions are checked, if they don't hold then the left actions and condition is checked. If the right conditions hold then a successful entry is added to the result set (the until has been satisfied), otherwise if the left conditions match then recursion continues, if they don't a failure result is added to the result set. This guarantees that the left conditions were true until the right conditions were, otherwise there is a failure.

Using recursion to traverse the model allows all possible states to be searched exhaustively and allows the query or constraint to be applied to all possible paths through the model.

3.2 Constraints











To apply a constraint to the model the model checker checks the constraint against the model in a loop. While the constraint fails the last transition in the path is removed. Once this is complete the model is left in a state where all paths through the model that satisfy the condition. Additionally, if a start state no longer has any transitions out of it that means all paths from it are invalid so it as a start state doesn't match the constraint so it is set to not be a start state. Once all of the invalid transitions have been removed states that have no transitions either to or from them and aren't the start state are removed.

An alternate strategy for handling the constraint would be to pass the constraint down through the formula evaluation methods and have the path formulae only consider paths which meet the constraint. However, processing the constraint ahead of time has the advantage that paths that don't meet the constraint will only be considered once (as they are removed when they're found, otherwise they may be found again). Additionally it has the advantage that once constraint has been applied to the model the constrained model could be saved as JSON again. For large models with complex constraints this could save a lot of time if you want to execute multiple queries as the constrained model only has to be computed once instead of every time.

4 Testing

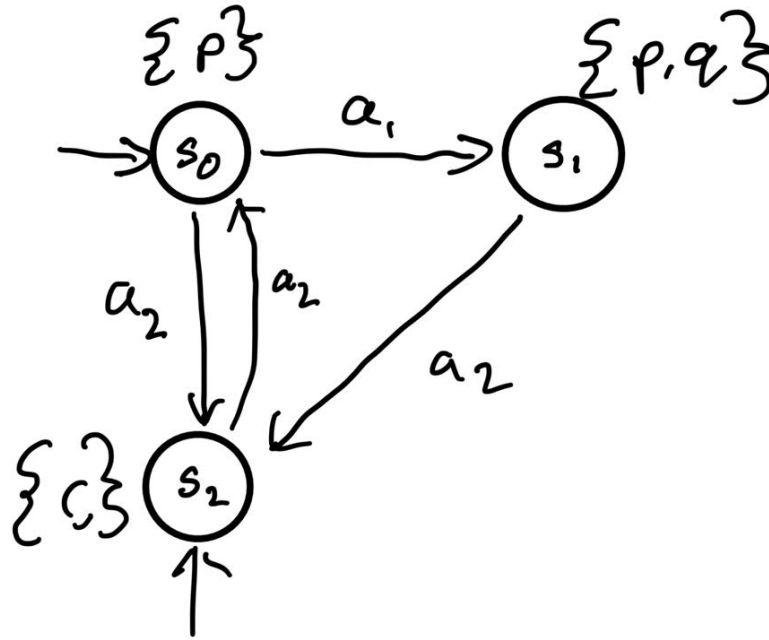
For testing our model checker we used JUnit tests with various models, constraints and queries. We also used the test results page supplied by Gradle to check the output of tests and check that the constrained models were of the correct form by viewing the printed before and after models. We ensured that we tested our entire model checker by using code coverage to check that all areas of code had been reached (Apart from some getters and setters).

Simple

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
formula		81%		79%	15 56	18 127	3 22	0 3
formula.pathFormula		97%		98%	7 67	6 181	6 23	0 5
modelChecker		95%		94%	3 25	2 59	1 6	0 1
model		99%		93%	1 34	1 59	0 26	0 4
formula.stateFormula		100%		100%	0 40	0 112	0 23	0 8
Total	142 of 2,578	94%	16 of 235	93%	26 222	27 538	10 100	0 21

