Model Checker

Hendrik Walter

August 22, 2019

Contents

1 Overview

In this Practical we were asked to implement a model checker for *asCTL*. *asCTL* allows to make statements about the sequence of actions before a particular state is reached. The underlying ideas and theories are explained in 2, followed by the implementation in 3 which is tested against different cases in 4. The additional work beyond the basic model checker is described in 5. A summary is given in 7.

2 Approach

The approach of this submission follows the idea that there is a conversion possible between CTL and ENF. In the following asCTL is covered as a concept prior the explanation of the conversion between asCTL and ENF.

2.1 asCTL explained

Just like *CTL*, *asCTL* is composed form a **path** and **state** formula. Φ represents a state formula which is composed of an atomic proposition $p \in AP$ and/or a path formula ϕ. ϕ itself is composed of Φ and actions $\alpha \in Act$ which can form subsets $A, B \subseteq Act$.

In a more condensed version this is

$$\Phi = true|p| \neq \Phi|\Phi \wedge \Phi|\exists\phi|\forall\phi \tag{1}$$

and similarly for the path formula

$$\phi = \Phi\mathcal{U}_B\Phi|\Phi_A\mathcal{U}_B\Phi|\Phi_A\mathcal{U}\Phi|\Phi\mathcal{U}\Phi \tag{2}$$

The structure above can also be found in [1].

The noteworthy difference to LTL and CTL is that an action α can be checked, allowing not only the existence of the state but also checking how the model transformed into the said state.

An open question arises if one considers the implications of $A, B = \emptyset$ and $A, B = Act$. Considering this example might give an answer to this question. Taking $A = \emptyset$ should be treated the same way as if A was not given, e.g. the A slot was left blank. However, leaving the A slot blank in this formula $\Phi_A\mathcal{U}\Phi$ should then yield $\Phi\mathcal{U}\Phi$. Which is the same if $A = Act$. For this assignment, the behaviour of the model checker is equivalent in both cases.

In addition to the until operator $\mathcal{U}$, the following path operators were specified:

- **Next** with symbol $\mathcal{X}$. Example: $\mathcal{X}_B\Phi$ evaluates to true for a state s_1 if and only if $s_2 = \Phi$ and $\alpha_1 \in B$
- **Always** with symbol $\mathcal{G}$. Example: $\mathcal{G}_B\Phi$ evaluates to true if and only if for all states $s_i = \Phi$ $\alpha \in B$. Note that this can be defined as $True\mathcal{U}\Phi$.
- **Eventually** with symbol $\mathcal{F}$. Example: $\_A\mathcal{F}_B\Phi$ evaluates to true for path π if and only if there is a state $s_f = \Phi$ with $\alpha_f \in B$ and up to that point, all $\alpha_i \in A$ with $i \leq f$. Note that this can be defined as $\neg\mathcal{F}\neg\Phi$, which then can be transformed using the Until operator.

Using the relationships allows to reduce the operators to be checked, simplifying the task of checking for correctness. Satisfaction is described in [2] for a model $\mathcal{M}$ at state s_0 as:

- M, $s_0| = a$ iff $s_0 \in V(a)$ for $a \in A$
- M, $s_0| = \neg\phi$ iff $\mathcal{M}, s_0| \neq \phi$
- M, $s_0| = \Phi \wedge \phi$ iff $\mathcal{M}, s_0| = \phi$ and $\mathcal{M}, s_0| = \Phi$
- M, $s_0| = EX\phi$ iff $\exists$ a fair path $s_0, s_1, ...$ of $\mathcal{S}$ such that $\mathcal{M}, s_1| = \phi$
- M, $s_0| = EG\phi$ iff $\exists$ a fair path $s_0, s_1, ...$ of $\mathcal{S}$ such that for $i \geq 0 \mathcal{M}, s_i| = \phi$
- M, $s_0| = E\phi X\Phi$ iff $\exists$ a fair path $s_0, s_1, ...$ of $\mathcal{S}$ such that there exists $1 \geq 0$ for which $\mathcal{M}, s_i| = \Phi$ and for $0 \leq j < i, \mathcal{M}, s_j, | = \phi$

The algorithm follows these ideas.

