# CS4052 Practical 2 (Report)

160021428
160021429

22/11/2019

# 1 Overview

The main aim of this practical is to implement the satisfaction model checker for the logic asCTL. The asCTL is an action and state-based version of CTL. Unlike CTL, the asCTL considers not only the states but also the actions of the transitions. The model checker we implemented is able to compute the satisfaction model checking for given asCTL queries and constraints. Additionally, a CTL to ENF converter has also been implemented as an extension, which the model checker is also able to parse.

Some components such as file reader and a model builder are provided, thus, we only had to implement the model checker.

For the extension, we implemented the ENFConverter, which converts the asCTL formula to the existentence normal form.

## 1.1 Instruction

Basically, this practical uses the gradle to provide the automation tools for testing and building the project. Thus, by using the gradlew in the root directory, you could easily build and test the project by using the commands below.

- How to build : "./gradlew build"
- How to run the tests : "./gradlew test"

# 2 Design and Implementation

## 2.1 SATChecking

Taking after the design of the CTL model, constraint, and query parser, the SATChecker makes use of a recursive implementation to find all the states that satisfy the provided query and the constraint (if provided). Instead of passing the initial states and building all the paths to the states that satisfy the formula, all states are passed and checked to find every satisfiable state. By doing this we eliminate the need to store all possible paths, reducing the space and time complexity of the application. Given the definition that a $Model \vDash \Phi$ $holds \leftrightarrow I \subseteq Sat(\Phi)$ as discussed in lectures, the model checker is designed to check if the returned set of satisfiable states contains all of the initial states. If any of the initial states are missing, as we know from the provided definition, the model does not satisfy the query given the constraint and will return a path trace to a possible counter example.

The Simple Model Checker is designed to first find all the states that satisfy the constraint before doing the same for the query. Initially, both constraint and query were "Anded" together to find all the states that satisfy both. This has been changed because there is a possibility wherein the model checker does not satisfy the constraint but may satisfy the query. However, given the design of the implementation, it is unknown as to which of the two does not satisfy the query. This is an important aspect of the model checker as it is designed to determine whether a query holds in the model, and if it is the case that the constraint is not satisfied within the model, it is inaccurate and unreliable to assume that the query is also unsatisfied by the model. With the separation of the

two formulas, we can accurately determine the cause of the unsatisfiability and respond appropriately. In the event that the constraint is unsatisfiable by the model, we assume that the model trivially satisfies the formula for the same reasons previously mentioned. It is also because stating that the model does not satisfy the query given the constraint requires a counter example to be provided, where a counter example should only be identified for the query and not the constraint, as we are checking if the model satisfies the query **given** the constraint and not checking if the model satisfies the query **and** constraint. In the event that the model contains some initial states that satisfy the constraint, we proceed to finding the states that satisfy query. If the intersection of both sets (constraint and query) contain the initial states, then the model satisfies the query given the constraint. However, if the intersection does not contain all the initial states then the model does not satisfy the query given the constraint and trace to a counter example will be provided.

When we implemented the model checking methods for ThereExists formula, we found that we actually do not need to implement the methods that handle the path formula for ThereExists and ForAll. Basically, the ForAll formula checks if all given states and paths satisfy the formula, and the ThereExists formula checks if there is any path that satisfies the path formula. Thus, we just implemented getSatThereExists() method and methods that handle the corresponding path formula for ThereExists formula (i.e. getSatExistsUntil(), getSatExistsAlways(), etc). Then, we made the getSatForAll() method to call the getSatThereExists() method, and compare the set that the getSatThereExists() method returns with the set of all given states. Since, the getSatThereExists() will return the set of states that satisfy the formula. As the ForAll expects the set to contain all states, we could just compare the original set of states and the returned states to check if all states satisfy the formula. This is the main reason why we did not need to implement individual methods that handled separate cases.

