

Practical 2: A Simple Model Checker

STUDENT IDs: 120017875, 110021627

MODULE: CS4052 LOGIC AND SOFTWARE VERIFICATION

LECTURER: JULIANA BOWLES

November 23, 2015

Project Description

The aim of the practical is to create a simple model checker for asCTL – an action and state based logic. The model should support fairness conditions (also expressed using asCTL) and provide a trace of execution to show what path satisfies or fails the given formula.

The first part of the report will give an insight in the mechanisms behind our formula evaluations. Then we will move on to describe how constraints are Incorporated, and how trace is produced. We will also discuss a realistic state transition model, and how well our module checker works on determining whether various formulae hold for this model. Finally, we will outline the strengths and limitations of the model, and the problems that we encountered while working on the practical.

Evaluation of Formulae

The formulae supported by the as-CTL can essentially be divided into two classes – state formulae and path formulae. They are defined by the following grammar:

State Formula $\Phi := \text{true} \mid p \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \Phi \rightarrow \Phi \mid \Phi \leftrightarrow \Phi \mid \exists\phi \mid \forall\phi$

Path Formula $\phi := \Phi U \Phi \mid \Phi_A U_B \Phi \mid \Phi_A U \Phi \mid \Phi U_B \Phi$

where p is an atomic proposition, and A and B are action sets that restrict the transitions that we consider for the path operator. We also consider X, F and G path operators that stand for "next", "eventually" and "always", respectively. U stands for "until" and is to be understood as the first formula holding in all of the states visited while actions of the set A are taken, and then at some point we can take an action of a set B and that leads us to a state where the second formula holds.

In our implementation formulae are evaluated recursively – we look at the main operator of the formula, take the formulae it connects (or the single formula in case of X, F, G and negation), evaluate them separately and then combine the results of evaluation depending on what the main operator is. The base cases are tautologies and atomic propositions. The way these are evaluated, and the way the operators connect them together is discussed in the following subsections.

Evaluating tautologies

In the case of tautologies, evaluation is trivial. Independently of the state which we evaluate them in, "True" will always return true, and "False" will return false. To check whether a formula is tautology or not we use the provided method *isSingleTt*.

Evaluating APs

Each state in the model can have multiple labels assigned to it – they stand for the atomic propositions that hold in this state. Therefore to check whether an atomic proposition holds at a state, we loop through all labels of the state until we have found the one corresponding to this proposition. We also need to check whether the atomic proposition asked for is a negation (it is obtained using `getApNeg` and is not reflected by `isNegation`, which makes the matters slightly more complicated). If the AP that we are looking for is a negation, and we find that it amongst the labels, we return false. If we do not find this AP, we return true. Similarly, if it is not a negation, we return true when we find it, and false if we do not find it.

Evaluating negations of formulae

Before evaluating a formula, we check whether it is a negation using the provided `isNegation` method. We then evaluate the formula without taking the negation into account, and switch the obtained result to the opposite in case the formula was found to be negated.

Evaluating formulae with state operators

By "state operators" we mean \wedge , \vee , \rightarrow and \leftrightarrow – operators that are concerned only about their operands holding at the state that they are evaluated for, and by themselves do not require looking at any other states. To evaluate the result, we firstly evaluate both operands separately (notice that all of the operations mentioned are binary) and then combine the results obtained depending on the logical meaning of the operator. \wedge and \vee directly correspond to built in `&&` and `||` operators, while $\Phi_1 \rightarrow \Phi_2$ is logically equivalent to $\neg\Phi_1 \vee \Phi_2$. $\Phi_1 \leftrightarrow \Phi_2$ is evaluated by checking whether the results returned by Φ_1 and Φ_2 match.

Evaluating formulae with path operators

Path operators are U, X, F and G. For all of these, it is not enough to know what holds at the current state; to find out their values we need to investigate what holds for a path.