2.2 ENF and SAT

AsCTL contains quantifiers which can be expressed through a smaller set of operators. Reducing the number of these operators to simplify model checking is the goal of the transformation into existential normal form (ENF). This is proposed as an idea in [3] and [4]. Primarily, the goal is to eliminate the universal path quantifier $\forall$. According to [1], each CTL formula has an equivalent ENF. The conversion from asCTL to ENF includes the following equivalences:

asCTL	ENF
$\forall X_B \Phi$	$\neg\exists X_B \neg\Phi$
$\forall G_B \Phi$	$\neg\exists(true \mathcal{U}_B \neg\Phi)$
$\exists_A F_B \Phi$	$\exists(true_A \mathcal{U}_B \Phi)$
$\forall(\Phi_A \mathcal{U}_B \phi)$	$\neg\exists(\neg\phi_A \mathcal{U}_B(\neg\Phi \wedge \neg\phi)) \wedge \neg\exists G_B \neg\phi$
$\forall_A F_B \Phi$	$\neg\exists G_B \neg\Phi$

An interesting aspect to point out is the fact that $\forall$ statements are proved showing that there doesn't exist a path where the opposite is true. While most of these are fairly easy to deduce, the 4^{th} formula might require some extra explanation. The asCTL version expresses: For all paths states fulfilling Φ with actions $\alpha \in A$ happen until $\alpha \in B$ happens and ϕ holds. The first statement says that it is impossible to have states not in ϕ connected with actions $\alpha \in A$ until a state neither in Φ nor ϕ is reached with $\alpha \in B$. In other words, with action B a state in Φ or ϕ is reached. Note that the second part says the same as equivalence 5: It is impossible to avoid states in ϕ if $\alpha \in B$, e.g. at least of the states reached with one of the actions must be in ϕ. The latter ensures that we will end up in ϕ with an action $\alpha \in B$, while the former expresses that it is impossible to have states not in ϕ connected with actions $\alpha \in A$ until a state neither in Φ nor ϕ is reached with $\alpha \in B$. Together they express the same as the asCTL version.

Using these relationships allow a neat conversion and simplify the satisfiability (SAT) problem. Where $SAT(\Phi) = \{s \in S | s| = \Phi)$ are the states fulfilling the statement. If initial states $\mathcal{I} \subseteq SAT(\Phi)$ the formula is true.

3 Implementation

The implementation extends the structure provided. The relevant java files can be found in src/main/java and in the directory "ModelChecker" the implementation is provided. Following the approach of first transforming the formula into ENF and then solving the SAT problem, there are files named after their purposes. Further, we used an additional files to define reoccurring constants. For our implementation, we also decided to adjust the formula parser slightly, with each type now having a field describing the nature of the operator. This simplifies casting the different objects. The overall work flow of the program is as follows:

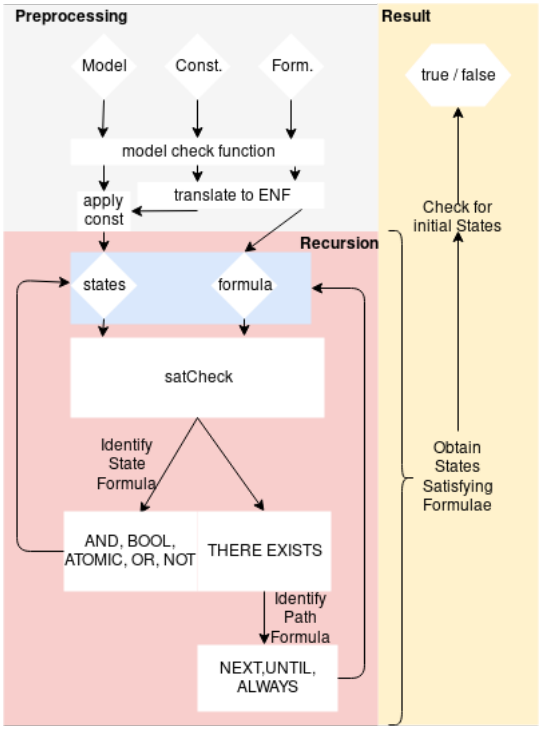

Figure 1: Workflow

From Figure 1 it is possible to see that satCheck is the heart of the implementation. Being called recursively, it is used to identify states fulfilling the formula. With each call the formula gets unwrapped more and more. Taking $!(q\&\&p)$ as an example. This call being allocated to NOT due to !, will trigger a call to satCheck for $(q\&\&p)$, which then is allocated to AND. This ends up being two calls to q and p, which are both Atomic propositions and don't call satCheck itself. From here states are returned and the recursive calls unwind again. Atomic and Bool can be considered the base cases as no more calls to satCheck are triggered from there, but states are returned instead.

