Course Script

IN 5110: Specification and Verification of Parallel Systems

IN5110, autumn 2021

Martin Steffen

Contents

1 Computation tree logic **1**
1.1 Introduction 1
1.2 Syntax and semantics of CTL 5
1.2.1 Satisfaction relation 7
1.3 CTL model checking 8
1.3.1 Existential normal forms 8
1.3.2 Bottom-up treatment of compound formulas and *sat*-sets 9
1.3.3 Characterization of *sat* & fixpoint calculations 11
1.4 Symbolic model checking 18
1.4.1 Switching functions 18
1.4.2 Encoding transition systems by switching functions 23
1.4.3 Exploring the encoding for model checking 25

Chapter 1
Computation tree logic

What is it about?

Learning Targets of this Chapter

The chapter covers CTL and its extension CTL*, and a corresponding model checking approach based on BDDs.

Contents

1.1 Introduction 1
1.2 Syntax and semantics of CTL 5
1.3 CTL model checking 8
1.4 Symbolic model checking . . . 18

1.1 Introduction

This chapter covers another prominent temporal logic, *computation tree logic* or CTL. Material is taken from the books [1] or [2]. CTL is pretty established, so there is not much difference in the essence independent from where one looks.

The logic shares quite some commonalities with LTL, for instance, we will encounter temporal operators like like $\bigcirc$ and U and the other ones again.

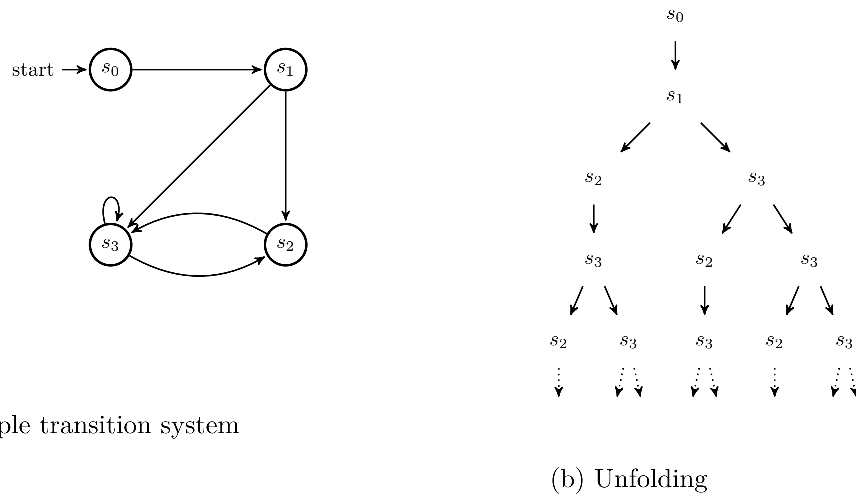
But CTL is also quite different from LTL. LTL treats time as *linear* and at the core the satifaction relation of LTL formula is defined for infinite liniear sequences (traditionally called paths in LTL).

In contrast, CTL is a *branching time* logic. It's not the only one, but a well-known and simple one. Formulas of CTL (and other branching-time logics) are not interpreted on paths, but on *trees*, hence the name (LTL is linear time logic or linear-time temporal logics, CTL does not stand for computation time (temporal) logic, but computation *tree* logic).

In the branching-time view of systems, the behavior of a system is not a linear, resp. a set of linear runs or paths, it's a tree (the computation tree). The tree has points where it branches, these are points where decisions are made one way or the other, caused by non-determinism or input from outside etc.

A transition system can be unfolded into the computation tree of its behavior. That is shown in Figure 1.1b. The nodes in the tree carry more information, like propositions that hold or do not hold at that point, analogous as was done for the paths in the context of LTL.

If one were to check properties of a transition from Figure 1.1a with LTL or another linear-time logic, the system would satisfy the property, if all runs or paths starting from



(a) Simple transition system

(b) Unfolding

Figure 1.1: Transition system and prefix of its infinite computation tree

the initial state would satisfy it. A tree as the one from Figure 1.1b as the unfolding of the transition system of course also contains all those paths. Additionally it contains information about branching, i.e., the points where decisions are made in the execution.

But what difference does that make, if any? Figure 1.2 shows two simple (edge-labelled) transition systems of automata. Their respective behavior (as far as the labels on the edges are concerned and ignoring prefixes) can be written as $a(b+c)$ and $ab+ac$ in regular expression syntax.

(a) $a(b+c)$

(b) $ab+ac$

Figure 1.2: Branching

In a linear picture, both transition system show two (complete) executions, namely $\{ab, ac\}$. Ignoring details like that LTL works about infinite paths (and that our transition systems were primarily state labelled, it means that both systems are equivalent as far as LTL is concerned. Similary, the regular expressions $ab+ac$ and $a(b+c)$ describe the same regular languages. Regular expressions represent are *word* languages, not *tree* languages...

Apart from that fact that the two automata have the same traces, not many would spontenously say that the machine from 1.2a is the "same" (equivalent, isomorphic...) as the one from Figure 1.2b.

In both system, there is a choice, which ultimately leads to ab or ac in one linear run, but the difference is where resp. when the choice is made, in the initial state, or in the middle, after having done a. This kind of information is present in the branching view but absent in the linear one.

That information about branching can be quite crucial. Ultimately, the transition systems represent often *reactive* systems in our context, i.e., systems, parallel or otherwise, that interact with each other and/or the environment. The labels, in that context, play the role of characterizing the interaction. Often, the interactions are classified in *input* and *output*. So instead of uninterpreted letters like a, b, c ... as in the example, one has notations representing input and output. One notation one often finds is writing $a?$ for input (like reading from channel a, obtaining input from port a, reading a value from a variable a ...) and $a!$ for corresponding output.

The issue is somtimes illustrate by the example of a vending machine for coffee and tea. The interaction with the environment, i.e. the customer, is that the customer drops a coin, then makes a choice by pressing a button for either coffe or tea, and then the vending machine serves a hot beverage. From the perspective of the machine, receiving the coin counts as input as well as pressing the coffee or tea button. Using names more suggestive than a, b, and c, two possible realizations of a trivial vernding maching are shown in Figure 1.3. The serving of the hot drink could be seen as out output, but it's not modelled in the simple example. One obviously could add additional fitting output-transitions for the serve steps at the end.

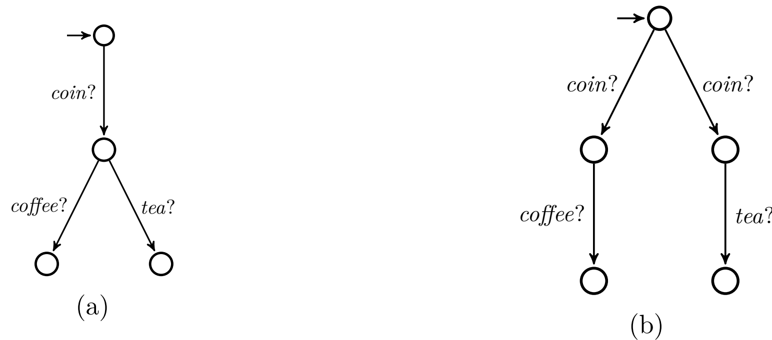


Figure 1.3: Coffee and tea vending machine

The two versions of the vending machine feel drastically different. The first from Figure 1.3a is probably the version the customer would expect: throwing in the the coin first, and then, with the second interaction, pressing either the button for coffee or for tea makes the choice. In the second representation of Figure 1.3b, inputting the coin makes the decision already, thought the customer has no word in it. That's an example of *internal* non-determinism: the decision between coffee and tea is done internally by the transition system. After the decision is made the customer can interact by either only pressing the coffee button or else the tea button. The alternative transition is not possible, it's not *enabled*. One could see it that the correspond button is blocked. So in the left-branch, the user can only "choose" coffee, and is afterwards served coffee (not modelled by an output transition).

