CS4052 - Practical 2

[190026921]

November 19, 2021

Word Count: 4343

Contents

1 Introduction and Overview

In this practical, an attempt was made at developing a model checker for *asCTL*. The *asCTL* logic extends *Computation Tree Logic* (CTL) by allowing the specification of additional conditions on the sequences of actions that are carried out in a transition system before reaching a given state.

In the following section, the logic itself as well as a possible implementation of the algorithms are explained.

2 Grammar of *asCTL*

Given that *asCTL* is an extension of *CTL*, the semantics are similar. As with *CTL*, *asCTL* formulae consist of two types of relations, path formulae and state formulae.

Therefore the grammer of *asCTL* for a state formulae is given by

$$\Phi ::= true \mid p \mid \neg\Phi \mid \Phi \wedge \Phi \mid \exists\varphi \mid \forall\varphi \tag{1}$$

and for a path formulae by

$$\varphi ::= \Phi\mathcal{U}_B\Phi \mid \Phi_A\mathcal{U}_B\Phi \mid \Phi_A\mathcal{U}\Phi \mid \Phi\mathcal{U}\Phi. \tag{2}$$

Here, $A, B \subseteq Act$, $p \in AP$, Φ is a state formula, and φ is a path formula.

3 Semantics of *asCTL*

Given a state formula Φ, we can reason about whether the formula holds in a given state s. More formally, $s \models \Phi$ holds if and only if the formula Φ holds in state s. Similarly, given a path formula ϕ and a path π, then ϕ holds for π if and only if $\pi \models \Phi$. So far, this corresponds to *CTL*.

Adding to that, *asCTL* introduces the concept of action-indices. Given the set of actions *Act* of a transition system TS, and two subsets $A, B \subseteq Act$, *asCTL* allows us to reason about the actions taken to reach a given state. In this report, we refer to an action index *before* the operator as the *pre-action index*, and to an action index *after* the operator as the *post-action index*. For the *until* operator, the action-indices work as follows.

The path formula $\pi \models \Phi_1 \ {}_A\mathcal{U}_B \ \Phi_2$ if and only if there is state s_i in path π such that $s_i \models \Phi_2$, with the action $\alpha_i \subseteq B$ leading into s_i, and until then all states s_k with $0 \leq k < i$ satisfy $s_k \models \Phi_1$ and are entered via $\alpha_k \subseteq A$.

Similarly, if one of the two action indices is missing, the corresponding part of the condition is omitted. In other words, given $\pi \models \Phi_1 \ \mathcal{U}_B \ \Phi_2$, we still require that $s_i \models \Phi_2$ and action $\alpha_i \subseteq B$ leading into s_i, and until then all states s_k with $0 \leq k < i$ satisfy $s_k \models \Phi_1$. However, we do not impose any restrictions on the action set of α_k. The case for $\pi \models \Phi_1 \ {}_A\mathcal{U} \ \Phi_2$ is analogous and restrictions are put on α_k but not on α_i.

Note that we assume that if an action index set $A = \varnothing$ or $A = Act$, we treat it as if the action index set were missing. The reason for this assumption is the following. Assume that $A = \varnothing$. This would mean that no action would be legal to enter the state, which would be nonsensical. Likewise, if $A = Act$, all actions are legal, which equates to the action index not being present at all. Note that this also implies that since the action indices are optional or can be specified as either $A = \varnothing$ or $A = Act$, we can obtain any valid CTL formula using $asCTL$.

Lastly, we also include the next operator $\mathcal{X}$ that is used to reason about the *next* state in a path. In terms of action-indices, we may only define a single action-index for the operator as we are only considering a single transition (write ${}_A\mathcal{X}$).

With this, we can define the satisfaction relations for $asCTL$ (see Equation 3). These relations were adapted from Baier and Katoen [2008, p.321] from the relations for CTL.

Let $a \in AP$ be an atomic proposition and $TS = (S, Act, T, I, AP, L)$ be a transition system. Further, let $s \in S$, and Φ and Ψ be valid $asCTL$ sate formulae and φ be a valid path formula. As before, $A, B \subseteq Act$ are subsets of the action set. The satisfaction relation $\models$ is defined for state formulae by

$$\begin{aligned}
s \models a \quad &\Leftrightarrow \quad a \in L(s), \\
s \models \neg\Phi \quad &\Leftrightarrow \quad \neg(s \models \Phi), \\
s \models \Phi \wedge \Psi \quad &\Leftrightarrow \quad (s \models \Phi) \wedge (s \models \Psi), \\
s \models \exists\varphi \quad &\Leftrightarrow \quad \pi \models \varphi \text{ for at least one } \pi \in Paths(s), \\
s \models \forall\varphi \quad &\Leftrightarrow \quad \pi \models \varphi \text{ for all } \pi \in Paths(s),
\end{aligned} \tag{3}$$

and further for path formulae by

$$\begin{aligned} \pi \models {}_A\mathcal{X}\Phi \quad &\Leftrightarrow \quad \pi[1] \models \Phi \wedge \alpha_1 \in A \\ \pi \models \Phi\, {}_A\mathcal{U}_B \Psi \quad &\Leftrightarrow \quad \exists j \geq 0 \ s.t. \\ &\qquad \pi[i] \models \Psi \wedge (\forall 0 \leq k < i, \pi[k] \models \Phi) \wedge \\ &\qquad \alpha_i \in B \ \wedge \alpha_k \in A \end{aligned} \tag{4}$$

where $\pi[i]$ denotes the $(i+1)$th state on the path π, and $\alpha_i \in Act$ is the action leading into $\pi[i]$. As explained above, if A or B are equal to $\varnothing$ or Act, the action index is treated as if no action restrictions were given.