4.1 Model 1

4.1.1 Model



4.1.2 Formulae

Constraint 1 = $\forall_a \Diamond_b (p \wedge q)$

$a = \{a1, a2\}$

$b = \{a3, a4\}$

CTL 1 = $\forall_z \Diamond_b (g \wedge \forall \Box (\forall (True_c \mathcal{U}_d \exists \Diamond (p \vee q))))$

$b = \{a1, a2\}$

$z = \{a3, a4\}$

$c = \{a1, a2\}$

$d = \{a3, a4\}$

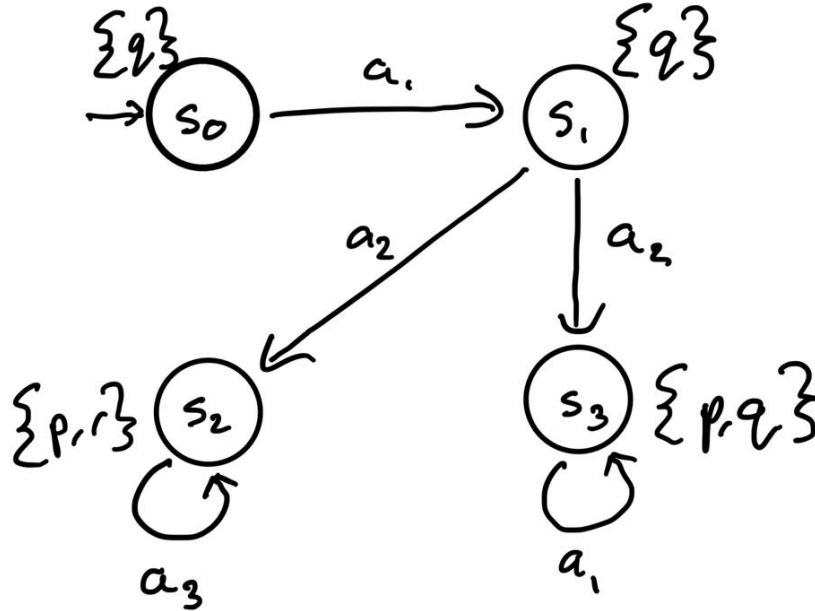
Atomic Prop = $p \vee r$

4.1.3 Tests

For the supplied model 1, we ran two tests querying with the formula CTL 1. The first test was constrained by constraint 1 and the second had no constraint. Constraint 1 will remove all paths from the model since there are no transitions in the model with actions in the set b and so the eventually can never be satisfied. This means the query is on an empty model which returns false and so we assert false for this test. When there is no constraint CTL 1 fails as the eventually requires an action in set z of which there are none in the model so we assert false for this test as well. We then wrote a simple test with no constraint and querying atomic prop to test our model checker when there's no path formula which passed.

4.1.4 Model 2

4.1.5 Model



4.1.6 Formulae

Constraint 2 = $\exists \Diamond_b(r \wedge p)$
 $b = \{a3\}$

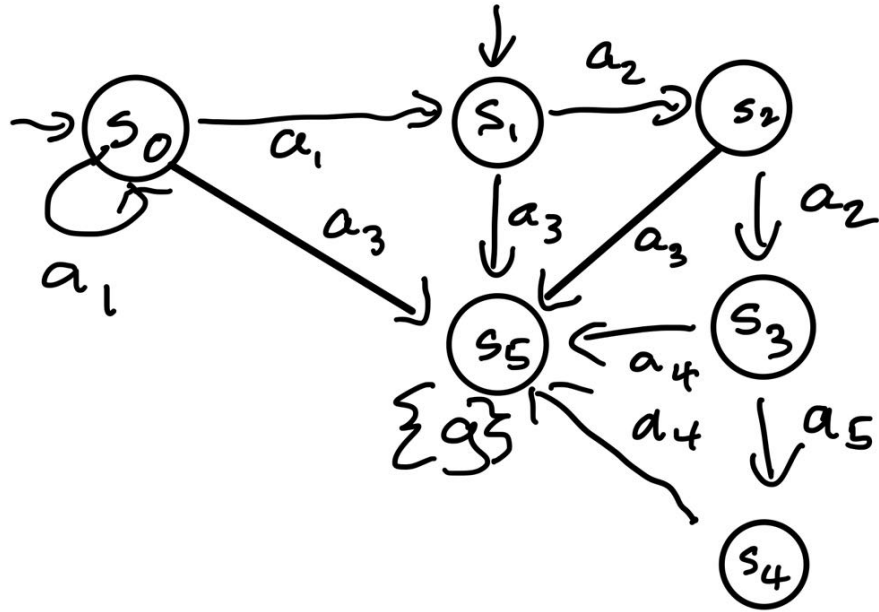
CTL 2 = $\exists(q_a \mathcal{U}_b p)$
 $a = \{a1\}$
 $b = \{a2\}$