In that sense the two transition systems behave pretty differently. The first one could be seen as deterministic, at least choices are made externally. In the field of control-theory, one could also see the first as more controllable, the second less so.

From the perspecitive of LTL or linear time logics, there is no difference between the two versions: they satisfy the same formulas. So one can specify the user interaction or

user interface of such a vending machine as detailed as one wants, still a version exhibing behavior like the one from Figure 1.3b, making choices on its own satisfies a specification perfectly. So even if the version of Figure 1.3a is the intended one, the other one is correct as well, and perhaps can be verified to be correct. Though customers will neither like the machine much nor the argument that's it's mathematically correct. . .

In CTL as branching logic, one will distinguish between the two behaviors. Let's have first another look at how satisfaction is defined in LTL. At the core, the satisfaction relation is define between paths and LTL formulas. But one "lifts" that core definition to define when a transition system resp. a *state* satisfies a formula, by saying, that a state satisfies a formula if *all paths starting in the state* satisfies the formula. A formula φ talking about states is thereby implicitly universally quantified. So one could write more explicitly.

$$s \models \forall\varphi \quad \text{iff} \quad \pi \models \varphi \quad \text{for all paths } \pi \text{ starting in } s \tag{1.1}$$

Since all LTL formulas, when interpreted over a state, are universally quantified (and quantified at the beginning) there is no need to mention the $\forall$ and it's left implicit. In summary, LTL formulas speaking about *states* can be seen as of a prefix-quantified form $\forall\varphi$, where φ does not contain further quantifiers and speak about a *path*.

CTL genealizes that, in that it also uses existential quantification and also allows quantification inside the formula. As we will see in the syntax, there will be a distinction also between *path formulas* and *state formulas*, the latter ones are those starting with a quantifier, specifying that all paths starting in the state satisfies a path formula or that there exists a path with the specified path property.

With being able to have a quantifier not just at the beginning of the formula, but allows nested inside some formula, talking about a situation that occurs after steps, allows to talk about when choices are made.

For instance, without giving the actually formula here, one could specify, that for all states reachable after a coin-transition, there *exists* a coffee-choice transition *and* there *exits* a tee-choice transition. That rules out that the coffee-choice or tea-choice is blocked after inserting the coin in the vendor example

LTL vs CTL? With the exposition so far, it seems plausible, maybe even almost obvious, that CTL is more expressive than LTL. CTL seem to have more formulas it seems so it should be more expressive. That's actually not the case. A The way the syntax of CTL is defined restricts slighthy how path formulas can be combined. As a consequence, CTL formulas are not actually a superset of LTL formlas.

As a further consequence, CTL is *not more expressive* than LTL, both logics are incomparable concerning expressiveness. Examples showing that may require some thinking over, it might not obvious even after we have clarified the syntactic restictions in CTL. Later, we will also talk about CTL*, which lifts those restrictions and that logic is more expressive than both LTL and CTL.

1.2 Syntax and semantics of CTL

Let's start by fixing the syntax. As for LTL, we assume the non-temporal part of the logic to be propositional, with $p, q \ldots$ from a set of propositional atoms P

Definition 1.2.1 (CTL syntax)**.** The syntax of CTL is given by the grammar from Table 1.1, where p represents propositional atoms from p.

$$\begin{array}{lcll} \Phi & ::= & \top \mid p \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi & \text{state formulas} \\ \varphi & ::= & \bigcirc\Phi \mid \Phi_1 \; U \; \Phi_2 & \text{path formulas} \end{array}$$

Table 1.1: CTL syntax

Formulas Φ are called *state* formulas and φ are *path* formulas. Both forms are strictly separate and defined mutually recursive.

The shown syntax is rather restricted, in particular it contains only $\bigcirc$ and U. As in LTL, those two operators are enough to express all the other familiar ones, like $\Diamond$ and $\Box$. Path formulas refer to paths, and the interpretation of "next" and "until", i.e. of $\bigcirc$ and U corresponds to their LTL counter-part. The exact definition will follow later.

In the grammar we also get a feeling for the aforementioned restriction of CTL (compared to CTL*). A path formula consists of a top-level temporal operator, and one or two *state* formulas as immediate sub-formulas. In other words, the grammar does not allow to have apply a temporal operator to a path-formula. Similarly, the quantifiers $\forall$ and $\exists$ for state formulas are followed by a path-formula, which means, a temporal operator inside a formula is immediately preceded by one of the quantifiers. This restriction of the syntax leads to a subtle restriction in expressiveness, which renders CTL as not more expressive than LTL, but incomparable in expressiveness.

As a not so important side remark: because of that restriction, both $\forall$ and $\exists$ is covered by state formulas, referring to all paths starts in a given state, resp. to some path starting there. Conventionally, universal and existental quantification are each other's duals, However, CTL does not allow to exploit that duality, because there is is no negation on path formulas, it's only supported for state formulas.

Let's look a few examples, often they will have some counter-part in LTL.

We have seen in LTL, how to express that some property holds infinitely often for a path. One can express the analogous property in CTL as a state formula that requires that a property holds infinitely often on *all* paths. In the following example, for instance that a traffic light is green infinitely often.

Example 1.2.2 (∞). The property, for instance of a traffic line that it is "infinitely often green", and that for all possible behaviors, is captured by the state formula $\forall\Box\forall\Diamond green$. $\Box$

Example 1.2.3 (Mutex). The safety property that a system never is in a state where two processes are in a critical section at the same time can be captured by the state formula

$$\forall\Box(\neg crit_1 \vee \neg crit_2) \ . \tag{1.2}$$

The example assumes two processes and $crit_1$ holds if process 1 is in the critical section, analogously for $crit_2$ for the other process. □

For parallel or concurrenct systems, an important general is *progress*. In the treatment of LTL, we did not explictly point to a LTL formula say "this is progress". It depends always a bit on what progress means. Generally it means that the system or one participant does not get stuck. In that sense, it's a liveness property stating that eventually something good happens in that the system continues. Mutual exclusion protocols are often given in an "repetetive" way: the processes don't just try to enter a critical section once, but a process, having entered its critical section and after (hopefully) leaving it again, it tries to enter again, in a big loop, and so for all processes. In such a set-up, progress for a process could mean that means it is in it's critical section infinitely often. If also mutual exclusion from Example 1.2.3 hold, it means the two processes enter and exits the critical section infinitely often. With mutual exclusion, it may be that both processes enter the criticial section at the same time, and then get stuck or deadlock, in which case one should not call the behavior progressing... The required behavior is also connected to *fairness*.

Example 1.2.4 (Progress). The safety property that a system never is in a state where two processes are in a critical section at the same time can be captured by the state formula

$$(\forall\Box\forall\Diamond crit_1) \wedge (\forall\Box\forall\Diamond crit_2). \tag{1.3}$$

□

All the examples could easily have been expressed in LTL as well. In particular the example with infinitely many occurrences from Example 1.2.2 and the variation thereof, the progress Example 1.2.4. The CTL formula $\forall\Box\forall\Diamond p$ is of course no LTL formula. However, it can equivalently expressed by

$$\forall\Box\Diamond p \ .$$

And that's how LTL formulas are interpreted on states. Not that this formula is *not* an CTL formula, it violates the discussed syntactic restrictions (but, as said, $\forall\Box\forall\Diamond p$ expresses the same and is syntactically correct CTL).

What makes that possible is that the formula is prefixed by $\forall$-quantifiers, not just two, but also the $\Box$ is a quantification over all points in a path.

For a formula like

$$\exists\Box\exists\Diamond p \ , \tag{1.4}$$

such a rearrangement does not work. The quantifiers involved are "exists-forall-exists-exists". So there is a quantifier change and the formula cannot be rearranged to

$$\exists\Box\Diamond p \tag{1.5}$$

The latter one is not LTL, at least not the standard variant that is interpreted in states over all paths, but it expresses the property that p holds infinitely often on some path. The CTL formula from equation (1.4) expresses *something different*.