First of all, if the operator is X, G or F, we transform it to a formula with U. We do it not to over complicate the model checker – increasing the number of methods would lead to a more error prone and more difficult to test model. Plus, each of the other operators can be expressed using U in a way that the formula is logically equivalent to the original one, and testing it does not increase time and space complexities. Here are the equivalences we use:

- **F**: stands for "eventually" and $_A F_B(\Phi)$ can be translated to $True \wedge U_B \Phi$. This works, because `True` holds for any state, so the only way that the formula can not be satisfied is if we never find a state where Φ holds – when it is not the case that eventually Φ . Actions here restrict the transitions

that we consider. Thus we want an action of type B to lead us to a state where Φ holds, and to have taken only actions of type A beforehand.

- **G:** stands for "always" and $G_A(\Phi)$ can be translated to $\neg(F_A(\neg\Phi))$, as a formula always holds only if it is not the case that it eventually does not hold. By using the conversion of F to U we get $\neg(True \emptyset U_A (\neg\Phi))$. Notice that here we have only one action set. This is because there is no "before Φ holds" and "leading to Φ holding", as we want Φ to always hold. The action set that is specified should be regarded as "we want all the actions of type A to lead to a state where Φ holds".
- **X:** stands for "next" and $X\Phi$ is translated as $True \emptyset U_A \Phi$. We use empty set as the first action set, as we want Φ to hold in the next state – after a single transition. Therefore all we want to do is take a transition with an action of type B, and check whether Φ holds in the according state. Of course, for next, same as for all other path operators no action sets can be specified, which is regarded as all the actions being considered (different than an empty set of actions being specified!).

Existential and Universal quantifiers

Existential and Universal quantifiers tell us about where the formula has to hold, e.g. for \exists as long as one path in the model where the formula holds is found, the formula works and for \forall every single path has to satisfy the formula. The implementation of checks for these two quantifiers is done together to avoid repetition of the similar code twice. To be able to differentiate between two a boolean variable "forAll" (which is true if the formula has \forall or false in case of \exists) is passed to *getAllTransitions* and *checkUntil* methods. These are the only places where this information is essential.

- In the first method in case of \exists we only add the transitions that could lead us to the state where the formula is satisfied, otherwise if we have \forall as a quantifier, we have to add all the transitions. Although if we see a transition which does not contain any actions that are required by the formula, we already know that the formula would fail, we still have to continue the check in order to provide a trace with a counterexample.
- In the second method we need to know which quantifier it is in order to know at which point to stop the check. If we obtained a counterexample or a witness before we finished traversing the model, we can stop.

In the current model checker existential and universal quantifiers are always evaluated together with until, since other operators such as X, F and G are also expressed in terms of until. The method that checks whether until holds takes the following parameters:

- formula - the formula we are currently evaluating
- state - the current state

- `actionName` - the action that was taken to get to the state we are currently in
- `transitionsToCheck` - a hashmap from state name to array list of the transitions that can be made from that state and that have not been checked yet
- `model` - the model we are checking formula against
- `actionsA` - a string array that contains all the actions in A (for $\Phi_A U_B \Phi$); this array is set to null if a subset of actions was not specified and is empty if no actions are allowed (used in the case where X operator was specified)
- `actionsB` - a string array that contains all the actions in B (for $\Phi_A U_B \Phi$) and same as with `actionsA` it can be null or empty
- `forAll` - a boolean variable described above
- `isNext` - a boolean variable that specifies whether the operator next was transformed into until

In the `checkUntil` method we split the formula into the left hand side Φ_1 and the right hand side Φ_2 for $\Phi_1 A U_B \Phi_2$. We then first check whether Φ_2 holds and the action that is stored in the `actionName` is of type B, if both these conditions hold, return true, otherwise check whether Φ_1 holds and the action in the `actionName` is of type A, if so, enter the loop with all the possible transitions that can be made from this state, and check whether the formula holds for each one of them by recursively calling `checkUntil` method, otherwise either ignore the path if `forAll` variable is false or return false otherwise.

Looping through the initial states

To check whether a formula holds for the model, we have to check its validity at the initial states. There are three cases to consider when looping through the initial states:

- The specified formula has \forall quantifier - if the formula does not hold when starting from one of the initial states we return false immediately, because there is no need to check other initial states as we already know that it does not hold for one of them. Otherwise we keep checking till the last initial state.
- The specified formula has \exists quantifier - if the formula holds for one of the initial states, we return true immediately, as existence of one path that satisfies the formula is sufficient. Otherwise keep searching till all of the initial states are exhausted.
- The specified formula is a state formula rather than a path formula - then the formula has to hold on all of the initial states. If there is a state where it does not hold found, we stop checking and return false immediately.

