CS4052 Practical 2 Report

120011995 & 120009909

November 2015

1 Introduction

The aim of this practical was to create a model checker for the *an action and state-based logic (asCTL)*. Our implementation also had to take into account fairness constraints that could be supplied to limit the set of paths utilised for verification. Some components of the program were supplied for us although we had to implement the *SimpleModelChecker* which would conduct the model checking whilst considering supplied fairness constraints.

2 Semantics of asCTL

asCTL has very similar constructs to that of Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) although it allows for verification of statements that these two paradigms do not offer, for example a statement such as, after action α occurs is the system in state where π holds.

It shares most of the semantics of CTL in that it distinguishes between path and state formulae. In our implementation of the Until operator, a formula defined as: X_a U Y_b our model checker will parse this as the actions a must occur until actions b occur and then an optional tautology or atomic proposition will then be evaluated.

Our implementation of the quantifier F (eventually) is such that value (an AP, nested CTL or Tautology) followed by action a should occur until actions b occur.

For the quantifier X (next) we can disregard the first action a, as we assume this was carried out before the model reached its current state. We therefore must evaluate the second action to ensure that from the current state this action occurs next.

The quantifier G (always) means that action b must occur in all instances. Similar to X we have checked that action a has occurred so we can ignore it for this formula.

3 Assumed Logical Equivalences

We were also aware that we could have implemented the quantifiers F and G in terms of the quantifier U. For example we could have created the following logical equivalences using the until operator:

$\exists$F(p) $\equiv$ $\exists$(true U p) and $\forall$G(p) $\equiv$ $\neg$ $\exists$(true U $\neg$ p)
$\exists$G(p) $\equiv$ $\neg$ $\forall$(true U ($\neg$ P) and $\forall$F(p) $\equiv$ $\forall$(true U p)

Whilst this may have involved implementing less quantifiers, we felt that it would also be complex to implement the nestedCTL formulae, as these would require to be further expanded before they could be evaluated by the model checker. This would also add some complexity to the model checking process as the checker would be required to first iterate over the formula to expand these statements before the verification. We therefore decided to implement the F, G and U operators separately.

4 Design and Implementation Decisions

4.1 Formula Prime

Initially, we noticed that there was a large number of String arrays in the *Formula* class which appeared to be verbose and redundant. Thus, we felt more comfortable to move these arrays into Objects of their types. We created the *FormulaPrime* class to contain the new elements of our formula and then created a class for each element of an asCTL formula as follows *AtomicProp*, *Operator*, *Quantifier* and *Tautology*. This allowed us to create much more manageable code by minimising the amount of unnecessary comparison statements that String arrays would require. We also felt that this was more in keeping with Object Oriented style of Java to create an Abstract class of *FormulaElement* which only contains the field of *negation* each of the aforementioned elements extends the Abstract class meaning that also gain this negation property.

4.2 Simple Model Checker

The class *SimpleModelChecker* contains our implementation of an asCTL model checker. The class contains one private field of type ArrayList called *globHistory*. This field is utilised to store and access the paths the model checker has accessed, which is used throughout the program but most notably to return a trace of a path which illustrates that the formula has not been satisfied.

4.3 Check method

The *check* method performs verification of a model based on a formula and constraints. As parameters, it receives a model, a constraint and a formula.

The check method evaluates the path quantifier and then passes in variable *cont* as **True** if the quantifier is $\exists$ or as **False** if the quantifier is $\forall$. It begins this recursive process by first considering the constraint which is used to limit the number of paths the model checker should iterate over, and then runs the model check on this constraint. It deep copies the model at the beginning of each iteration. Any path which throws a not valid exception is then removed from the model. The resulting model is then checked with the copied model, if they are not equal this process is then repeated until the models are equal and thus all invalid paths have been removed.

The *check* method then repeats this process for the Formula. If a NotValidException is caught, it sets the path as the *GlobalHistory* field in the SimpleModelChecker which allows for the getTrace method to function.

In both of these methods the checker first considers the path quantifiers *A* or *E*. If neither of these quantifiers is present one of our custom exception classes will catch the exception and this will be returned to the user.

4.4 Exception Classes

The *OperatorNotSupportedException* was created to catch any misinformed formulae that contain boolean or temporal operators that are not contained in the asCTL grammar. Similarly, the *QuantifierNotFoundException* is utilised to to catch any erroneous quantifiers from appearing in fomrulae.

The *NotValidException* is utilised in the program to return the exact *PointofExecution* object which shows where exactly the formula is flawed. If this exception is thrown, it means that this Formula is invalid because of the path given in the program. The *CycleException* is used to handle any cyclic behaviour within a model and a formula.