As it turns out, infinite occurrence on some path, captured by equation (1.5), is an example of a property not expressible by CTL. The formula from equation (1.4) seem almost to express that, but it's not the same, in a subtle manner. One can show that the two formulations from equations (1.4) and (1.5) are not equivalent. That's done by a concrete example which satisfies equation (1.4) but has no path with infinitely many occurrences of p. We will show the example later, but you may try to figure out one example yourself.

The following is an example of a response property, like the one we had for LTL.

Example 1.2.5 (Response). For all behaviors, each request sooner or later will entail a response:

$$\forall\Box(\mathit{request} \rightarrow \forall\Diamond \mathit{response}). \tag{1.6}$$

□

Example 1.2.6 (Restart). Let *start* characterize the start state or start states of a system. The the following CTL formula expresses that the system never gets ultimately stuck in some dead-end, in that it's always possible to reach back the inital state(s):

$$\forall\Box\exists\Diamond \mathit{start}. \tag{1.7}$$

□

The last property is one which is intrinsically *branching*. It cannot be expressed in LTL.

1.2.1 Satisfaction relation

Now to the semantics, i.e., the satisfaction relation. Generally, the definition should present not much surprises. Since the syntax distinguishes between path an state formulas, $\models$ is defined for states as well as for paths, and both definitions are mutually recursive.Given a transition system, let's refer to all paths starting in a state s by $\mathit{paths}(s)$.

Definition 1.2.7 (Satifaction relation).

$$\begin{array}{lll} s & \models p & \textit{iff} \quad p \in V(s) \\ s & \models \neg\Phi & \textit{iff} \quad \text{not } s \models \Phi \\ s & \models \Phi_1 \wedge \Phi_2 & \textit{iff} \quad s \models \Phi_1 \text{ and } s \models \Phi_2 \\ s & \models \exists\varphi & \textit{iff} \quad \pi \models \varphi \text{ for some } \pi \in \mathit{paths}(s) \\ s & \models \forall\varphi & \textit{iff} \quad \pi \models \varphi \text{ for all } \pi \in \mathit{paths}(s) \end{array} \tag{1.8}$$

$$\begin{array}{lll} \pi & \models \bigcirc\Phi & \textit{iff} \quad \pi^1 \models \Phi \\ \pi & \models \Phi_1 \; U \; \Phi_2 & \textit{iff} \quad \exists j \geq 0.(\pi^j \models \Phi_2 \text{ and } \forall 0 \leq k < j.\pi^k \models \Phi_1) \end{array} \tag{1.9}$$

1.3 CTL model checking

This section covers algorithm(s) for model checking CTL. The presented technique is rather different from the one for LTL, in particular it will not conceptually be based on refutation.

CTL model checking is also often discussed in connection with symbolic, BDD-based model checking. We will do that afterwards, in this section we discuss the conceptual algorithm.

1.3.1 Existential normal forms

It is often advantagous to focus on some core set of operators. We did similarly for LTL, using a restricted set of non-temporal operators and only $\bigcirc$ and U for the temporal aspects. We do the same here. The advantage of such a focus is there are less algorithms to explain and understand. The remaing ones are syntactic sugar and can be model-checked indirectly thereby.

In practive, that may not be the best course. One sure has less model checking routines to implement, but as far as efficiency is concerned, one may be better off to take the effort and implement specific routings for operators left out from the core language.

One can choose the operators for the core calculus quite differently. One choice could be use negation $\neg$ only on proposional atoms, but not on compound operators. That's called *positive normal form*. We will use a different selection, one that allows negation on compound formulas. But we restrict ourselves on considering only $\exists$ as path quantifiers.

Definition 1.3.1 (Existential normal form)**.** CTL state formilas in *existential normal form* are given by the following grammar:

$$\begin{array}{rcl} \Phi & ::= & \top \mid p \mid \Phi_1 \land \Phi_2 \mid \neg\Phi \\ & \mid & \exists \bigcirc \Phi \mid \exists \Phi_1 \; U \; \Phi_2 \mid \exists \Box \Phi \mid \end{array} \tag{1.10}$$

Remember that in LTL, $\bigcirc$ and U where are complete set of operators, as far as the temporal part is concerned. Here, we have to include an additional operator, $\Box$. You may reflect on why it was not needed in LTL but necessary now.

We will *not* make use of the positive normal forms, so we don't cover the concept. But positive normal forms exists also for other logics, not just CTL, so we can reflect a bit on which criteria one should base a selection of operators or a normal form. We mentioned already that presentation-wise, it is advantagous to restrict to a reduced set, and that for implementation, a too reduced selection is at least double-edged.

In many contexts, not having to deal with negation for compound formulas is advantagous; that's why positive normal form are not uncommon. In many constructions, complementation is tricky. It may lead to a blow-up in the representation or is hard to do, or both. Remember in that context, that for LTL model checking, the construction was not building an automaton $\mathcal{A}_\varphi$ and then complementing it. It would be possible, but

unwise, since the construction is hard and at least the original construction by Büchi is double-exponential (there are better ones know in the meantime). That's why we avoided automaton complementation by constructing $\mathcal{A}_{\neg\varphi}$ directly. Of course, we did not make use of a positive normal form of LTL, but for the construction based on maximally consistent sets of formulas etc. negation posed no problems.

When doing model checking for CTL, when using a normal form that allows negation, we need to keep an eye on whether will not be costly in term of efficiency. As indicated, CTL model checking is often done what is called "symbolically" and based on BDDs. That is a particular "Boolean" representation of the transition system and information that is tracked by the model-checking implementaint. BDD stands for specific binary trees, and the particular form used (as ROBDDs), it is also a kind of normal form (for boolean formulas). The model checking algo with have to deal with negated, compound formulas, and when using BDDs, negation needs to be done efficiently for that representation (and for other formulas as well). Luckily, negation will be particularly efficient ...

1.3.2 Bottom-up treatment of compound formulas and sat-sets

Now we have agreed on the syntax (ENF), now how to do the model checking? The ENF contains only state-formulas, not path formulas (resp. the path formulas are implicit in the formulation of the grammar). The algorithm will have to check thus statisfaction wrt. state formulas only.

To check that $T \models \Phi$, the algorithm will proceed **bottom-up** through the formula Φ, i.e., it starts by treating the leaves of the formula, and then proceeds bottom-up until it has to finished treating the whole formula.

It does so by working with the interpretation of the (sub)-formulas as the set of states that satisfies is. So when we said, the alorithms "treats" a sub-formula Φ' of Φ, means it calculates the set $\{s \in S \mid s \models \Phi'\}$. A crucial part *symbolic* model checking is that this list is not treated as an enumeration of indiviual states, i.e., the approach does not solve $s \models^{?} \Phi'$ for each individual state. That would be *explicit state* model checking. They key for a **symbolic** treatment for sets $sat(\Phi')$ is capture them "formulaically". which will here be *propositionally*. It's likewise important that this symbolic representation is compact and can be manipulated efficiently when running the algorithm. Ultimately, the propositional representation will be based on a form graph or tree like representation of boolean functions called binary decision trees (BDDs); more about that later. Let's summarize the approach first

The basic strategy of $T \models \Phi$

1. calculate $sat(\Phi)$ recursively over the structure of Φ
2. $T \models \Phi$ iff $I \subseteq sat(\Phi)$.

As mention earlier, the recursive strategy treats the formulas bottom-up. So when the algorithm treats a compound (sub-)formula, say an until formula $\Phi_1 \; U \; \Phi_2$, the sets $sat(\Phi_1)$ and $sat(\Phi_1)$ have already calculated. They can be treated by the algorithm as *propositions*. The recursive algorithm will contain a case-swich for each syntactive form (in ENF), but

in the bottom-up strategy, in each case, the ask will be to treat the corresping constructor applied on propositions. For instance, the U-case requires a treatment for $A_1 \ U \ A_2$, with A_1 and A_2 propositions representing the results of determining the *sat*-sets of the immediate subformulas Φ_1 and Φ_2.