In terms of the semantics for transition systems, the satisfaction set $Sat(\Phi)$ of the transition system TS for the *asCTL* formula is given by

$$Sat(\Phi) \models \{s \in S \ : \ s \models \Phi\}. \tag{5}$$

The transition system satisfies the formula Φ if and only if Φ hods in all initial states, that is

$$TS \models \Phi \quad \Leftrightarrow \quad I \subseteq Sat(\Phi). \tag{6}$$

4 Derived Path Operators

In addition to the $\mathcal{U}$ and $\mathcal{X}$ operators encountered in Section 3, we can derive two additional path operators.

Eventually $\mathcal{F}$ operator

The $\mathcal{F}$ operator can be used to describe that a formula will *eventually* hold. As with the $\mathcal{U}$ operator, two action indices are used and the interpretation is analogous in this case. The following satisfaction relation shows how the $\mathcal{F}$ operator can be derived from the $\mathcal{U}$ operator.

$$\begin{aligned} \pi \models {}_A\mathcal{F}_B\Phi \quad &\Leftrightarrow \quad \pi \models true \ {}_A\mathcal{U}_B \ \Phi \\ &\Leftrightarrow \quad s_i \models \Phi \text{ for some } i \geq 0 \ \wedge \\ &\qquad \alpha_k \in A \wedge \alpha_i \in B, \text{ for all } 0 \leq k < i. \end{aligned} \tag{7}$$

Always $\mathcal{G}$ operator

Secondly, the $\mathcal{G}$ operators requires that a state formula will *always* hold. Here, as with the $\mathcal{X}$ operator, only one action index is used. Since the formula has to hold in every state along the path, differentiating between an action before the operator and after the operator would be nonsensical. The interpretation of the action index in the case of the $\mathcal{G}$ operator is analogous

to that of the $\mathcal{X}$ operator. The following satisfaction relation shows how the $\mathcal{G}$ operator can be derived from the $\mathcal{F}$ operator (which in turn can be derived from the $\mathcal{U}$ operator).

$$\begin{aligned} \pi \models {}_A\mathcal{G}\Phi \quad &\Leftrightarrow \quad \pi \models \neg {}_A\mathcal{F}_A \neg\Phi \\ &\Leftrightarrow \quad \pi[i] \models \Phi \wedge \alpha_i \in A,\ \forall i \geq 0 \end{aligned} \tag{8}$$

5 Logic Equivalences

Many *asCTL* formula can be simplified or at least transformed into another form by applying valid equivalence rules to them. To *asCTL* formulae Φ and Ψ are said to be equivalent if and only if $Sat(\Phi) = Sat(\Psi)$ for all transition systems over the same set of atomic propositions AP.

The following equivalence rules were derived from the equivalence rules of *CTL* as presented by Baier and Katoen [2008, p.330] and Mulligan [2016-2017].

Equation 9 contains the duality laws for path quantifiers. Using these equivalences, one can transform either path quantifier ($\exists$ or $\forall$) into the other.

$$\forall\ {}_A\mathcal{X}\ \Phi \quad \Leftrightarrow \quad \neg\exists\ {}_A\mathcal{X}\neg\Phi \tag{9a}$$

$$\exists\ {}_A\mathcal{X}\ \Phi \quad \Leftrightarrow \quad \neg\forall\ {}_A\mathcal{X}\neg\Phi \tag{9b}$$

$$\forall\ {}_A\mathcal{F}_B\ \Phi \quad \Leftrightarrow \quad \neg\exists\ {}_B\mathcal{G}\neg\Phi \tag{9c}$$

$$\exists\ {}_A\mathcal{F}_B\ \Phi \quad \Leftrightarrow \quad \neg\forall\ {}_B\mathcal{G}\neg\Phi \tag{9d}$$

$$\forall\ {}_A\mathcal{G}\ \Phi \quad \Leftrightarrow \quad \neg\exists\ (true\ \mathcal{U}_A\ \neg\Phi) \tag{9e}$$

$$\exists\ {}_A\mathcal{G}\ \Phi \quad \Leftrightarrow \quad \neg\forall\ (true\ \mathcal{U}_A\ \neg\Phi) \tag{9f}$$

$$\forall(\Phi\ {}_A\mathcal{U}_B\ \Psi) \quad \Leftrightarrow \quad \neg\exists(\neg\Psi\ {}_A\mathcal{U}_B\ (\neg\Phi \wedge \neg\Psi))\ \wedge\ \neg\exists\ {}_B\mathcal{G}\neg\Psi \tag{9g}$$

The two Equations 9a and 9b correspond trivially with the *CTL* case in which no action indices are used. If the Next operator requires an A-action to enter the next state in which Φ holds, then the same must be the case when the $\forall$ quantifier is converted to $\neg\exists$, and analogously for the opposite conversion.

Equation 9c and 9d are more complicated in the sense that the action index gets reduced from both a pre-index action set (A) and a post-index action set (B) to only a pre-index action set (B). The reason for this is that in the Eventually case, the state in which Φ holds must be entered by an action from the set (B). Converting this into the Always formula equivalent means that all transitions with a B action would lead to a state in which $\neg\Phi$.

For Equations 9f and 9e the pre-action index of the Always operator moves to the post-action index on the Until operator. Saying that all transitions via an action *always* lead to Φ being satisfied in the state that is entered, equates to requiring those same actions in the post-action index of the true-Until, that occur when entering the new state.