It is clear that implementing the ExsitsUntil formula checker was the hardest part of this practical. Unlike ExistsAlways and ExistsNext formula, the ExistsUntil has right actions, left actions, left formula and right formula. To handle the left and right formula, we used the getSat() method recursively to get the set of right states and left states. Then, we made a function called getSetOfStatesByCheckingActionConstraints(), which filters the states that does not have the transitions, which have the actions that we are looking for. By using this method, the getSatExistsUntil() could get the set of right states and set of left states, where each set only contains the states that satisfy the corresponding formula and action constraints. Next, we made the getSatExistsUntil() to check if all right states have the transitions, whose source state is one of the left states. If the right state does not have the transition whose source state is one of the left states, then that means that there is no path to that right state that satisfies the given formula. Thus, by doing this we could make the set of right states to only keep the states that we could derive from the left states. By merging the set of left states and set of right states, we could get the set of states that satisfy the ThereExists Until formula.

Additionally, the implementation of the ExistsEventually formula checker is implemented with a variation of the ExistsUntil implementation which is based off of the Minimal set of operators property of CTL. This property states that $EF\phi == E(true\ U\ \phi)$ which is then adapted to take into account for the actions required by asCTL. A similar implementation has been made for the And formulae wherein it can be translated into an Or

StateFormula using principles from CNF.

## 2.2 Finding a Counter Example

Given what is known about the satisfiability of a model, there will always be at least one initial state that is unsatisfied for the model to not satisfy the query given the constraint. Using this logic, we retrieve all the states that do not satisfy the model and build a path from one of the initial states in this set. From the selected state we check if there is a transition to another state in the list of unsatisfied states. This is repeated until a transition to another state in the list is not found or a loop is hit. If the former is the case, we select a random transition from the list of transitions in the current node and add that state to the list, which continues until a terminal state is found. If the latter occurs, we add the repeated state to the end of the trace and break. This shows that the trace has hit a loop and may go on forever.

To find a counter example for a ForAll formula, only a single trace needs to be found that shows that the formula is not satisfied in every state. This proves that the formula does not hold for every state and is simple to implement. However, for a ThereExists formula, a counter example needs to show that **all** states do not satisfy the formula. Printing every trace is difficult and inefficient as the number of possible traces may grow exponentially. But given the definition of ThereExists, it is understood that this formula is only unsatisfiable if there are no states that satisfy the formula from any initial state. Therefore, we can reliably choose any initial state and find a trace to a valid counter example from there.

## 2.3 ENF

While googling to find some additional resources to implement the model checker, we found the logic form called ENF. The ENF stands for Existential Normal Form, which does not have the universal quantifiers but only has the existential quantifiers. All CTLs could be converted to the ENF by replacing the universal quantifiers by the existential quantifiers. Since all CTLs could be converted to the ENF, it is possible to say that we could also convert the asCTL to the ENF.

In the modelChecker directory, you could find the java file called "ENFConverter.java", which contains the ENFConverter class. The aim of ENFConverter is to convert the given asCTL formula to the identical ENF formula. Basically, the ENFConverter checks the type of the state formular by using the "instanceof", and call the corresponding method that handles the specific path formula. By using the recursive method, the ENF converter will convert all asCTL formula to the corresponding asCTL formula with ENF. In the convertToENF() methods in the ENFConverter class, we wrote comments to show which formula will be converted to which ENF formula.

The main benefit that we could get by using the ENF is that we could reduce the number of cases to be covered. This means that by using the ENF, we could simplify the complicate asCTL formula by converting to the normal form. Clearly, this will reduce the number of cases that the model checker should handle. By reducing the number of cases, we could reduce the number of unexpected errors that might happen with the complicate asCTL formula. Henceforth, it is possible to say that we could improve the SAT model checker by using the ENF.

# 3 Testing

To conduct more comprehensive tests on the model we have converted the Mutex model from the lecture slides into the appropriate format to be used by the model checker. This model is used for the Unit tests of each formula and tested against more complex queries and constraints in the Model tests section of the practical.