Example 1.3.2. Consider the transition system of Figure 1.4 and the CTL property from equation (1.11). The "names" of the states are considered also as propositions, for instance, the initial state is called *born* which is interpreted that in this state, the proposition *born* holds, but in the other states not.

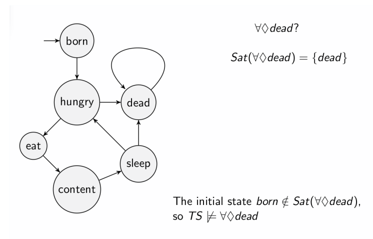


Figure 1.4: Transition system

$$\forall \Diamond dead. \tag{1.11}$$

□

Example 1.3.3. Consider again transition system of Figure 1.5. This time, let's look at the CTL property from equation (1.11):

$$\exists \bigcirc hungry \wedge \exists(eat \ U \ \neg dead). \tag{1.12}$$

Figure 1.5 shows the syntax tree of the formula. Additionally, the *sat*-sets are indicated for most nodes.

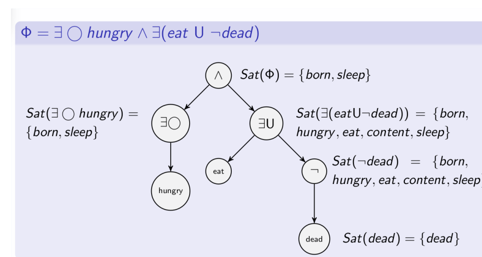


Figure 1.5: Bottom-up calculation of *sat*-sets

Since all initial states —there is only one— of the transition system are contained in the *sat*-set of the formula, the transition system satisfies Φ.

As mentioned, the model checking algorithm calculates the *sat*-sets bottom-up in such syntax trees. The figure does not illustrates that calculation, only its results.

To get a feeling of how it works, consider, for instance, the node for $\bigcirc hungry$ and its child node *hungy*. That *sat*-set for the proposition *hungy* is the singleton set $\{hungy\}$ of states from the transition system from Figure 1.4. The set for the parent node $\bigcirc hungry$ is calculated from the *sat*-set $\{hungy\}$ for the child-node by following the edges in the transition system **backwards**; in this particular example, there is only transition to follow backwards, the edge from *born* to *hungy*. So the algorithm will treat the $\bigcirc A$-case for a proposition A, given $sat(A)$, by calculating $pre(sat(A))$

The treatment for U is similar insofar that it involves following the transitions backwards. Instead of a single-step calulation of predecessors, it will involve an iterated calculation of predecessor set, in a iterated backward-exploration of the transition system. Such a calculuation can be understood as the calculation of a *fixpoint*. □

```
1  input:  finite transition system T and Φ
2  output: T ⊨ Φ
3  ————————————————————————————————————————
4  for all i ≤| Φ | do
5    for all Ψ ∈ sub(Φ) with | Ψ |= i do
6      compute sat(Ψ) from sat(Ψ′) (* max. genuine Ψ′ ⊊ sat(Ψ) *)
7    od
8  od
9  return I ⊆ sat(Φ)
```

Listing 1.1: Basic algorithm

```
1  input:  finite transition system T and Φ
2  output: T ⊨ Φ
3  ————————————————————————————————————————
4  switch Φ
5       ⊤            => return S
6       p            => return {s ∈ S | p ∈ V(s)}
7       ¬Φ           => S \ sat(Φ)
8       Φ1 ∧ Φ2      => return sat(Φ1) ∩ sat(Φ2)
9       ∃ ◯ Φ        => return pre(sat(Φ))
10      ∃(Φ1 U Φ2)  => ``lfp for ∃ U''
11      ∃□Φ          => ``gfp for ∃□''
```

Listing 1.2: Recursive algorithm

1.3.3 Characterization of *sat* & fixpoint calculations

Next we have to fill in the blanks in the algorithmic skeleton of Listing 1.2. We will do so by *characterizing* the different cases for the *sat*-calculation, i.e. the different cases of the switch-construct. Some of the cases don't require much further explanation, as they are basically covered in the code of Listing 1.2.

Not surprisingly, the trickiest cases are those for U and $\Box$. The one for $\exists\bigcirc$ is quite straightforward. One simply needs to explore "backwards", calculating *pre*. Later though, we will express that slightly differently. But for now, we will focus on the last two cases.

In Listing 1.2, it's mentioned that those cases are treated by calculating a fixpoints, th greatest fixpoint resp. the least fixpoint, gfp and lfp.

Fixpoint calculation for $\exists\Diamond$

That requires some elaboration, I assume. The following illustrates the fixpoint idea not for $\exists(A_1\ U\ A_2)$, but for $\exists\Diamond A$, which is a special case of that, with $A_1 = \top$. That's slightly easier to present, as the formulas has only one subformula not two. It's probably also easier to understand than the $\exists\Box A$ case, which is why we do the exposition about fixpoint calculation using $\exists\Diamond$.

Before we get into the fix-point business, we can think how to solve the model-checking problem for

$$\exists\Diamond A$$

with A given. It's very easy, actually. $sat(\exists\Diamond A)$ correspond to all states from which a state in A can be *reached*. That can be calculated in a natural way by a **backward** exploration starting at A. Forward or backward, no matter, a graph can be explored in different ways. Depth-first for instance, or breadth first, or something more fancy. One can also leave the strategy open, resp. follow the most unspecified stategy possible, following edges randomly, here pre-edges. See Listing 1.3.

```
1 T := sat(B);
2 while pre(T) \ T ≠ ∅ do
3   let s ∈ pre(T) \ T ;
4   T := T ∪ {s};
5 od;
6 return T;
```

Listing 1.3: Backward exploration for $\exists\Diamond A$

The algo starts with A, resp. the corresponding *sat*-set. In each round, following some edge backwards it picks a state that has not been explore yet, and adds that state to the set T of explored ones. If no such state exists, it stops. At that point T contains the desired *sat*-set.

Let's remark two or three things, maybe obvious ones. The first is, that T is properly enlarged in each round. Because if no new state can be fround be the exploration, the algorithm stops. This, secondly, implies that the algorithm *terminates*, because we assume finite-state systems. That's also a triviality: one can for sure do graph searches or graph explorations on finite graphs. Thirdly, the sketched exploration adds one state in each round, i.e., the iteration treats states *individually*. That's not in our interest of doing *symbolic* model checking.

The last point we will address a bit later, let's continue with a discussion of what that pretty simple algorithm has to do with **fixpoints.**

For the sake of this discussion, let's massage the code from Listing 1.3 a bit. As said, in each round the set T is increased by one state. Let's capture that effect in one operator, say F.

To do so, call *pick* a function that when applied to a non-empty set, gives back a random element from that set; it's undefined on the empty set. $pick_\emptyset$ the function that, when applied to a non-empty set, gives back a singleton-set with one element randomly chosen, but when applied to the empty set, gives back the empty set. With this function, we can equivalently write for the loop-body of Listing 1.3 as $T := T \cup \{pick(pre(T))\}$. There

is always a state s that can be picked and *pick* is well-defined, as that's checked in the loop-condition. We can thus write for the loop body

$$T := T \cup pick_{\emptyset} T \ .$$

The loop condition $T \setminus pre(S) \neq \emptyset$ can be equivalently expressed as

$$T \supset T \cup \{pick(pre(T))\} \quad \text{or} \quad T \neq T \cup \{pick(pre(T))\} \ .$$

```
1 T := sat(A);
2 while   T ≠ T ∪ {pick(pre(T))}
3   T := T ∪ pick(pre(T));
4 od;
5 return T;
```

Listing 1.4: Backward exploration for $\exists\Diamond A$

At the exit of the loop, we have therefore

$$T = T \cup \{pick(pre(T))\} \ . \tag{1.13}$$

i.e., each iteration step i increases the current set T_i properly to $T_{i+1} = \{pick(pre(T))\}$, adding one randomly picked state not yet explored. That's done until no new such states are found, and that is captured by the equality from equation (1.13).