A similar line of reasoning may be employed for the last part of the right hand-side of Equation 9g. The right side of the until operator is (Ψ) reached via an action from the set B. Therefore, using the Always operator, we require that a B action does not always lead to a state in which $\neg\Psi$.

A key realisation from these duality laws is that all formulae using the $\forall$ quantifier may be expressed using the existential quantifier $\exists$ instead. This finding proves very useful for the model checking algorithm and is discussed in more detail in the following section.

6 Existential Normal Form for *asCTL*

6.1 Definition

The existential normal form (*ENF*) allows the simplification of *asCTL* such that they only use the operators $\exists\mathcal{X}$, $\exists\mathcal{U}$, and $\exists\mathcal{G}$. The main advantage is that this reduces the number of cases that a model checking algorithm has to handle, since the number of equivalence relationships necessary to fully define the logic is reduced. The model checking algorithm introduced in the next section therefore only takes ENF formulae as input.

The definition of ENF for *asCTL* is based on that for *CTL* as explained in Baier and Katoen [2008, pp.332].

The grammar of *asCTL* in ENF is given by

$$\Phi ::= true \mid p \mid \neg\Phi \mid \Phi \wedge \Psi \mid \exists_A \mathcal{X}\Phi \mid \exists(\Phi_A \mathcal{U}_B \Psi) \mid \exists_A \mathcal{G}\Phi. \tag{10}$$

Note that for any of the action-indices may be omitted to not impose a restriction on the types of actions allowed for the transition as was explained in Section 3.

6.2 Implementation

As the model checking algorithm will only accept ENF formulae as input, a converter is required that transforms the given *asCTL* into ENF.

In the implementation, this is handled by the `ENFConverter` class. To convert a formula to ENF,

the state formula is passed to the `convertToENF` method, which recursively transforms it into ENF. This is implemented using the Visitor pattern. Every `PathFormula` and `StateFormula` has its own `convert*` method, which in turns calls the corresponding converter method in the `ENFConverter` class.

For example, assume a top level formula of type `And` is passed to the `convertToENF` method. This will call the `And.convertToENF` method, which will delegate the call to the `ENFConverter.convertAnd` method. There, the `ENFConverter.convertToENF` method will be called with both the left and the right side of the original `And` formula.

The converter algorithm will keep recursing the formula until it reaches either a boolean proposition (which in ENF can only be `true`) or an atomic proposition, which is simply returned as such.

The collection of Equations 11 depicts the recursive calls made in the `ENFConverter`, where $f()$ represents a call to `ENFConverter.convertToENF`.

Formula	**Input**		**Output (ENF)**
AP	p	$\longrightarrow$	p
$Bool$	$true$	$\longrightarrow$	$true$
	$false$	$\longrightarrow$	$\neg\ true$
And	$\Phi \wedge \Psi$	$\longrightarrow$	$f(\Phi)\ \wedge\ f(\Psi)$
Not	$\neg\Phi$	$\longrightarrow$	$\neg\ f(\Phi)$
Or	$\Phi \vee \Psi$	$\longrightarrow$	$\neg(\neg\ f(\Phi)\ \wedge\ \neg\ f(\Psi))$
$Forall : Next$	$\forall\ {}_A\mathcal{X}\Phi$	$\longrightarrow$	$\neg\exists\ ({}_A\mathcal{X}\neg f(\Phi))$
$Forall : Always$	$\forall\ {}_A\mathcal{G}\Phi$	$\longrightarrow$	$\neg\exists\ (true\ \mathcal{U}_A\ f(\Phi))$
$Forall : Eventually$	$\forall\ {}_A\mathcal{F}_B\Phi$	$\longrightarrow$	$\neg\exists\ ({}_B\mathcal{G}\ \neg\ f(\Phi))$
$Forall : Until$	$\forall\ (\Phi\ {}_A\mathcal{U}_B\ \Psi)$	$\longrightarrow$	$(\neg\exists\ (\neg f(\Psi)\ {}_A\mathcal{U}_B\ (\neg f(\Phi) \wedge \neg f(\Psi)))) \wedge$
			$(\neg\exists\ {}_B\mathcal{G}\ (\neg f(\Psi)))$
$Exists : Next$	$\exists\ {}_A\mathcal{X}\Phi$	$\longrightarrow$	$\exists\ {}_A\mathcal{X}\ f(\Phi)$
$Exists : Always$	$\exists\ {}_A\mathcal{G}\Phi$	$\longrightarrow$	$\exists\ {}_A\mathcal{G}\ f(\Phi)$
$Exists : Eventually$	$\exists\ {}_A\mathcal{F}_B\ \Phi$	$\longrightarrow$	$\exists\ (true\ {}_A\mathcal{U}_B\ f(\Phi))$
$Exists : Until$	$\exists\ (\Phi\ {}_A\mathcal{U}_B)\ \Psi$	$\longrightarrow$	$\exists\ (f(\Phi)\ {}_A\mathcal{U}_B\ f(\Psi))$

(11)

These conversions can be derived from the equivalence relations described in Section 5. More details on the implementation of the ENF converter are provided in Section 8.1.2

7 Model Checking Algorithm

In this section a model checking algorithm for *asCTL* formulae is introduced. Note that this algorithm assumes that any formula passed as input is in ENF. This can achieved through the conversions described in the previous Section. The basic idea of the model checking algorithm is adapted from the *CTL* algorithm described by [Baier and Katoen, 2008, p.341].

7.1 Basic Idea: Satisfaction Sets

In essence, the model checking algorithm calculates the satisfaction set $Sat(\Phi)$ of the formula Φ (see Equation 5), and checks if all initial states are included in the satisfaction set (as per Equation 6). Note that this means that the algorithm effectively checks for all states in the model if they satisfy the provided formula (a so-called *global* model-checking procedure).