There is another aspect that has to be pointed out here – we already described how we have separate methods of evaluating state formulae and path formulae. But how do we know which method to call? This problem is solved by checking whether the formulae has a quantifier, as the grammar requires all path operators to be prefixed by one, but none of the state operators can have it. Thus whenever we see a quantifier, we call *checkPathFormula* and when we do not, we call *checkStateFormula*. We take the same approach all the way through the recursive formulae checking.

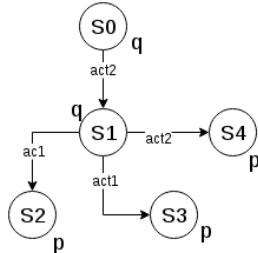
Evaluating constraints

Constraint evaluation happens before the formula check. There are several scenarios that need to be considered when evaluating constraints:

- Constraint has \forall quantifier and
 - Constraint holds - then the formula can be checked on the model without any restrictions
 - Constraint does not hold - then there is no need to check the formula
- Constraint has \exists quantifier and
 - Constraint holds - then the formula needs to be checked on those paths that satisfy the constraint
 - Constraint does not hold - then no paths are found where formula needs to be checked and again there is no need to check formula

The third case is the only one that requires some more work to be done while checking the formula. For this case, a hashmap *transitionsAfterConstraint* with all the state name to transitions mappings was created and when the constraint is checked, those that do not satisfy the constraint are removed from it. The formula is then checked with those transitions only rather than with the whole set from the model. This is implemented in the *getAllTransition* method; an extra check is added in which when we decide if the transition needs to be added to the hashmap to check we first see if this transition still exists in the *transitionsAfterConstraint*, if not we do not add it.

A simple example shows that it does work in the way it should. Consider a model below:



When we check if the formula $\forall aFb(p)$, $a=\{act2\}$, $b=\{act1\}$ holds we get a counterexample " $s0$ " $act1$ " $s1$ " $\{ \}$ $act1$ " $s2$ " $jump\ back$ $act1$ " $s3$ " $jump\ back$ $act2$ " $s4$ " which shows how it searched for the path that led to the state where the formula does not hold. $jump\ back$ goes back to the $\{ \}$ and continues from there.

On the other hand when we apply the constraint $\exists aFb(p)$, $a=\{act2\}$, $b=\{act1\}$ and then check the same formula again, the result is *All paths satisfy the formula*. More testing was done to check that the formula checks with constraints work. Although constraints with \exists quantifier are quite useful as they can limit the paths that we are taking when checking a formula, the constraint with \forall quantifier does not much sence as in our understanding it would be the same as to simply check whether a formula and the constraint work separately.

Providing a trace

Traces are quite important in model checking, they can show faults in the model design and help quite a lot with debugging. To implement traces when checking a model against a formula or a constraint we needed to keep track of which states we visited and which state we are currently at. In this way when we need to provide a counterexample for formulas with \forall or a witness for formulas with \exists it can be done quite easily.

If the formula contains a state formula only, it either holds on all initial states, or it does not. In the latter case, the initial state where it does not hold is added to the trace.

If a path formula is specified it made sense to provide a trace in two cases:

- When the formula contains \exists quantifier and the formula is satisfied, the trace for where the formula works (a witness) is provided
- When the formula contains \forall quantifier and the formula is not satisfied, the trace with the counterexample is created

In other cases a corresponding message is created: "No paths that satisfy the formula found" or "All paths satisfy the formula".

The trace is created in the format "state name", action taken, "state name", etc. In addition, every time there are several options of transitions that can be taken we add a separator to the trace and when that branch fails in case of \exists or is satisfied in case of \forall and we are going back to the state where we made a decision we add "jump back" to the trace.