IF we write F for the function $F(X) = pick_{\emptyset} pre(X) \cup X$, the different incarnations T_0, T_1 etc. are given as $T_0 = sat(A)$, $T_1 = F(T_0)$, $T_2 = F(T_1) = F^2(T_0)$, etc. i.e.,

$$\begin{aligned} T_0 &= A \\ T_{j+1} &= F(T_j) \qquad \text{where} \quad F(X) = pick_{\emptyset}(pre(X)) \cup X \end{aligned} \tag{1.14}$$

The T_i's from Unlike the algorithm code, the definition from equations (1.14) correspond to the value of T in the different iteration rounds, though the formulation in equation (1.14), unlike the algorithm, does not specify when to stop, by for instance saying that j has a value from $0 \ldots k$.

If the loop stops after, say, k iterations, it means with the loop's exit condition from equation (1.13) that $T_{k+1} = F(T_k) = T$. Additionally, it's immedeate to see that not only $T_k = T_{k+1}$, but also $T_{k+1} = T_{k+2} = T_{k+3}$. So, once an application of F to the current value of set T does not increase any more (and the loop stops), no further applications would increase it later, perhaps after further iterations without any increase.

$$A = T_0 \subset T_1 \subset T_2 \subset \ldots \subset T_k = T_{k+1} = T_{k+2} \ldots \tag{1.15}$$

It's also said that the iteration or the chain of T_i's *stabilizes* at the point where $T_{k+1} = T_k$ for the first time, and it's characteristic that, once stabilized, it will remain at that point forever. Thus it makes perfect sense to make stabilization the exit condition for the iteration.

For all elements T_i in the chain, we know that

$$A \subseteq T_i, \quad \text{and} \quad T_i \subseteq T_{j+1} \ . \tag{1.16}$$

Equation (1.15) is more explicit about that for indices lower than k, it's $T_i \subset T_{j+1}$, for those higher it's $T_i \subset T_{j+1}$, but the $\subseteq$-relations from equation (1.16) are stating something correct, nonetheless. Actually, without knowing the concrete k where the iteration stabilzise, that seems to be best we can say about how the T_i's hang together. Actually, also the code of the algorithm does not operate with a concrete k: it's based on a while-loop, iterating until stabilization not on a for-loop.[1]

The sets T_i are representing the status of the interation in the different rounds, but based on the observation from equation 1.16, we could more abstractly (without mentioning any looping construct) require

> **Goal (a)**: Find me a set T such that (a) it contains A and such that (b) $F(T)$ does not make it larger.

There will later be a refinement of that goal, but anyway, the current version can be captured by the following two in-equations, inequations in the sense of $\supseteq$

$$\begin{array}{rcll} T & \supseteq & A & \\ T & \supseteq & F(T) & \quad \text{where} \quad F(X) = \mathit{pick}_{\emptyset}(\mathit{pre}(X)) \cup X \end{array} \tag{1.17}$$

That can be seen as (in-)equation system, where T is the variable over which the equation system is formulated. Actually, it's a *recursive* equation system; at least the second (in-)equation is recursive in T. Solving it corresponds to the goal stated just above.

That's all fine and good, but what has it to do with fix-points? Well, at the stabilization point, say T_k, we have $T_{k_i} = F(T_k)$, and that means T_k, calculated iteratively **solves it.**

We can also combine the two in-equations from equation (1.18) into a single line.

$$T \quad \supseteq \quad F'(T) \qquad \text{where} \quad F'(X) = \mathit{pick}_{\emptyset}(\mathit{pre}(X)) \cup X \cup A \tag{1.18}$$

Now we have one single, recursive (in).equation, and we know that T_k solves it, i.e., $T_k \supseteq F'(T_k)$, actually, as explained, $T_k = F'(T_k)$. In other words

> T_k is a fixpoint of F', i.e., it solves $T = F'(T)$.

I.e., we have found a solution not only of equation (1.18) using $\supseteq$, but also of the corresponding constraint using equality (and using X instead of T, to stress that it's a variable of the equation):

$$X \quad = \quad F'(X) \tag{1.19}$$

[1] For-loop in the sense of an loop with a fixed number of iterations, like `for i = 0 to k`, not in the sense of loops using the `for`-keyword in Java.

After all this massaging, we have cast **goal (1)** into the form of a **fixpoint equation**, resp. an "fixpoing inequation" in equation (1.18). The technical term for the latter is *pre-fixpoint*; there are also post-fixpoints, where $\supseteq$ is used the other way around, but here, for the shown construction for $\exists\Diamond$, we are into pre-fixpoints. Actually, for the dual of "eventually" temporal property, the $\Box$, it works the other way around and operates a post-fixpoint (and a fixpoint).

We said, that the T_k calculated by the algo is both a pre-fixpoint as well as a fix-point of F', i.e, it solves both equations (1.19) and (1.18). Obviously, each fix-point is at the same time a pre-fixpoint as well (and also a post-fixpoint, for that matter). But the fixpoint formulation with $=$ and the pre-fixpoint formulation with $\supseteq$ are not the same constraint: the former imposes stricter requirements on the set of solutions, in other words, there are pre-fixpoints which are not also fixpoints.

Actually, that fact us not really important. It's true that equations (1.19) and (1.18) are not expressing the same constraints and one has more solutions than the other, but they *do have the same solution(s) where it matters.*

That has to do with one missing piece of the story. So far, we have covered only what we called **goal (a)**, which we have turned into a (pre-)fixpoint requirement, and we said T_k solves both formulation. But indeed, T_k not only satisfies $T_k = F'(T_k)$ and $T_k \supseteq F'(T_k)$, in the chain from equation (1.15), k is the *first* point where that happens. In other words, T_k is the *smallest* set from the chain that solves the fixpoint resp. the pre-fixpoint formulation.

Actually, it's not just the smallest set in the chain with that property, one can prove that it's generally the smallest fixpoint as well as the smallest pre-fixpoint (not just in the particular chain-construction). So, while the pref-fixpoint formulation has more solution, as far as the smallest solutions are concerned, and that's what we are after, both formulations are indeed equivalent. Additionally, one can prove that there the smallest solution is *unique*. Thus, it's the (unique) smallest solutution for **goal (a)** that we are after as **the** set $sat(\exists\Diamond)$.

Goal (b): Find the **smallest** set T satisfying **goal (a)**.

That may seem a long and winded explanation for something quite simple, namely how do a (backward and random) graph exploration to determine $sat(A)$. That may be so, but also semantics resp. algorithms for $\exists\ U$ and $\exists\Box$ can be explained and justifiedsimilarly (the one for $\exists\Box$ calclating greatest (post-)fixpoints, gfp's). Also other temporal operators left out of the restricted ENF syntax are given as fixpoints (some as least (pre-)fixpoints, some as greates (post-)fixpoints). With the slightly longish explations in case of the simpler $\exists\Diamond$ setting, boiling down to a very straightforward graph exploration, we can keep the presentation of the slighty more complex cases short. But before that, there is another issue to address. With all the talk about fixpoints and how to solve them, we lost sight of another aspect of CTL model checking, namely that it's a well-known example of *symbolic* model checking.

Explicit state vs. symbolic model checking

The skeleton recursive model checking algorithm from Listing 1.2 is working with *sets* of states, the $sat(lstateform)$-sets. As explained, the two last and most complicated cases case switch are doing fixpoint iteration, calculating some gfp or lfp.

If we look at the way it has been explained,the basic step picks one individual state s in each round, adding it to the set T of explored states.