These returned states $satCheck(\Phi)$ are compared against the initial states. If and only if the initial states are part of $satCheck(\Phi)$ the formula is accepted on the model. This is due to the fact that the implemented SAT algorithm

"climbs" its way from the leave nodes all the way up to the initial states. If there a state prohibiting access to one of the initial states, this initial state will not be part of the return states. Note that is equivalent to the more formal statement made in the list in the end of 2.1. Hassan puts this as "$\mathcal{M}$ models formula ϕ if all its initial states do in [2].

Important to highlight in this context is the fact, that satCheck is also used to apply the constraint to the model. By using the constraint on the model and using the returned states, which satisfy the constraint, as possible states before checking the formula allows a straightforward way to enforce constraints. Have a look at the models below.

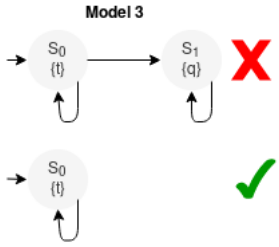

Figure 2: Result of model check without constraint (top) and after constraint was applied (bottom)

Using query $\mathcal{AG}t$, the formula fails. s_1 is a state with $!t$. However, now using constraint $\mathcal{AG}t$ on the model, removes this state and the query $\mathcal{AG}t$ is positive. This is a very basic example how the constraint is applied to the model.

In general the code is strongly modularised, in particular the ENF conversion is not mixed with the SAT Checker and similarly the tokens with formula parser is detached from the other two modules.

The trace is obtained using $ModelStates \setminus SatisfactoryStates$. This will inevitable contain an initial state from which a trace can be build. By reconstruction a path in this fashion, the user can follow where the model fails. An example output for CheckModel2Ctl7 is $-- > s0 - - > s1 - - > s3$. s_0 serves as initial state and the formula fails ultimately in s_3. An indeed, this is the correct deduction of the model checker. Consider the following formula:

```
{
        "formula": "A ((q) aUb (p && r))",
        "a": ["act1","act2"],
        "b": ["act2","act3","act1"]
}
```

The formula expresses that it is always q with actions from a until an action b occurs and then p && r. Now see the model below:

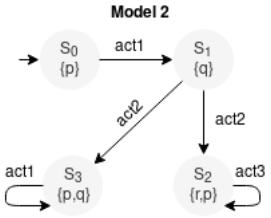

Figure 3: Model 2

Clearly for $s0 -> s1 -> s3$ the machine will end up in state with p && q, hence the initial formula was disproved. The trace functionality is expandable to many problems. However, for some problems, printing the trace is less sensible, which is why we implemented the extension in 5.1 to help with visualization. Assume a problem $\exists$, but this turns out to be wrong. The only way to show the user is to print all traces, which doesn't seem sensible.

4 Testing

Different test cases have been covered in /src/test/java. The relevant resources are in /src/test/resources. The mutual exclusion checker is covered separately in 4.1.

The main testing efforts cover different models and then running expressions on them. Having almost 100% coverage in the SAT solver, the code has been tested thoroughly. This includes variation on the actions, different states and models as well as adding constrains and removing them. Note that there are test cases in which they fail without the constrain, but with the constrain added onto the model, they pass. This shows that the implementation of the checker works fine. Having the test reports and the coverage information, combinations of combinations were covered in simple cases, however, extending these cases is now an iterative process and knowing that the different pieces of the puzzle work as expected, gives confidence that the solution will scale up as expected.