To compute the satisfaction set of the given formula, the algorithm recursively computes the satisfaction sets of all components of the formula. More specifically, the algorithm traverses the parse tree of the *asCTL* formula from bottom to top, while computing the satisfaction set for each node and then combining them on the next higher level. This approach works since all leaves of the parse tree are either atomic propositions or boolean propositions that are then combined by other operators (the inner nodes of the tree). The action-indices are attached to the inner nodes that represent the operators.

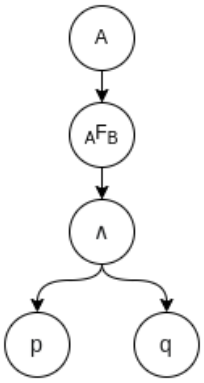


Figure 1: Simple parse tree of the *asCTL* formula $\forall\ {}_A\mathcal{F}_B\ (p \wedge q)$.

Figure 1 gives a simple example of such a parse tree for the *asCTL* formula $\forall\ {}_A\mathcal{F}_B\ (p \wedge q)$. The model checking algorithm would first compute the satisfaction sets $Sat(p)$ and $Sat(q)$. These two sets would then be combined on the next higher level, i.e. the node with the $\wedge$ operator. In this case, this would reduce the two satisfaction sets from the leaf nodes into one by taking

their intersection (due to the $\wedge$ operator, see Equation 14). At this stage, the algorithm has successfully computed $Sat((p \wedge q))$. Again, this would be passed to the next higher level in the tree, in this case the node with the operator ${}_A\mathcal{F}_B$. Now, a more sophisticated algorithm is employed to find the subset of the current satisfaction set for which the Eventually formula is satisfied (to be explained later). Lastly, the result is in turn passed to the next higher level, which is the top level node, where again another algorithm works out the updated satisfaction set.

Algorithm 1 formalises this process. The ComputeSat procedure computes the satisfaction set of a formula Φ, whereas the CheckModel procedure returns the actual result of the model checking by comparing the satisfaction set to the initial states. The Combine procedure within ComputeSat combines the satisfaction set received from the next lower level into a single satisfaction set according to the algorithm derived in the next section.

Algorithm 1 Basic *asCTL* model checking algorithm

```
procedure ComputeSat(Φ, TS)
    if (Φ is AtomicProp) or (Φ is BoolProp) then
        return Sat(Φ)                                   ▷ Reached a leaf node.
    end if
    return Combine(ComputeSat(Ψ, TS) for all Ψ in Φ)    ▷ (Ψ is sub-formula of Φ)
end procedure

procedure CheckModel(Φ, TS)          ▷ Φ: asCTL formula; TS: transition system.
    Sat(Φ) = ComputeSat(Φ, TS)
    if I_TS ⊆ Sat(Φ) then            ▷ All initial states must be in satisfaction set
        return true
    end if
    return false
end procedure
```

The same general principle works for pre *CTL* formulae, however the algorithms for the computation of the satisfaction sets in the inner nodes is different for *asCTL*.

7.2 Satisfaction Set Computation

In Algorithm 1, the crucial logic in the computation of the satisfaction set was abstracted in a call to the Combine procedure. In this section, the details of this are laid out as they form the most crucial part of the *asCTL* model checking algorithm.

7.2.1 Definition of the Satisfaction Sets for ENF

In the following, the computation of the satisfaction set for all types of possible ENF formulae are detailed.

Note that,

1. $a \in AP$ is an atomic proposition,
2. Φ, Ψ are *asCTL* formulae.
3. S is the set of states of the transition system,
4. $Sat(\Phi) = \{s \in S : s \models \Phi\}$ is the satisfaction set of Φ,
5. $Sat_{in}(A, \Phi) \subseteq Sat(\Phi)$
   s.t. all element states s have an *incoming* transition into s with action $\alpha \in A \subseteq Act\}$,
6. $Sat_{out}(A, \Phi) \subseteq Sat(\Phi)$
   s.t. all element states s have an *outgoing* transition into s with action $\alpha \in A \subseteq Act\}$,
7. $Sat_{in,out}(A, \Phi) \subseteq Sat(\Phi)$
   s.t. all element states s have an *incoming* and *outgoing* transition with action $\alpha \in A \subseteq Act\}$,
8. $Post(s) = \{t \in S : \exists$ transition from s to $t\}$ is the set of successors of s,
9. $Post_A(s) \subseteq Post(s)$ is the set of successors of s that are reachable via a transition from s that has an action from A,
10. $Post_{in,out:A}(s) \subseteq Post(s)$ is the set of successors of s that have an incoming and an outgoing transition with an action from A.

The following was adapted for *asCTL* from Baier and Katoen [2008, p.344].