That's the way, explicit-state model checking works. In the spirit of symbolic model checking, it's better to explore the graph by adding whole sets of states in each iterative step. That can easily be written down. Instead of the iteration from equation (1.14), using $pick_\emptyset(pre(X))$, we simply add *all* states of $pre(X)$ in one swoop:

$$\begin{array}{rcl} T_0 & = & A \\ T_{j+1} & = & F(T_j) \qquad \text{where} \quad F(X) = pre(X) \cup X \end{array} \tag{1.20}$$

Likeise, the code from Listing 1.4 is readily adapted to the one of Listing 1.5.

```
1 T := sat(B);
2 while T ≠ T ∪ pre(T) do
3   T := T ∪ pre(T)
4 od;
5 return T;
```

Listing 1.5: "Breadth first" backward exploration for $\exists\Diamond A$

The latter seems even even simpler than the one from before, so why did we not start the fixpoint explanation with this one? Indeed we could have done that, everything would have worked analogously. Nonetheess, we started the exposition with the explicit-state version for two reasons. One is, showing both versions may rub in the difference between symbolic model checking on the one hand and explicit-state model checking on the other. Secondly, the graph explication here can be seen as a specific explication strategy, namely breadth-first exploration. The previous one is a random exploration. The random exploration can also be seen as the most "general" strategy, a strategy that covers all specific exploration strategies, like breadth-first, depth-first and other traversal strategies. If one can convince oneselve that even this random strategies has good properties (like doing the job, and terminating), then all other strategies, being included in the non-deterministic one, also have those properties (and one does not have to re-consider the correctness or termination question for all possible more concrete strategies). Of course, in practise, some strategies and heuristic may be more efficient than others, but that's an optimization question, not one whether the strategy works at all.

Talking about optimization: who actually said that bread-first exploration is better than some alternatives? Indeed, explicit state model checking, say for LTL, is often based on depth-first search, the memory footprint of breadth-first search is mostly not managable. Also with the code from Listing 1.5, if the step

$$T := T \cup pre(T) \tag{1.21}$$

is nothing else but an innler loop through $\mathit{pre}(T)$, like

$$\textbf{forall } s \in \mathit{pre}(T).\textbf{do } T := T \cup \{s\}$$

we have not gained much except that it's not breadth-first instead of a random exploration, and that's of dubious value in itself. To make sense, the step from equation (1.21) but be done efficiently, and that implies, not by interating through the predecessors individually.

Indeed, it's *crucial* for symbolic model checking not just "have" sets returned as result, as sketched in the code of Listing 1.1, but all steps of the algorithms should be calculated more or less efficently on those sets, and more often a step is to be executed, the higher the pay-off if that steps is efficent. Certainly, the exploration steps for the temporal formulas are done repeately, so in particular the calculation of $\mathit{pre}(T)$ should be effcient. Likewise, checking for equality is needed for finding out f the desired has been reached, and union $T \cup \mathit{pre}(T)$ will be needed in the inner loops of the algorithm. In the next section we cover how that can be achieved. It will be based on a propositional encoding, conceptually working with (a particular representation of) boolean functions, known as binary decision diagrams, BDDs. The BDDs will be a particular form of binary DAGs, but to connect them to the algorithms, it's more conventent not explain the encoding *directly* in terms of the BDDS, but rather in propositional formulas.

Now that we have explained in some depth the nature of the exploration for $\exists\Diamond$ as solving a (pre-)fixpoint inequation in an iterative manner, we give the correponding characterizations for $\exists\ U$ and $\exists\Box$. The until-case is a slight generalization of the "eventually" case. The case for $\exists\Box$ is dual insofar it is defined as *greatest* fixpoint, and consequently, the iteration does not work by enlarging a set until stabilization, but dually be making it smaller step by step, until stabilization.

1. $\mathit{sat}(\top) = S$.
2. $\mathit{sat}(p) = \{s \in S \mid p \in V(s),\ \text{for any } p \in P\}$.
3. $\mathit{sat}(\Phi_1 \wedge \Phi_2) = \mathit{sat}(\Phi_1) \cap \mathit{sat}(\Phi_2)$.
4. $\mathit{sat}(\neg\Phi) = S \setminus \mathit{sat}(\Phi)$.
5. $\mathit{sat}(\exists \bigcirc \Phi) = \{s \in S \mid \exists s'.s \to s' \wedge s' \in \mathit{sat}(\Phi)\}$
6. $\mathit{sat}(\exists(\Phi_1\ U\ \Phi_2))$ is the *smallest* subset T of S such that
   a) $\mathit{sat}(\Phi_2) \subseteq T$ and
   b) $s \in \mathit{sat}(\Phi_1)$ and $\exists s' \in T.s \to s'$ implies $s \in T$.
7. $\mathit{sat}(\exists\Box\Phi)$ is the *largest* subset T of S such that
   a) $T \subseteq \mathit{sat}(\Phi)$ and
   b) $s \in T$ implies $\exists s' \in T.s' \to s$.

Figure 1.6: Characterization of CTL operators

```
1 T := sat(Φ2);
2 while {s ∈ sat(Φ1) \ T | post(s) ∩ T ≠ ∅} ≠ ∅ do
3   let s ∈ {s ∈ sat(Φ1) \ T | post(s) ∩ T ≠ ∅};
4   T := T ∪ {s};
5 od;
```

```
6  return T;
```

Listing 1.6: Fixpoint calculation for $\exists \Phi_1 \; U \; \Phi_2$

```
1  T := sat(Φ);
2  while {s ∈ T | post(s) ∩ T = ∅} ≠ ∅ do
3      let s ∈ {s ∈ T | post(s) ∩ T = ∅};
4      T := T \ {s};
5  od;
6  return T;
```

Listing 1.7: Fixpoint calculation for $\exists \Box \Phi$

1.4 Symbolic model checking

This section will show how the previous approach can be “encoded” or “represented” in a way that leads to an efficient implementation. The encoding will ultimately “binary”, i.e., by bits. That this is possible should be clear, since we are dealing with a finite problem, the transition system has finitely many states, we are checking propositinal formulas, and also there are finitely many sets if states *sat*, the algorithms uses. With everything being finite, it’s clear that one can represent it by bits. With everything finite, it uses a finite amount of memory when implemented. But there’s more to it than that the problem is finite and when implemented on some computer, everything is ultimately bits and bytes anyway.

We look more systematically at how the nececcary encodings and operations on that can be encoded or represented. And for that we start by introducing a particular form of boolean functions, called switching functions.

1.4.1 Switching functions

The symbolic approach here makes heavy use of boolean functions, representing the transition system and the model checking algorithm operates on boolean function, and the ultimate data structures, the BDDs, are som particular representation for boolean functions. This section here fixes a few definitions and notations needed later.

As for boolean values, we use 0 and 1 (and not $\bot$ and $\bot$, as we did in connection with logics (but it’s notation only anyway). Boolean functions are functions of type $\{0,1\}^n \to \{0,1\}$. For later developments, it’s more handy (and more conventional) to identify the function arguments not by their position, but by name, i.e., th fuctions are seen as of type $Var \to \{0,1\}$, where Var is a finite set of variables, typically $z_1, z_2, \ldots$ etc., or similar. Let’s call those function (boolean) *evaluation functions*, with typical element η, and write $Eval(z_1, \ldots, z_m)$ for evaluation functions over the set $\{z_1, \ldots, z_m\}$, i.e.

$$Eval(Var) = Var \to \{0,1\} \; . \tag{1.22}$$

What we call evaluation function here is basically a fixed-size *bit-vector*, only that it’s addressed by names of variables (not offsets from its start, so to say). For a concrete

such vector with values $b_1 \ldots b_m$, we use the notation $[z_1 = b_1, \ldots, z_m = b_m]$. We may abbreviate that as $[\vec{z} = \vec{b}]$ (or even $\vec{b}$...)

A central role play also a particular form of boolean functions, called *switching functions*. Those are boolean-values function over evaluation functions.