Testing

As it is very difficult to tell what has gone wrong when an incorrect result is returned by the model checker, and whether all the formulae are still validated correctly after slight adjustments to one of the checker methods have been made; we decided to write unit tests. It also goes nicely with the recursive structure

of the model checker, as we can see exactly at which level the evaluation has gone wrong, if something does not work. Thus we wrote 39 unit tests in total, starting from testing tautologies and atomic propositions to evaluating complex nested formulae for bigger models with multiple initial states. All of the models, formulae and constraints used for tests can be found in the folder *ourTests*.

Experimentation

To show that our implementation works correctly we decided to consider a famous challenge in which there are a ferryman, a wolf, a goat and a cabbage on one side of the river and they all need to get to another side. The problem is that the only way they can get to another side is by boat which can carry at most two passengers. For obvious reasons one of the travelers has to be a ferryman but he can also travel alone. Furthermore, the goat and the cabbage cannot be left unobserved and the same applies to the goat and the wolf. The question is whether it is possible to get them all from one side to another.

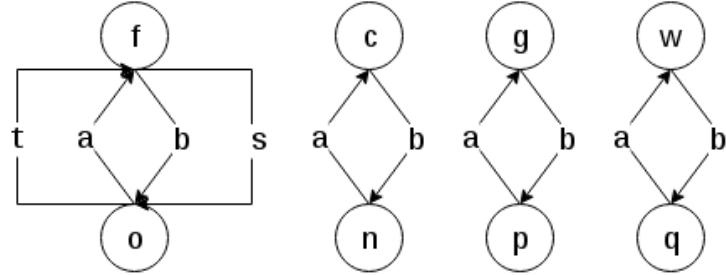
In the model the following labels and actions were used:

- c - cabbage is on the left side of the river
- f - ferryman is on the left side of the river
- g - goat is on the left side of the river
- w - wolf is on the left side of the river
- n - cabbage is on the right side of the river
- o - ferryman is on the right side of the river
- p - goat is on the right side of the river
- q - wolf is on the right side of the river
- s - ferryman travels alone from the left side to the right side
- t - ferryman travels alone from the right side to the left side
- b - ferryman brings a cabbage or a goat or a wolf from the left side to the right side
- a - ferryman brings a cabbage or a goat or a wolf from the right side to the left side

States on the model that we created and can be found in the cabbageGoatWolf-Model.json file have names such as "cf" meaning that a cabbage and a ferryman are on the right side of the river and that the rest are on the left side, or "none" meaning that all of them are on the left side, which is the initial state. On the diagram of the model for better visualisation we will write labels that are true

at states rather than state names.

Therefore, the separate models for all the participants look like this:



The combined model is shown below. This is the modified model from the one found in Principles of Model Checking (C. Baier, J.P. Katoen, 2008).