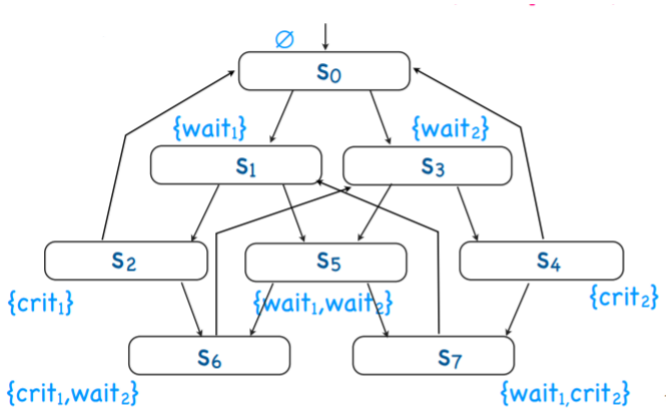



Figure 1: Mutex Model

As can be seen in Figure 2 below, all of the tests pass and respond with the behaviour that was expected from them.

**Test Summary**

| 24 tests | 0 failures | 0 ignored | 0.329s duration | 100% successful |
|---|---|---|---|---|

Packages | Classes

| Package | Tests | Failures | Ignored | Duration | Success rate |
|---|---|---|---|---|---|
| formula | 11 | 0 | 0 | 0.029s | 100% |
| modelChecker | 13 | 0 | 0 | 0.300s | 100% |

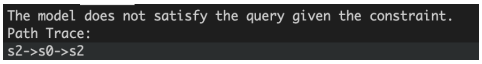
Figure 2: Summary of test Results

Below are the different types of messages that may be outputted depending on the status of the model. Figure 3a is the output for the test, buildAndCheckModel5 which tests if the model can satisfy the constraint *mtxconstraint.json* and query *mtxctl.json*.

Figure 3b on the other hand shows the provided model1 being unable to satisfy the query with a custom constraint that is satisfiable by the model. The custom constraint and query used are *constraint-1-pass.json* and *ctl3.json* from the test buildAndCheckModel3.

The final output shown in figure 3c is a trivially true output from the test buildAndCheckModel2, wherein the constraint is not satisfied by the model and therefore, returns that the formula is trivially satisfied. The constraint and query used by this test are *ctl1.json* and *constraint1.json* which were provided with the initial files.


(a) Satisfies Formula




(b) Counter Example


(c) Trivially Satisfies Formula

Figure 3: Output Messages

The figure below shows the resulting trace from the 3rd Mutex model test which shows the successful detection of a loop in state s5.

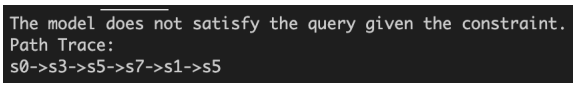



Figure 4: Path Trace (Mutex Model)

# 4 Evaluation

As can be seen from the figures above, the checker passes all 11 unit tests for the formulas and the 13 tests that test the checker with different queries, models, and constraints. And all 24 unit tests are done within 0.4 seconds. Thus, it is possible to say that our model checker is fairly efficient.

By using the ENF converter, we could convert the complex asCTL formula to the normal form, which will reduce the number of possible cases. This will help the model checker to compute the model checking more easily, since the ENF reduces the possible cases.

For the potential extension, it might be possible to improve the trace generator by using the BFS so that the trace generator could build the counter example concurrently. According to this paper, it is possible to trace all paths concurrently by using the BFS rather than using the DFS. Thus, by using the BFS and concurrent tracing algorithm, we might be able to improve the model checker to find the best counter example, or find all possible counter examples. Apparently, we could improve the scalability of the trace generator by tracking all possible paths concurrently.

# 5 Conclusion

To conclude, based off the tests conducted and the findings that have been discovered we were able to successfully develop a simple model checker that takes in the action and state

based logic, asCTL, which can be used to determine whether a given query is satisfiable within the model. Using the provided code as a basis for the structure for our implementation, we were also successfully able to implement an ENF converter as an extension to the implemented SATChecker. In spite of this, the model checker may still be improved further by the aforementioned

# 6 Reference


- ENF: https://disco.ethz.ch/courses/hs10/des/lectures/des_chapter6c.pdf
- Parallel Model-Checking: http://formalverification.cs.utah.edu/papers/pdmc-submission.pdf?fbclid=IwAR0EosDLg7RSOd9A7cJaaQM0zzSOHk5ciYiDjBz7IjaYRnrSGNgwUmaFV84