Boolean Proposition
Since in ENF, boolean propositions are all true, we can simply return S.

$$Sat(true) = S \tag{12}$$

Atomic Proposition
An atomic proposition can be trivially evaluated by returning all states whose labels include the proposition.

$$Sat(a) = \{s \in S : a \in L(s)\} \tag{13}$$

Conjunction

A conjunction of two propositions may be combined by taking the intersection of the satisfaction sets of both individual propositions.

$$Sat(\Phi \wedge \Psi) = Sat(\Phi) \cap Sat(\Psi) \tag{14}$$

Negation

For a negation, the difference between the set of all states and the (recursively computed) satisfaction set of the proposition is the satisfaction set.

$$Sat(\neg\Phi) = S \setminus Sat(\Phi) \tag{15}$$

Exists Next

Exists Next cases are handled by returning all states that have a successor that satisfies the proposition Φ and has an incoming transition with the correct action.

$$Sat(\exists\ ({}_A\mathcal{X}\Phi)) = \{s \in S : Post(s) \cap Sat_{in}(A, \Phi) \neq \varnothing\} \tag{16}$$

Exists Until

The computation of the satisfaction set in the Exists Until case is difficult. The satisfaction set is the smallest subset T of S, such that two conditions hold:

$$\begin{aligned} Sat(\exists\ (\Phi\ {}_A\mathcal{U}_B\ \Psi)) = T \subseteq_{smallest} S,\ \text{s.t.} \\ Sat_{in}(B, \Psi) \subseteq T, \\ \forall s \in Sat_{out}(A, \Phi) : Post_{A,B}(s) \cap T \neq \varnothing \Rightarrow s \in T \end{aligned} \tag{17}$$

The first condition requires that all states with incoming action B that satisfy Ψ are included in the satisfaction set. This is required for the proposition Ψ *after* the Until operator.

The second condition requires that all those states from the satisfaction set $Sat_{out}(A, \Phi)$ are included in T whose successors are already in T. These successors must be reachable either by an *incoming* transition A or have an *outgoing* transition B. This ensures the left side of the Until operator, i.e. the Φ proposition.

Requiring the set T to be the *smallest* subset of S means we need to build the starting from $Sat_{in}(B, \Psi)$ and start adding states per the second condition until it is satisfied.

Exists Always

The Exists Always operator works in a similar fashion. The satisfaction set is the largest subset T of S, such that two conditions hold:

$$\begin{aligned} Sat(\exists({}_A\mathcal{G}\Phi)) = T \subseteq_{largest} S,\ \text{s.t.} \\ T \subseteq Sat_{in,out}(\Phi), \\ \forall s \in T : Post_{in,out:A}(s) \cap T \neq \varnothing \end{aligned} \tag{18}$$

The first condition ensures that T only includes states that satisfy Φ and have an incoming and outgoing transition with action A.

Secondly, all states included in T must have at least one successor with incoming and outgoing transition of action A (i.e., is a member of $Post_{in,out:A}(s)$) and is also in T.

Since we require the largest subset T of S for which the conditions hold, we can build the satisfaction set by iteratively adding states that satisfy the conditions to an initially empty set, until no more such states exist.

7.2.2 Algorithms for the Computation of the Satisfaction Sets for ENF

The mathematical definition of the satisfaction sets suffices for a characterisation of the sets but not for a programmatic implementation. Some of the definitions are trivially converted into algorithms (specifically those described by Equations 12 - 16).

However, for the computations for Exists Until (Equation 17) and Exists Always (Equation 18) the implementation can be facilitated by proposing appropriate algorithms.

Algorithm 2 $Sat(\exists\ (\Phi\ {}_A\mathcal{U}_B\ \Psi))$ computation algorithm

procedure SATEXISTSUNTIL(Φ, Ψ, TS)
  $E = Sat_{in}(B, \Psi)$ ▷ helper set from which contenders for T are drawn
  $T = E$ ▷ all states in $Sat_{in}(B, \Psi)$ initially support the formula

  while $E \neq \varnothing$ **do** ▷ no more contenders
    Let $s' \in E$ ▷ pick a contender
    $E = E \setminus \{s'\}$

    for all $s \in Pre_A(s')$ **do** ▷ direct predecessors s.t. can reach s' via A action
      if $s \in Sat_{out}(A, \Phi) \setminus T$ **then** ▷ must also satisfy Φ and not already in T
        $E = E \cup \{s\}$ ▷ add to E: new contender found
        $T = T \cup \{s\}$ ▷ add to T: new satisfaction state found
      end if
    end for

  end while
  return T
end procedure

Algorithm 2 (adapted from Baier and Katoen [2008, 351]) gives the procedure in detail to

calculate the satisfaction set for Exists Until type formulae of the form $Sat(\exists\ (\Phi\ {}_A\mathcal{U}_B\ \Psi))$. Note that, as mentioned earlier, the satisfaction set is the *smallest* possible set T satisfying the conditions from Equation 17. Hence, the procedure starts with only the satisfaction set $Sat_{in}(B, \Psi)$, which trivially satisfies the formula. From there on, sates are added to T if they (1) have a direct successor already in T reachable via an action $\alpha \in A$ and (2) satisfy Φ.

Algorithm 3 $Sat(\exists\ ({}_A\mathcal{G}\ \Phi))$ computation algorithm

```
procedure SatExistsAlways(Φ, TS)
    E = S \ Sat_{in,out}(A, Φ)          ▷ all unvisited states that do not satisfy the formula
    T = Sat_{in,out}(A, Φ)              ▷ superset of final satisfaction set with contender states

    for all s ∈ Sat_{in,out}(A, Φ) do
        count[s] = |Post_{in,out:A}(s)|                ▷ initialise hashmap count
    end for

    while E ≠ ∅ do
        Let s' ∈ E
        E = E \ {s'}

        for all s ∈ Pre_A(s') do                ▷ direct predecessors, s.t. can reach s' via A
            if s ∈ T then
                count[s] = count[s] − 1
                if count[s] = 0 then            ▷ s has no successor via A in T → remove
                    T = T \ {s}
                    E = E ∪ {s}
                end if
            end if
        end for

    end while
    return T
end procedure
```

Algorithm 3 (adapted from Baier and Katoen [2008, 351]) details the procedure for formulae of the type $Sat(\exists\ ({}_A\mathcal{G}\ \Phi))$. Here, as per Equation 18, the largest set T satisfying the conditions is to be found. This is achieved by starting out with the set of all states satisfying only $Sat_{in,out}(A, \Phi)$, which must be a superset of the final set T. From here on, states are iteratively removed from T if they are found to violate the formula. This is the case for states whose

successors reachable by an action from A are not in T themselves, i.e. remove states s for which $Post_{in,out:A}(s) \cap T = \varnothing$.