4.5 Point of Execution class

To correctly deal with exceptions, the exception classes specified above needed to be given some concept of the current state of the model checker. Thus a construct of where the program was in terms of its execution was created. We called this the *PointOfExecution* class. This class amalgamates the information for a given execution path such as the current state, the previous states, previous transitions and possible future transitions. This allowed us to both effectively debug our model checker, and make errors in the trace be correctly

4.6 Check init states

To begin the model checking process we created the method *checkInitStates*. This method is used to set the current point of execution to the first init state. It then calls the *helper* method to begin the model checking process from this initial state. The boolean variable *cont* is used within this method to represent

continuing through the formula if it is shown unsatisfiable. As in the path quantifier A, the variable *cont* would be set to false, this is because the formula must be satisfiable for all paths within the model. Therefore, if one path is found that violates the satisfiable of the formula, the checker can return that the formula is unsatisfiable as it does not hold in all states. Whereas, if the path quantifier is E the formula must hold in any path therefore you must iterate through all paths to ensure that the formula's satisfiability.

4.7 Check operators

This method is implemented to conduct all comparisons on a per operator basis at the current point of execution. This includes all Boolean operators available in asCTl as well as the temporal operator Until denoted as 'U' which has a different behaviour than the other operators due to fact that it is a Temporal Operator where the other operators are logical operators. This means it requires a concept of previous and next actions.

4.8 Helper

The *helper* method is used to check the state quantifiers of asCTL: F,G,X. The method begins by recursing to the inner most asCTL formula it then checks the state quantifier of this inner most formula. Each of the quantifiers has differing behaviour, with G and X only requiring the action after the quantifier to be checked, with the action before not being checked for the reasons already mentioned above.

4.9 Traverse

The *traverse* method is used to iterate through all of the available transitions of the model from a given state. This is its own method to avoid code duplication in **helper**.

5 Implementation of SimpleModelChecker

Each formula of the logic is either true or false in a given state; its truth is evaluated from the truth of its subformulae in a recursive fashion, until one reaches atomic propositions that are either true or false in a given state. This means that if a formula is not the most nested CTL (a field that is true if none of its sub values are nestedCTL objects), the nested CTL is first evaluated before the overall formula. A formula is satisfied by a system if it is true for all states of the system. As already mentioned, if the formula is invalid at any point, an exception is thrown which correctly saves the stack trace to that given set of execution that creates that error.

5.1 Fairness Constraints

In our implementation fairness constraints are used to prune the branches of our tree of possible paths. First we create the paths of the tree, during checking constraint if we encounter an error an exception is thrown. When this exception is caught, the path passed as the exception parameter is removed from the model. When our model checker begins to check the formula we are iterating over a constrained number of paths, which should provide efficient verification.

6 Testing

6.1 Mutual Exclusion

We decided to follow the advice of the practical specification to create a Model based upon the mutual exclusion example within the lectures. Our model for this can be seen in the file *mutualExclusion.json*. This model contains the states shown in Figure 1. The corresponding set of transitions between these states are given in the file.