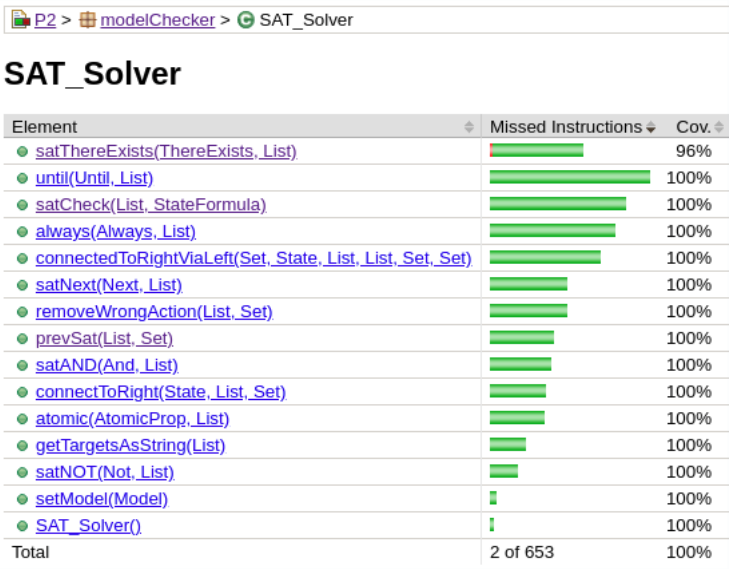

Figure 4: Coverage

The above screen shot shows a missed instruction in satThereExists. Note that this is dues to the fact that is a default in a switch statement, which only occurs with malformed models.

In addition to tests covering different aspects of the model, tests have been implemented to prove that the conversion presented in 2.2 is done adequately. This can be found in the class ENFChecks.

A summary of the distribution of checks and their success rate is given below:

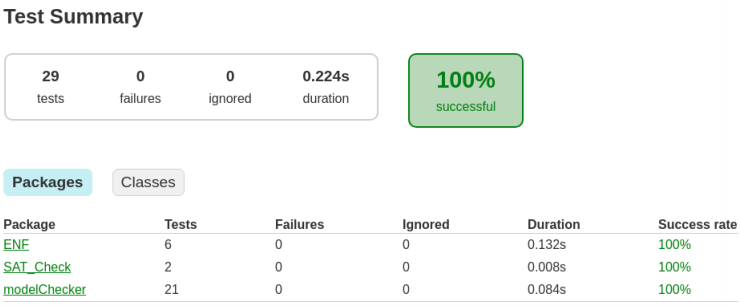

Figure 5: JUnit summary

4.1 Mutual Exclusion

To show that model checker successfully tests a mutual exclusion algorithm, a model from lectures was used, please see [5] for details. Two processes have critical sections c_i and waiting section w_i for $i \in 1, 2$. In Figure 6 the reachable states are displayed.

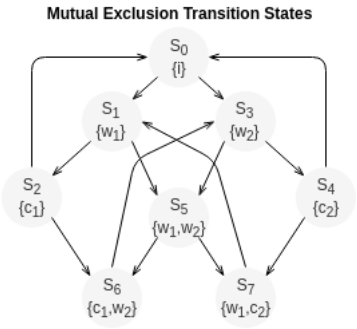

Figure 6: Transition State Diagram

As expected the model checker reports that the formula $AG(!c1||!c2)$ is valid for the model above.

5 Extension

The implementation related extensions covered in this practical include:

- The conversion to **ENF**: Using ENF as logical structure for the SAT solver adds a little bit more computational effort in pre-processing, but allows to simplify the computation of the SAT solver, which ultimately yields a slightly better efficiency.
- The treatment of **terminal states**: in our implementation, terminal states automatically receive a transition looping to themselves. This simplifies input for the user.
- More than one property uses **actions**: the basic specification demand that one property supports actions. In our implementation more than one property supports actions.

5.1 Visualization

In an effort to improve understanding why a model might have failed, we introduced a visualization tool, which displays a graph and colours the paths which cause a formula to fail in red. Initial nodes are coloured in orange. The tool is based on the library GraphStream and can be found in src/main/java/graph. In this directory there is also an example how to use the code. Revisiting the problem from Figure 3, the library plots the following:

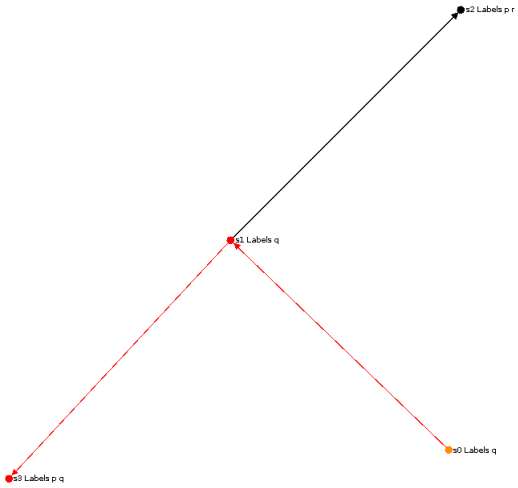

Figure 7: Graph with failed Path

The colour coding is the following:

- **Orange**: Initial states serving as stating point for a failed path
- **Red**: States part of the path not satisfying the query
- **Black**: States not part of a failed path
- **Blue**: Initial states not part of a failed path

Unlike before it is now clearly visible which path is taken to prove that the statement failed. This allows a much easier handling of statements of disproving $\exists$ statements. All states had to printed out, now the graph will just be displayed in red, making the investigation much easier.