In the algorithm, this is achieved by keeping a counter for each state in $Sat_{in,out}(\Phi)$ (the hashmap `count`), that tracks the number of direct successors of s in $T \cup E$ with incoming and outgoing action from A. After initialisation, the value of `count[s]` in the algorithm is `count[s]` $= |Post_{in,out:A}(s) \cap (T \cup E)|$. Once this counter hits 0 for a state, we know that this state does not have the necessary successors to satisfy the formula and it is removed from T.

7.3 Counterexamples and Trace Generation

Counterexamples in *asCTL* allow for the identification of paths and states for which the provided formula does not hold. However, due to the nature of *asCTL* and their ability to reason about *possibility* of existence etc., such counterexamples cannot be generated for all formulae. For example, if one were to check if a path satisfying the path formula ϕ exists in a transition system, i.e. $\exists\phi$, and the model checker returns *false*, we cannot return a single counterexample that shows the violation of the formula. This is because only by knowing that *none* of the possible paths in the transition system satisfy the formula, can we disprove it. Therefore, counterexamples for path formulae with the existential operator do not make sense. Rather, one would return *witnesses* that indicate the satisfaction of such path formulae, however, this is out of scope for this practical (some details can be found in Shankar and Sorea [2003] and Clarke et al. [1995]).

Derived from the counterexamples for *CTL*, the generation of the counterexamples for *asCTL* are explained in the following for the operators assuming ENF [Baier and Katoen, 2008, p.377].

Next

The simplest case for the generation of a counterexample is the Next operator. Given a path formula $\phi = \forall\ {}_A\mathcal{X}\ \Phi$, a counterexample is given by a pair of states (s_0, s_1), where $s_0 \in I$ is an initial state and either

1. $s_1 \in Post_A(s_0)$ is a successor to s_0 reachable via an action A, and crucially $s_1 \not\models \Phi$, or
2. $s_1 \in Post_{Act \setminus A}(s_0)$ is a successor to s_0 reachable but not reachable via an action A.

Until

Given the path formula $\phi = \forall\ (\Phi\ {}_A\mathcal{U}_B\ \Psi)$, a counterexample is given by an initial path fragment (i.e., a path fragment starting from an initial state $s_0 \to s_1 \to ... \to s_k$), such that the path satisfies either

1. $\pi \models \mathcal{G}\ (\Phi \wedge \neg\Psi)$, that is Ψ is not reached, or
2. $\pi \models\ (\Phi \wedge \neg\Psi)\ {}_A\mathcal{U}_B\ (\neg\Phi \wedge \neg\Psi)$, that is once Φ no longer holds, Ψ also does not hold, or
3. $\pi \models\ (\Phi\ {}_C\mathcal{U}_D\ \Psi)$ where either $C \cap A = \varnothing$ or $D \cap B = \varnothing$, that is the Until operator itself is satisfied but the actions of the transitions to do so do not conform with at least one of the allowed action sets A and B.

Always
A counterexample for the formula $\phi = \forall\ {}_A\mathcal{G}\ \Phi$ is given by an initial path fragment $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_k$ such that either

1. $s_i \models \Phi$ for $0 \leq i < k$ and $s_k \not\models \Phi$, that is the path contains a state on which Φ no longer holds, or
2. $s_i \models \Phi$ for $0 \leq i < k$ and $\alpha_k \notin A$, that is the path contains a state that is reached with and action that is not part of the allowed set of action s A.

In order to implement this programmatically, one approach would be to traverse the parse tree from the bottom up (similar to how the satisfaction set computation works). On each level, the counterexample would be returned to the next highest level and combined with other paths from the level below, until the final counterexample is returned.

Due to time limitations, the generation of counterexamples was not included in the implementation (see Section 8).

7.4 Fairness

As per the specification, fairness can be supported in the model checker by adding constraints to checking procedure that limit the paths that are considered.

Further, note that fairness constraints are themselves *asCTL* formula. This presents a major advantage for the implementation and leads to an almost trivial solution to support fairness. Since both the actual formula as well as the constraint are valid *asCTL* formulae, the constraint is respected if the conjunction of the constraint and the main formula is taken.

Consider the case where a formula ϕ is to be checked and a constraint formula ψ is specified. To support fairness, the two are combined and the new formula $\phi \wedge \psi$ is passed to the model checker.

Note that this is only possible since the constraints themselves are valid *asCTL* formulae.

8 Notes on the Implementation

The practical part of this submission consisted of implementing the model checker for *asCTL* as outlined in this report. Overall, an attempt at this was made and significant parts of the checker were successfully implemented. However, the implementation is not fully functional yet and some features are still not implemented yet. The implementation is detailed in this section.

8.1 Components

In addition to the provided code, four main components are used in the implementation. The components are depicted in a rough class diagram in Figure 2.