4.1.7 Tests

For the supplied model 2, we ran the same two tests as with model 1 but with constraint 2 and CTL 2. Constraint 2 removes nothing from the model as from the initial state, there exists a path that satisfies the constraint (Taking the transition from $s2$ to $s2$ satisfies the eventually). Therefore both tests were testing on the same model and CTL 2 was satisfied as the path $s0$ to $s1$ to $s2$ satisfies the formula.

4.2 Model 3

4.2.1 Model



4.2.2 Formulae

Next Constraint = $\forall X_a g$
 $a = \{a3\}$

Next Fail = $\forall X_a g$
 $a = \{a4\}$

Until Constraint = $\forall (True_a \mathcal{U}_b g)$
 $a = \{a2\}$
 $b = \{a4\}$

Until Fail = $\forall (False_a \mathcal{U}_b g)$
 $a = \{a2\}$
 $b = \{a4\}$

Always Fail = $\forall \Box_a True$
 $a = \{a5\}$

$Or = False \vee True$

Goal Exists = $\exists \Diamond g$

4.2.3 Tests

Model 3 was a model we made to test aspects of our model checker we felt hadn't been tested with other models. This model is designed to allow us to test for paths to a goal state g and we also used it for simple tests to ensure good code coverage.

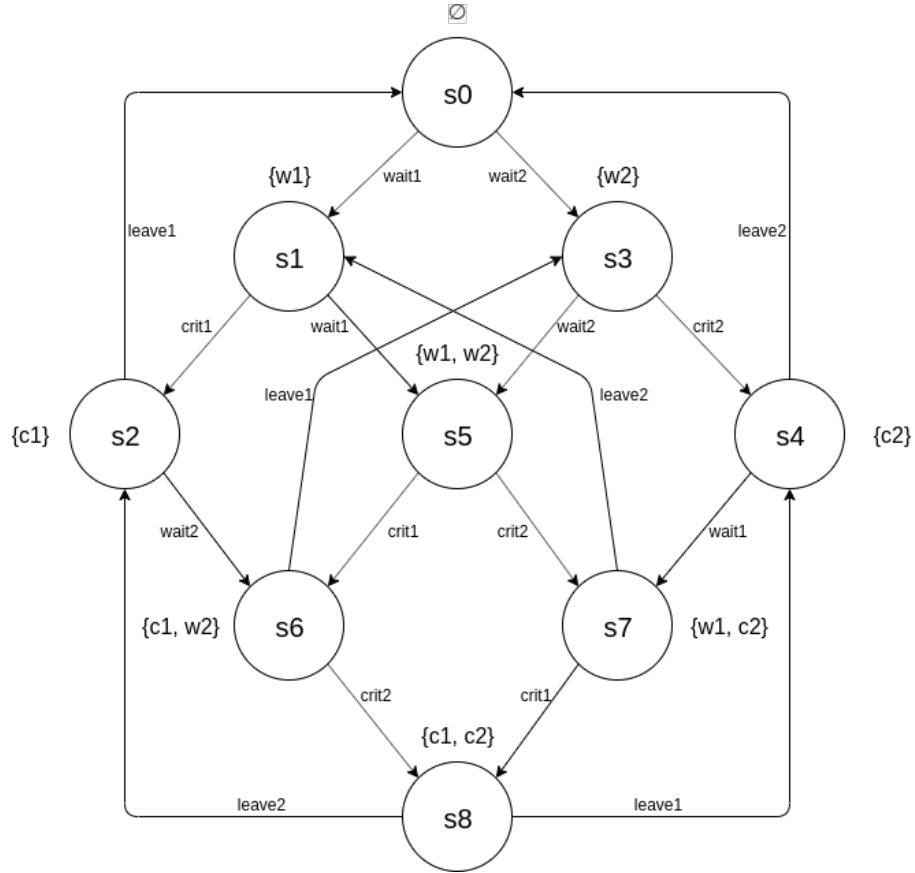
Our first tests on this model were to check that our model checker dealt with path formulas containing next correctly. Using next constraint to constrain our model and goal exists to query it, our test successfully passed and the model had constrained all paths but the paths from $s0$ and $s1$ to $s5$. We also tested that next rejected an incorrect action which we used next fail to test.