Not that for larger graphs the overview can be fairly complex and that actions are not displayed in the graph to improve readability.

6 Petri Nets

Petri nets are were introduced by Carl Adam Petri in the 60s and have evolved since then. Being used for multiple purposes, such as true concurrent modelling, they come in various shapes and form. One of these is a place/transition net, which is one of the more common forms of Petri nets. As a purely logic approach, aims to represent the order in which actions occur rather than represent an actual program. Petri nets have a major advantage in capturing systems with a large state space into a compact mathematical and graphical presentation. Many of the states don't need to be defined explicitly, but given the initial states, it is possible to derive them on the fly. This allows to even describe an infinite number of states in a finite notation. Another advantage is the representation of subnets, e.g. independent processes within a larger net [6].

This formulation brings up the question in what way it differs from a concurrent processes modelled previously. If two systems were meant to be concurrent,

say action a and b, then a valid TS includes two paths, one with a first, then b and b first and then a, which implies that one action finishes before the other one starts. This however is different for *true* concurrency. To have a and b happening at the same time means that one can start and finish at any point in time [7].

Unlike dividing a system into states and actions, Petri nets, use different structures. There are *tokens* which flow through the system. There can be one or more tokens, in every *place*. Each place can contain tokens for an extended period of time, before passing the tokens off to through *transitions*. These three elements are in fact the only structures need to define a Petri net. As there can be multiple tokens in the system, multiple transitions can be fired at the same time, which brings true concurrency.

Assuming a simplified specification for true concurrency, this now allows to turn the specification into a parse tree itself, but with more edges and hence more states. For the above given example, not only a then b and b then a, but also a and b at the same time and this is only assuming that they take the same time to execute, e.g. one unit in system time. This idea can be extended to n actions happening at the same time, however, note that n needs to be finite to be easily assembled into a Finite State Machine. This of course massively increases the state space and leads to the problem of state space explosion.

This problems led to the development of another way of thinking about states. The approaches above all create a computation tree, Kenneth McMillan developed a technique which captures the unfolding [8] instead. By doing so, it became possible to use standard LTL with the newly developed structure, for example in [9], which requires to rethink how model checking works in this submission.

7 Conclusion

The submitted code fulfills all requirements as set by the practical specification and goes beyond in certain areas. Models can be checked and traces can be viewed in case a model fails. AsCTL is explained and equivalences were discussed. Our testing class has extensive coverage and includes a mutual exclusion check. Multiple extensions, both on a conceptual level as well as implementation level, have been implemented and were mentioned above. Further, Petri nets were introduced as a different approach to concurrency and as stepping stone to develop other forms of model checking using the unfolding of graphs instead of the whole computation tree.

Overall, this practical helped in making sense of the course content and helped us to understand the implications of certain tokens in the context of CTL and asCTL.

References

[1] C. Baier and J.-P. Katoen, *Principles of Model Checking (The MIT Press)*. The MIT Press, 2008.

[2] Z. Hassan, A. R. Bradley, and F. Somenzi, "Incremental, inductive ctl model checking," in *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, (Berlin, Heidelberg), pp. 532–547, Springer-Verlag, 2012.

[3] J.-P. Katoen, "Ctl model checking," 2018.

[4] D. Mulligan, "Hoare logic and model checking," 2018.

[5] J. Bowles, "Lecture nine lsv," 2018.

[6] S. C. Cabasino M.P., Giua A., *Control of Discrete-Event Systems: Automata and Petri Net Perspectives (Lecture Notes in Control and Information Sciences)*. Springer, 2012.

[7] J. Esparza, *Model Checking Software: 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010, Proceedings (Lecture Notes in Computer Science)*. Springer, 2010.

[8] K. L. McMillan, *Symbolic Model Checking*. Springer US, 1993.

[9] J.-M. Couvreur, S. Grivet, and D. Poitrenaud, "Designing a ltl model-checker based on unfolding graphs," in *Application and Theory of Petri Nets 2000* (M. Nielsen and D. Simpson, eds.), (Berlin, Heidelberg), pp. 123–145, Springer Berlin Heidelberg, 2000.