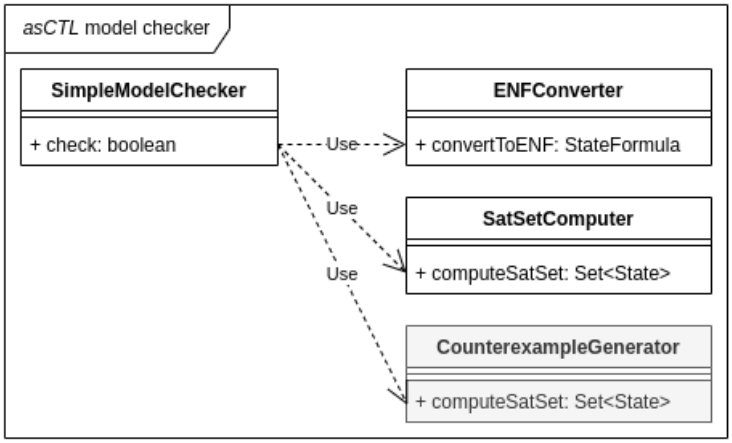


Figure 2: Implementation components. `CounterexampleGenerator` is greyed out as it has not been implemented.

8.1.1 SimpleModelChecker

The `SimpleModelChecker` is the central component with the public `check` method, which executes the necessary steps for model checking. Specifically, the input query is transformed using the `convertToENF` method of the `ENFConverter` class, which transforms it into ENF as per Section 6.

Secondly, the `SatSetComputer.computeSatSet` is called on the ENF query, which returns the final satisfaction set as per the algorithms described in Section 7.

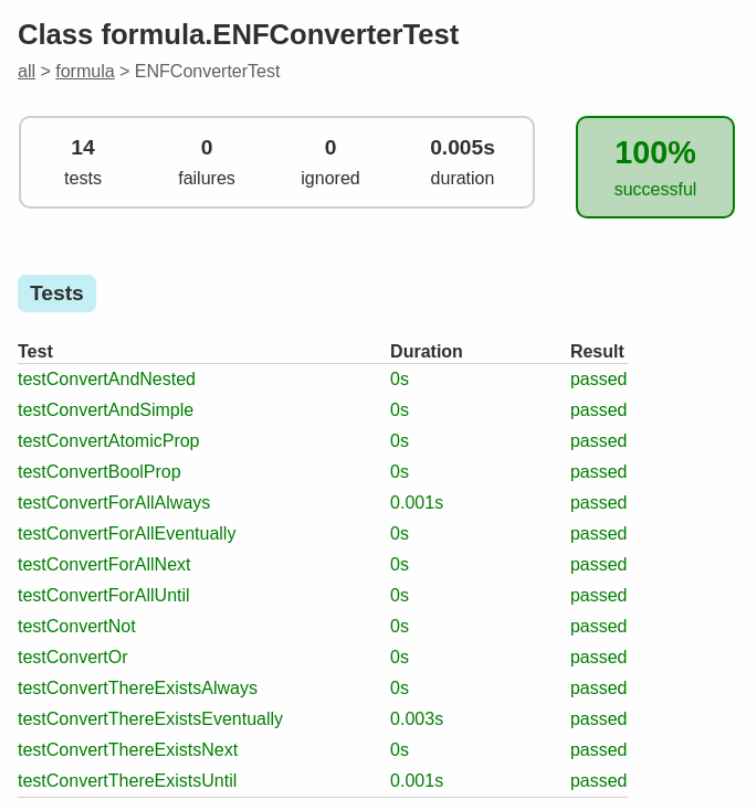
With this, the model checker compares the satisfaction set to the initial states and returns `true` if all initial states are contained. If not, the `CounterexampleGenerator` would be called to

generate a counterexample. Unfortunately, this has not been implemented yet.

8.1.2 ENFConverter

The `ENFConnverter` implements the ENF conversion algorithms explained in Section 6. The `convertToENF` method uses the visitor pattern to delegate to the appropriate conversion algorithm in a recursive manner as described before.

The `ENFConverter` is fully implemented and fully functional. The unit tests for the conversions all pass as shown in Figure 3.



Class formula.ENFConverterTest

all > formula > ENFConverterTest

14	0	0	0.005s
tests	failures	ignored	duration

100% successful

Tests

Test	Duration	Result
testConvertAndNested	0s	passed
testConvertAndSimple	0s	passed
testConvertAtomicProp	0s	passed
testConvertBoolProp	0s	passed
testConvertForAllAlways	0.001s	passed
testConvertForAllEventually	0s	passed
testConvertForAllNext	0s	passed
testConvertForAllUntil	0s	passed
testConvertNot	0s	passed
testConvertOr	0s	passed
testConvertThereExistsAlways	0s	passed
testConvertThereExistsEventually	0.003s	passed
testConvertThereExistsNext	0s	passed
testConvertThereExistsUntil	0.001s	passed

Figure 3: JUnit test report for the `ENFConverter`.

8.1.3 SatSetComputer

The `SatSetComputer` is arguably the backbone of the implementation, as it is responsible for computing the satisfaction sets that ultimately determine if a formula holds for the transition system.

As with the `ENFConverter`, the `SatSetComputer` has a public method (`computeSatSet`) that recursively explores the parse tree until the final satisfaction set is found. This, again, is implement with the visitor pattern: the `computeSatSet` method calls the `visit` method of the formula, which in turn calls the appropriate conversion method in `SatSetComputer` depending on the type of formula. This is a very clean way of implementing this, however, it does complicate the code base compared to simply using if-else statements in the `computeSatSet` method.

A vast number of algorithms implemented in `SatSetComputer` are functional. Unfortunately, the `visitAlways` method, which implements Algorithm 3 does not work reliably. I believe that I have identified the issue, although I was not able to find an appropriate solution: it appears that during the iterative reduction of the set T, initial states are sometimes removed when they should not be. The reason for this is that for some formulae, the initial states are valid members of the satisfaction set, however, they do not have any transitions with the required actions (since they are initial, after all). Despite numerous attempts of fixing this error, I was not able to arrive at a solution, which unfortunately renders this method useless (and in turn large parts of the model checker itself.)