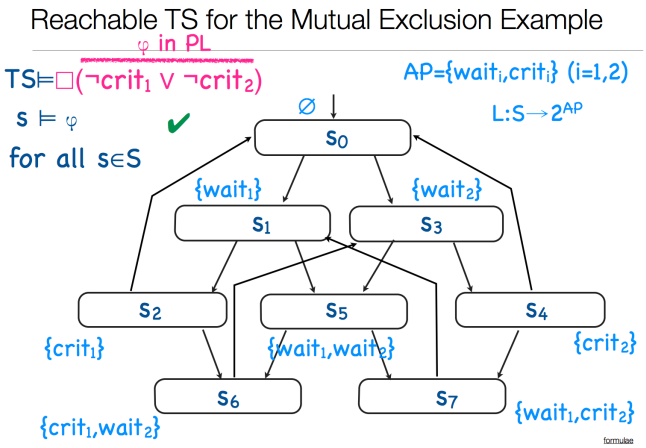

Figure 1: A transition system for the Mutual Exclusion example

6.2 Diagrams of the models used for testing

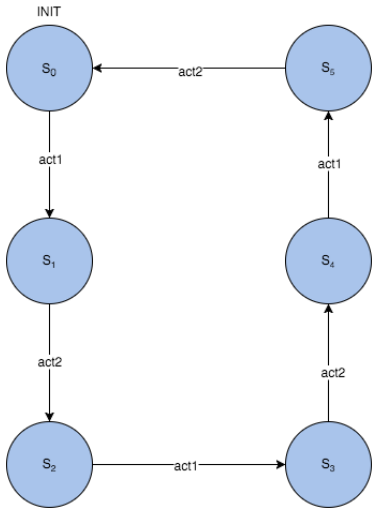

Figure 2: An example transition system for our model 1

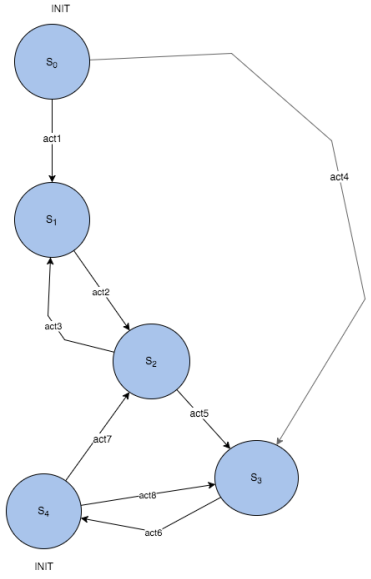

Figure 3: A transition system for our model 2

This mutual exclusion check passes using our model checker, with the formula A(¬ c ∨ ¬ d) being given. If this formula is negated, this test then fails, thus

correctly verifying behaviour of our model checker.

6.3 Testing Methodology

We tested throughout implementation using the Intellij IDE debugging suite. This allowed us to get an indepth understanding of where our problems in our logic were and helped us to solve them. However when we reached the testing stage, we used the JUnit test framework to test different properties of formulae. We split this into several tasks to be comprehensive in testing. These were:

6.3.1 Operators

We attempted to write tests for all of the operators that the checker should support.

These are:

- Single Implication
- Double Implication
- Logical OR
- Temporal Until
- Boolean negation
- Logical AND

7 Extensions

Due to time constraints, not all of these tests were able to be created and/or run. If more time were available, these should be run in future to fully verify the model checker.

7.0.2 Quantifiers

All possible supported quantifiers should also be tested. These are:

- AF
- AG
- AX
- A
- E
- EF

- EG
- EX

A and E were fully tested. G was also tested. However, due to time constraints we were not able to fully test the F constraint.

7.1 Improving the Model Checker

7.1.1 Implementing the Formula Class to Be split into Objects

Using objects of different types instead of just different string arrays allows the Model checker to be both more robust and expandable. This means we can change the behaviour of the model checker relatively easily depending on what type of values we could accept. For example, we could theoretically add support for a formula containing transition information or state name information relatively easily, with very little changes having to be made to the actual evaluation method of the formula.

7.1.2 Adding Javadoc documentation to the code

The implementation has been fully written with Javadoc documentation syntax. This makes documentation consistent throughout the code.

7.2 Optimising the Code to Improve Scalability

7.2.1 Optimisation and Scalability using Exceptions

Using exceptions to handle if a path is not valid allows us to correctly determine how we handle each failed execution attempt of a path. This means we could infinitely scale our model checker depending on the requirements specified.

This is what allows us to use the same method to prune the tree due to the constraint as to evaluate the formula.

This also aids error handling and allows us to correctly interpret what causes an error, whether that is an unspecified quantifier/operator or a cycle occurring where it should not.

7.2.2 Future Optimisation Possibilities

To improve Optimisation in future, the constraint and formula could be validated at the same time, and thus if the constraint fails that branch will no longer be evaluated. This would have the benefit of not having to iterate through whole model to check if any paths should be removed, and if all paths are valid the whole model has to be iterated through again for the actual verification of the formula.

To further improve optimisation, verification of nestedCTL could be threaded for evaluation for the rest of the formula. This would help as there are no dependencies to evaluate each side of the formula. However, to do this in tandem

with the optimisation described above would be difficult. This is due to the fact that there is then a dependency to first evaluate the constraint then evaluate the formula.

8 Difficulties

The largest difficulty we encountered with this practical was the time it took us to understand both the grammar of asCTL and the supplied code. A large proportion of our time for the practical was spent, attempting to create a model checker conforming to this specification although eventually felt it was best to create the *FormulaPrime* class to make our development easier and the final solution more elegant.

We also felt that the more comfortable we became with the asCTL formula grammar, we had to alter our approach to make it more efficient or actually viable. Had we had the in depth knowledge we have now of asCTL and evaluation of the formula, this would have allowed us to implement the model checker in a much more effective and time efficient manner.

9 Conclusion

To conclude this practical was thoroughly worthwhile experience, although our difficulties in evaluating a nestedCTL took a large proportion of our time and effort, without this difficulty we would have liked to more extensively tested our resulting model checker and attempted a large number of the extension activities.