Our next tests were to test that actions with until worked correctly. We first checked using an until that required actions from set a until an action from set b was taken to the goal and this passed successfully. We then checked that until failed when the left condition wasn't met and this failed successfully.

To provide good code coverage we then checked that or returned the correct result when only the right hand side was true and this was successful.

4.3 Mutual Exclusion

4.3.1 Model



4.3.2 Formulae

$$\text{Mutex} = \forall \Box \neg (c1 \wedge c2)$$

$$\text{Mutex 2} = \neg \exists \Diamond (c1 \wedge c2)$$

$$\text{Strong Fair} = \forall (((\neg(\forall \Box \forall \Diamond w1)) \vee (\forall \Box \forall \Diamond c1)) \wedge ((\neg(\forall \Box \forall \Diamond w2)) \vee (\forall \Box \forall \Diamond c2)))$$

$$\text{Strong Fair Actions} = \forall \Box \exists_w \Diamond_c \text{True}$$

$$w = \{w1, w2\}$$

$$c = \{c1, c2\}$$

$$\text{Weak Fair} = \forall(((\neg(\forall\Diamond\forall\Box w1)) \vee (\forall\Box\forall\Diamond c1)) \wedge ((\neg(\forall\Diamond\forall\Box w2)) \vee (\forall\Box\forall\Diamond c2)))$$

4.3.3 Tests

This model is a mutual exclusion model which we used to test various properties of mutual exclusion algorithms. We mostly used these tests to check how our models were pruned with various constraints.

Our first test was to run our mutex constraint (guarantees safety) and check what it did to the resulting model. As expected, it removed the transitions from $s6$ and $s7$ to $s8$, ensuring that both processes couldn't enter their critical sections concurrently.

Our next test was to evaluate what strong fairness did to the model. Our strong fairness constraint removed the transitions from $s1$ and $s3$ to $s5$ as this forces the processes to alternate which one is in its critical section by removing the ability for one to get access twice in a row. We then ran a test to check if a model constrained with mutex and querying strong fairness was valid and it was.

We then decided to show equivalence between mutex and a rearranged version mutex 2 by checking that the result was true, no matter which was the constraint or the query. We then did this same check between strong and weak fairness and discovered that they give an equivalent model.

Our final test was to compare our strong fairness formula to an action based version. These were equivalent and it shows the advantage of the addition of actions to CTL as the action based formula was much simpler.

5 Critical Evaluation

This submission meets the requirements of the specification as we have implemented a model checker that checks models in the format described in the specification.

6 Extensions

As an extension we added the ability to convert a model back into JSON. This allows a model to be constrained and then saved in its constrained form (as described in the "Constraints" section). This means that if multiple queries are

to be run on the same model (with the same constraint) then the model can be constrained once, saved and reused multiple times. This saves computation time if the model or constraint is extremely complex.