Definition 1.4.1 (Switching function)**.** A *switching function* for $Var = \{z_1, z_2, \ldots, z_m\}$ is a function

$$f : Eval(Var) \to \{0,1\} = (Var \to \{0,1\}) \to \{0,1\} \ . \tag{1.23}$$

The special case $Var = \emptyset$ is allowed. In this case, the switching functions for the empty set are the constants 0 or 1

The terminology of “switching functions” is perhaps a bit peculiar, perhaps peculiar for BDDs or bit-valued functions in the tradition of Shannon. As said, nn evaluation function is nothing else than a bit-vector (where variables are used as name for the different slots) and a switching function is a set of such bit vectors (or a *predicate* over such vectors).

In a way, we have encountered evaluation functions and switching functions already, though we did not use the terminology (as it's not common to speak of switching functions there). For *propositional logic*, formulas were interpreted over the domain of boolean values $\mathbb{B}$, but that's of course the same as the set $\{0,1\}$ of bits. Propositional formulas contain propositional variables, where we used p, q, etc., back then, where here we write z_i and similar for the same thing. A propositional formula φ, containing some variables, is interpreted as the set of all variable assignments to the involved propositional variables, that make the formula true. The standard notation for that is

$$\sigma \models \varphi \ , \tag{1.24}$$

the assignment σ satisfies φ; we also called σ a *model* of φ. Alternatively (but equivalently), we said, the semantics of φ is the set of a variables assignemnts that satisfies it, i.e.

$$[\![\varphi]\!] = \{\sigma \mid \sigma \models \varphi\} \ .$$

The variable assignments σ are nothing else than the evalution functions (here denoted by η) and $[\![\varphi]\!]$ is nothing else than what we call here switching functions.

So, a switching function as defined in equation (1.23) represents the semantics of boolean formula or the solution set of a boolean sat-constraint etc. It's an explicit representation of the solutions, i.e., it's not a formula constructed from a given grammar (with a particular set of operators). The representation is also in the form of bit-vectors, which is promising with respect of efficency. It's also “standardized”: With the variables fixed, each solution set of a formula is represented by *one* switching function, representing *the* semantics of the formula. Syntactic, formulaic representation are not unique. A semantics of a particular formula can be expressed in infinitely many different ways.

Switching functions (perhaps realized with bit-sequences) sounds like a decent route to implement propositional formulas. As fine as it sounds, one thing such a representation is not: *compact*. An array of bit-vectors is basically nothing else than a **truth table** representation.

The BDDs that are central for this section is nothing else than a better, on average more compact representation for boolean functions, based on trees. Compactness is an important criterion for choosing a representation, but it's not the only one. Data is not just stored, it's also accessed, worked with and changed. For determining if a variable assignment satisfies a formula, as in equation (1.24) a tabular representation may be fast, but we need other operations as well, also when using the representations for model checking, here CTL model checking, as in one of the different flavors of the core algorithm from earlier. The operations we need should be at least on average be executable efficiently.

Another positive property when choosing a representation is uniqueness of representation. Those unique representations are sometimes called *canonical forms*. Some also call them /normal forms/ in some fields, though in our context, being a normal form does not imply unqiueness. For instance, conjunctive normal forms and its dual disjunctive normal forms, and others, are not unique. They still are some standard representation, and can be used as basis for a genuinely canonical forms, then e.g. called canonical conjunctive normal form.

Often, having a canonical representation can be a very good thing, especially when none needs to comparing the elements one implements. For instance, if one has a truly canonical representation of propositional formulas, one can check that they are equivalent by checking if their representation is identical. That's typically faster to establish than checking for equivalence, which basically means that the two formulas imply each other.

But working with canical (or normal) forms can also be a "burden". Often one operates on the data, here "formulas" or representations of switching functions. It's not generally the case, that the combination of two canonical forms is again canonical. Also other operations, one needs to perform on the data may not preserve canonicity (or normal-ity).

A common and prominent form of CTL model checking is based on so-called BDDs, *binary decision diagrams*. As tree-like data structure, the representation is generally much more compact than a tabular form of the switching functions. Additional conditions on top of the BDDs make assure that the data structure will be a *canonical* representation of boolean functions. Finally, it turns out that the operations we need to do on the boolean functions can be done pretty painlessly.

We have talked a bit vaguely about "operations" we will need to do for model checking, but which are those? We can consult for instance the simple recursive formulation of the model checking algorithm from Listing 1.2. Model checking works by calculating the $sat(\Phi)$ in a fashion working bottom-up through the formula in question. Since the sets $sat(_)$ of states are encoded propositionally, ("symbolically") (see Section 1.4.2), ultimately by BDDs, the code shows what needs to be calculated on BDDs. Based on the reduced syntax of CTL, and besides the base cases of $\top$ and the propositival variables, it's conjunction, negation, the next operator, and the fixpoint calculations for $\exists\Box$ and $\exists\ U$.

Before we come to some technical definitions, let's wrap up and summarize points about switching functions from Definition 1.4.1. The concept can be seen as a *set* of variable assignments or evaluations (or a set of bit-vectors). It can also be seen as representing propositional formulas or propositional constraintsl. A boolean formula represents a set of valuations as in equation (1.24), but one can also turn it upside down: a switching function represents as set of different but equivalent propositional formulas.

Boolean operators for switching functions

Boolean operators like $\vee$, $\wedge$ and the other construct larger formulas from smaller ones. The operators have their corresponding semantical counter-parts for switching functions. Their definition is straightforward, it's kind of like defining truth tables but using the more fanciful definitions based on functions over functions.

Let's cover the base cases of propositional formulas first, propositional variables and the constants 0 and 1.

Given an evaluation or bit-vector from $\mathit{Var} \to \{0,1\}$, a *projection* on some variable $z \in Var$ pick that variable's value in the evalution. A projection function onto a variable z_i, written proj_{z_i}, is of type $(\mathit{Var} \to \{0,1\}) \to \{0,1\}$, i.e., it's a switching function.

In the view if switching function as a set of solutions to a propositional constraint, a projection function focuses on one variable one. The outcome depends on the value of that variable one, the value of all others play no role. That means, the switching function proj_z conresponds to the atomic boolean formula or constraint z. I.e., $[\![z]\!] = \mathit{proj}_z$. Often one writes just z for the projection. Similarly, for the two constant switching functions, one that maps all variable assignments to 0 and the other all to 1, one simply write 0 and 1.

Let's show, as example for composing switching functions, the definition for disjunction.for *composing* switching functions

Note that the switching functions are defined over a set of variables. When combining two switching functions, one has to make a decision, if one defines it only for functions over the *same* set of variable, or if one is more relaxed and don't insist on that. It's not a big difference either way. If one is strict, one would simply need a way to extend the domain of a function to operate on a larger set of variables. This way, before combining two functions, one would extend both, if needed, to operate on the common, joint set of variables.

The definition is is the more relaxed one and allows to combine switching functions with different domains. But as said, it's not a crucial choice.

Let's consider two switching functions f_1 and f_2, over the respective variable domains

$$\{z_1, \ldots z_n, \ldots, z_m\} \qquad \text{and} \qquad \{z_n, \ldots z_m, \ldots, z_k\}$$

with $0 \leq n \leq m \leq k$. In other words, the two function have the variables $z_n, \ldots, z_m$ in common, the variables $z_1, \ldots, z_{n-1}$ resp $z_{m+1}, \ldots, z_k$ are exclusive for f_1 resp. f_2. The switching function over the combined set $\{z_1, \ldots, z_k\}$ representing the disjunction is given as

$$\begin{aligned}(f_1 \vee f_2)([z_1 = b_1, \ldots, z_k = b_k]) = \\ \max(f_1([z_1 = b_1, \ldots, z_m = b_m]), f_2([z_n = b_n, \ldots, z_k = b_k]))\end{aligned} \tag{1.25}$$

max represents the disjunction of bits (assuming that 1 is larger than 0 ...). Each function f_1 and f_2 takes only the variables in its domain into account, of course. The outcome of f_1 only depends on the values chosen for $z_1, \ldots, z_m$, the other variables are irrelevant. will define the related notion of *essential* variables a little further down below in Definition 1.4.3.