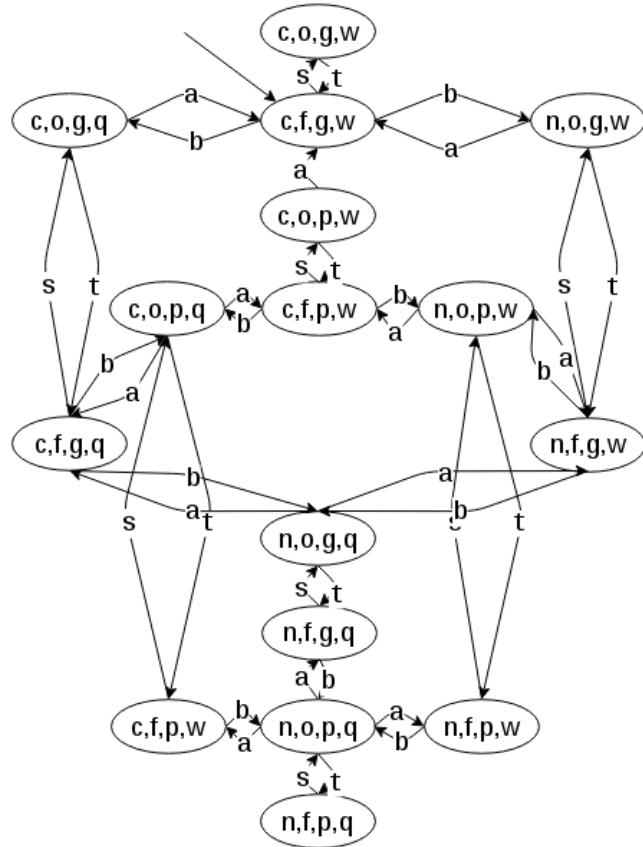


Figure 1: Full model for wolf,goat,cabbage problem

This model is very well designed that is why we had troubles with finding constraints to eliminate paths, since they all are essential. Therefore, to check that the constraint works on this model an impossible transition was added from the initial state when they all are on the left side of the river to the state when they all are on the right side of the river. The specified constraint is then

$$\exists X (!n||!o||!p||!q)$$

After the constraint is applied we see that the path was removed (for this print statement was added and it showed *Removed none-[s]-cfgw*) and the properties are checked.

The properties that we checked are the following:

- It is possible for a cabbage, a goat and a wolf to eventually be moved from the left side to the right side of the river, or the more direct translation of the formula is, for each of them if a participant is on the left side, then there exists a path when eventually this participant gets to the right side by taking action b. The formula can then be represented as

$$(((c \rightarrow (\exists x Fy(n))) \wedge (g \rightarrow (\exists x Fy(p)))) \wedge (w \rightarrow (\exists x Fy(q))))$$

where x and y are sets of actions, x={a,b,s,t} and y={b}

- The ferryman always has to move from one side to another by taking legal actions. In other words, for all the paths it is always the case that when a ferryman is on the left side, then for all the paths he gets to the next state by taking either an action s or b and in that next state he is on the right side of the river or the other way round with actions t or a. The formula then looks like this

$$\forall G((f \rightarrow (\forall X y(o))) \parallel (o \rightarrow (\forall X x(f))))$$

where x={t, a} and y={s,b}

- For every state it is always the case that if a participant is on the left side of the river, it cannot be at the right side at the same time and the other way round. The formula for this is

$$\forall G((c \rightarrow !n) \wedge (f \rightarrow !o) \wedge (g \rightarrow !p) \wedge (w \rightarrow !q) \wedge (n \rightarrow !c) \wedge (o \rightarrow !f) \wedge (p \rightarrow !g) \wedge (q \rightarrow !w))$$

- The main property of the challenge, it is possible for all of the participants to be moved from the left side to the right side by making legal actions and without leaving a cabbage with a goat or a goat with a wolf unobserved on either sides of the river. The final formula is the one below when eventually this participant gets to the right side by taking action b. The formula can then be represented as

$$\exists (((((w \wedge g) \rightarrow f) \wedge ((c \wedge g) \rightarrow f)) \wedge (((q \wedge p) \rightarrow o) \wedge ((n \wedge p) \rightarrow o))) U y(n \wedge o \wedge p \wedge q))$$

where $y=\{b,s\}$

Splitting formula into parts makes it easier to understand, thus

$$((w \wedge g) \rightarrow f) \wedge ((c \wedge g) \rightarrow f)$$

means that if a wolf and a goat are on the left side, then there is also a ferryman there and if there is a cabbage and a goat on the left side, then the ferryman is also there;

$$((q \wedge p) \rightarrow o) \wedge ((n \wedge p) \rightarrow o)$$

is the same but on the right side;

And the whole formula says that these both conditions are true until they are all on the right side of the river and the actions that have to be made before they are all on the right side are either b or s.

The results obtained for each of them were:

- The witness for the formula was: "none" {} s "f" t "none" {} b "cf" "none" {} s "f" t "none" {} b "cf" {} a "none" {} b "fw" t "w" {} s "fw" jump back b "fgw" "none" {} s "f" t "none" {} b "cf" {} a "none" {} b "fw"

This witness does show that the property is satisfied since when we get to the "cf" state the first part of the formula is satisfied, when we get to the state "fgw" the second part is satisfied and finally at fw the last part is satisfied. We can also see that at fgw the last part was already satisfied, but since we are splitting the formula and checking it in parts we did not notice at that point that the last part also worked.