However, the model checker is able to correctly parse *CTL* formulae, since the presented issue only occurs due to the action indices.

If the `visitAlways` method were to be fixed, the model checker should be fully functional.

8.1.4 CounterexampleGenerator (not implemented)

Due to time limitations, the `CounterexampleGenerator` was not implemented. However, an approach for a possible implementation was given in Section 7.3.

8.2 Testing

Overall 29 unit tests were implemented, of which 14 relate to the `ENFConverter`, 9 relate to the `SimpleModelChecker` using *CTL* formulae, and the remaining 6 test the `SimpleModelChecker` using *asCTL* formulae.

As required, all JUnit tests may be run using the build tool gradle.

```
$ cd <project_root>
$ ./gradlew clean build test coverage
```

Note that, since there are two unit tests that are not passing, the gradle build will raise an exception for the tests.

The resulting report can be found in the `build/reports/test/index.html` and the main result is depicted in Figure 4.

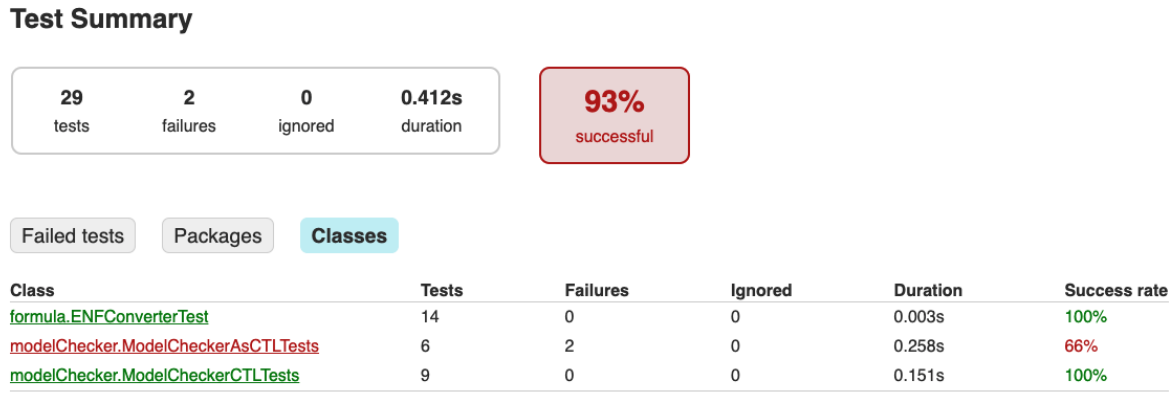


Figure 4: JUnit test report

The formulae used for testing are included in the `test/resources/test-formulae/` directory. Further, I included a transition system (`test/resources/test-models/model.json`) to test the *CTL* and *asCTL* formulae. A visual representatin of the model is provided in Figure 5.

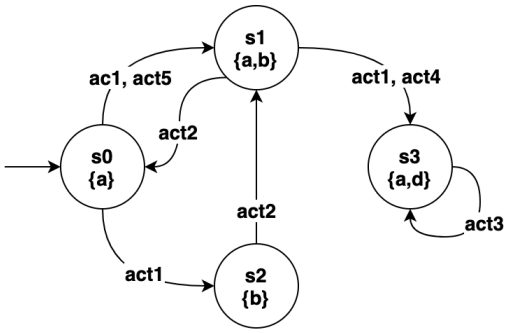


Figure 5: Model

The two tests that are failing are `model_exists_always` and `model_forall_always` of the `SimpleModelCheckerAsCTLTests` test class. The reason for these failures was explained in Section 8.1.3.

9 Conclusion

This report gave a detailed explanation of the steps and algorithms necessary to implement a model checker for the *asCTL* logic. The grammar and the semantics were discussed and the equivalence rules between significant formulae were explained. Crucially, the introduction of ENF for *asCTL* provides a convenient method to simplify formula and reduce the complexity of the model checker. Further, the generation of counterexamples as well as the concept of fairness was discussed briefly. For the model checking algorithm itself, a *CTL* approach was adapted to include the action-indices of *asCTL*. The algorithm recursively computes the satisfaction sets of the components of the provided formula, combining them with appropriate algorithms.

In terms of the implementation itself, significant parts of the model checker were successfully implemented. In particular, a converter for ENF was implemented as well as the `SatSetComputer`, which forms the backbone of the model checker. While the majority of the algorithms for the computation of the satisfaction sets was successful, the computation for formulae of the type Exists Always does not work reliably. This means that the model checker works for *CTL* formulae but fails to do so for *asCTL* formulae whose ENF formula requires the Exists Always operation. Further, I was not able to implement a counterexample generator due to time constraints.

If this work were to be expanded, fixing the issues with the Exists Always algorithms were a main priority, and would lead to a fully functioning *asCTL* model checker. Further, the counterexample generator would need to be implemented.

References

C. Baier and J.-P. Katoen. *Principles of model checking*. MIT press, 2008.

E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 427–432, 1995.

D. Mulligan. Hoare logic and model checking: Lecture 10, computation tree logic, 2016-2017. URL `https://www.cl.cam.ac.uk/teaching/1617/HLog+ModC/slides/lecture-10.pdf`.

N. Shankar and M. Sorea. Counterexample-driven model checking. *SRI International, Menlo Park, CA*, 94025, 2003.