- The result is *All paths satisfy the formula*
- The result again is *All paths satisfy the formula*
- The witness is "none" {} s "f" jump back b "cf" jump back b "fw" jump back b "fg" {} a "none" jump back t "g" {} s "fg" jump back b "cfg" {} a "g" b "fgw" {} a "g" jump back a "w" {} s "fw" jump back b "fgw" t "gw" jump back b "cfw" {} a "c" s "cf" jump back b "cfg" {} a "c" b "cfw" {} a "w" jump back t "cw" {} s "cfw" jump back b "cfgw". If we get rid of all the paths that were tried out, but were unsuccessful, we would get the sequence: the ferryman takes the goat to the right side(state "fg"), leaves it there and travels back alone (state "g"), takes the wolf to the right side (state "fgw"), takes the goat back to the left side (state "w"), takes the cabbage and brings it to the right side (state "cfw"), travels back alone (state "cw"), takes the goat to the right side (state cfgw). At this point the formula is satisfied and at neither of these states the goat and the cabbage or the goat and the wolf were left unobserved.

All the json files (the model, the formulas and the constraint) used can be found in the resources folder.

Strengths and limitations of the model

By introducing quantifiers, CTL gains additional expressiveness, when compared to LTL. LTL is a linear time logic – it interprets paths linearly as sequences of states. While CTL is a branching time logic and introduces the concept of many different possible futures, depending on which transition we choose to take (which branch we choose). Quantifiers \forall and \exists express the idea that a property holds for all possible futures or for some of them. as-CTL extends this idea even further, allowing to specify action sets that we are interested in. Perhaps, we want to check whether starting some processes (actions of type S) will never lead to a system overload (let us denote it by Φ). This can be expressed in as-CTL as $\neg\exists F_B(\Phi)$.

However, this is a simplified model, and does not account for, say, changing variables whose value is influenced by taking certain actions. It also does not allow us to test time sensitive systems, as there is no way of expressing how long an action takes or for how long we can stay in each state.

Extensions

In an attempt to improve efficiency of the model checker several changes in the implementation were made:

- When iterating through initial states and while evaluating path formulas the check is interrupted when a witness or a counterexample is obtained in cases with \exists or \forall quantifiers, since we already have the results, there is no need to keep traversing the model. Initially, the checker would do it anyway, thus carrying out extra work and being less efficient.
- In the case when there is a \exists quantifier in the formula, *getAllTransitions* method that is used as a helper method for *checkUntil* only adds those transitions to the hashmap that can potentially lead us to the searched path (the ones that are specified in the given action sets), therefore decreasing the number of paths that are checked within *checkUntil* method.

Furthermore, it was considered to improve efficiency by checking constraint and formulas on paths simultaneously, thus traversing the model only once, but it appeared to be too complicated in a case when the formula is satisfied on a path before the constraint is. We then have to move forward to check whether the constraint will be satisfied and if so only then do the backtracking. Combining results and continuing the check from the correct place required making lots of changes to the existing code and thus the idea was not implemented.

Problems encountered

Although having a parser and the structure for Formula and Model already implemented greatly simplified the task, we ran into several problems when

it comes to the setup. Firstly, we found that it is not possible to pass only one action set to X, F and G, it simply does not get evaluated. That made testing X and G with specified actions difficult. The second biggest problem was the fact that negation of formulas was not expressed in a consistent manner. Negations for atomic propositions are stored using boolean array, yet negation of any other formulae will be expressed by the single boolean variable *negation*. We think that it would have been easier to regard atomic variables as formulae by themselves (so a single AP would be a formula, and its negation would also be expressed by *negation*). Finally, unary operators X, F and G are not stored as operators but rather as part of quantifiers (so $\exists X(b \& \& a)$ would return $\&\&$ as the operator and $\exists X$ as quantifier), which should not be the case. Also, it is impossible to specify action sets when creating a new formula (which we do when we, for example, express X, F and G using U), therefore we found that we need to pass the action sets to *checkUntil* explicitly.

Running the program

A ModelChecker.jar can be used to run the program and further instructions need to be followed.

Bibliography

C. Baier, J.P. Katoen, 2008 *Principles of Model Checking*, The MIT Press