Cofactors and Shannon expansion

Next some other operations on switching function that help explaining how to encode the data structures for the algorithm and also BDDs later. We start with *cofactors* and the related notion of Shannon expansion. Note in passing, the master thesis of Claude Shannon, the inventor or discoverer of information theory is sometimes called "the most important Master thesis ever written"[2] (I did not find the exact origin of that quote), and the thesis was concerned with boolean logics and electric circuits (not yet with what became known as information theory, actually Shannon coined the term).

Back to the issues at hand. When viewed as constraints, the positive resp. negative cofactor of a switching function wrt. a variable simply means setting the variable to 1 resp. to 0. That can be captured as follows.

Definition 1.4.2 (Cofactors)**.** Assume a variable set $Var = \{z, y_1, \ldots, y_m\}$ and let $f : (Var \to \{0,1\}) \to \{0,1\}$ be a switching function over it. The *positive cofactor* of f for variable z, written $f\mid_{z=1}$ is the switching function given by

$$f\mid_{z=1}(b_1, \ldots, b_m) = f(1, b_1, \ldots, b_m) \tag{1.26}$$

The *negative cofactor* of f for z, written $f\mid_{z=0}$ is defined analogously. If f is a switching function for $\{z_1, \ldots, z_k, y_1, \ldots, y_m\}$, then we write $f\mid_{z_1=b_1,\ldots,z_k=b_k}$ for the *iterated cofactor* of f, given by

$$f\mid_{z_1=b_1,\ldots,z_k=b_k} = (\ldots(f\mid_{z=b_1})\ldots)\mid_{z_k=b_k} \tag{1.27}$$

Switching functions are defined over a given set of variables, one could call it their domain. It's similar to boolean formulas. Often, for formulas, the set of variables is not explicitly given, it's just all for variables occurring in the formula. Sometimes it's useful to make it more explicit also there, for instance, considering a formula like $x_1 \wedge x_2$ to be a formula over, say x_1, x_2, and x_3, even though x_3 is not mentioned. It just means, to be true, x_1 and x_2 have to be true, but the value of x_3 is arbitrary, it does not affect the outcome.

If course, even if a variable is mentioned in a formula, it does not mean its value has an influence on the outcome. For instance, taking the proposition $x_1 \wedge x_2 \wedge (x_3 \vee \neg x_3)$, the variable x_3 now occurs, but its value is still inessential.

Anyway, whether a value for a variables influences the outcome is the criteron for being essential. That can use used to define essential variables for switching functions.

Definition 1.4.3 (Essential variable)**.** A variable z is *essential* for a switching function, if

$$f\mid_{z=1} \neq f\mid_{z=0} \ . \tag{1.28}$$

Next the important concept of Shannon expansion. Actually it's also known as Boolean expansion, i.e., it's approximately a century older than Shannon's works.

[2]See `https://dspace.mit.edu/handle/1721.1/11173` or `https://www.cs.virginia.edu/~evans/greatworks/shannon38.pdf`. Coincidentally, Shannon's Master thesis [3] also uses the term "symbolic analysis" in its title "A Symbolic Analysis of Relay and Switching Circuits".

Lemma 1.4.4 (Shannon expansion)**.** *Let f be a switching function for Var. Then*

$$f = (\neg z \wedge f\,|_{z=1}) \vee (z \wedge f\,|_{z=0}) \tag{1.29}$$

for all variables z from Var.

Figure 1.7

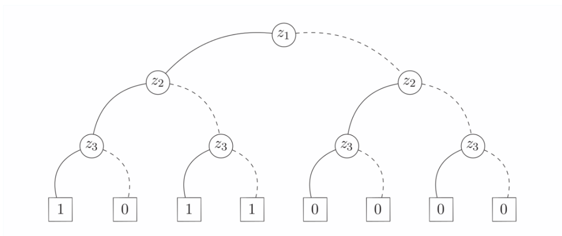


Figure 1.7: Binary decision tree for $z_1 \wedge (\neg z_2 \vee z_3)$

Definition 1.4.5 (Existential quantification)**.**

$$\exists z.f = f\,|_{z=1} \vee f\,|_{z=0} \tag{1.30}$$

1.4.2 Encoding transition systems by switching functions

The CTL model checking algorithm does its job by exploring the transition system by calculating $sat(_)$ of the subformulas of the given CTL formula. Next we show how the transition system and the sat-formulas can be represented by switching functions, i.e. that this data can be propositionally represented.

That this is possible, in principle, should be obvious, as mentioned. To find such an propositional encoding just means ultimately represents them by "bits" and bitectors. The transition system is finite, and all possible subsets of states are finite, as well, so that certainly is possible. But let's still review how it can be achieved systematically. By systematically, we mean "symbolically", using boolean formulas, propositional variables, etc.

Let's start by reminding us about transition systems. A transition system is of the form

$$T = (S, (Act), \to, I, P, L)$$

and we need to find an systematic way to encode it. In the incoding, we ignore the (transition) labels or actions Act) as irrelevant.

Encoding the states and sets of states

That's easy, we basically need enough bits so that there are enough different bit-patterns to have one for each state. For the states $s \in S$, one uses propositional variables $x_1, \ldots, x_n$, each state represented by some concrete evaluation $[x_1 = b_1, \ldots, x_n = b_n] : \{x_1, \ldots, x_n\} \to \{0, 1\}$, so the encoding of a state s is given by a bit-vector $enc(s) = b_1, \ldots, b_n$. Should the transition system not have exactly 2^n for some n is not a problem, the are simply unused bit-patterns. We *identify* in the following the set of states with the bit patterns, i.e. we assume that

$$S = \vec{x} \to \{0, 1\} \ .$$

We need to represents sets of states as well, i.e., sets $B \subseteq S$. That's pretty simple, and based on the fact that the set of all subsets of a set, say subsets of S is isorphic to all functions for S to $\{0, 1\}$ (or $\mathbb{B}$ or any 2-element set). I.e. 2^S is the "same" as $S \to \{0, 1\}$. A subset $B \subseteq S$ is equivalently represented by a function, conventionally call its *characteristic function* and written χ_B. For all elements of S, the function gives back 1 if the element is in B, and 0 otherwise.

$$\chi_B : (Eval(\vec{x}) \to \{0, 1\} = (\vec{x} \to \{0, 1\}) \to \{0, 1\} = S \to \{0, 1\} \ .$$

Being a finite function, it can also be seen as a bit-vector implementation of B, with the vector of length $\mid S \mid$.

The transition function

What works for sets of states, works analogously for the transition relation $\to \subseteq S \times S$, i.e., working with the characteristic function. We represent that as switching function, i.e., a $\{0, 1\}$-valued function over variables. To represent the tuple space $S \times S$, one needs to "duplicate" (the encoding of) S. We refer, when describing the encoding, to the states by variables x_i, and to get another copy of it, we simply assume a second, disjoint set of variables written $x'_1, \ldots, x'_n$, and taking the original variables x to refer to the source state of a transition and the primed versions y' as the target. The corresponding switching function is then of type $Eval(\vec{x}, \vec{x}') = (\vec{x}, \vec{x}') \to \{0, 1\}$. We write Δ for functions of that type (not $\chi_{\to}$, using the general notation for characteristic functions).

$$\Delta : ((\vec{x}, \vec{x}') \to \{0, 1\}) \to \{0, 1\}, \qquad \Delta(s_1, s_2[\vec{x}' \leftarrow \vec{x}]) = \begin{cases} 1 & \text{if } s_1 \to s_2 \\ 0 & \text{else} \end{cases} \ . \tag{1.31}$$

In equation (1.31), the $s_2[\vec{x}' \leftarrow \vec{x}]$ represents state s_2 with the variables $\vec{x}$ renamed to or substituted by their primed counter-parts $\vec{x}'$. Actually, we have not made an argument that we can actually do that. We are using variables and symbols, and of course one can substitute variables in a formula. But ultimately the formulaic writing refers to *switching function*. For instance, when mentioning a variable x, we are actually meaning a corresponding *projection function*. Section 1.4.1 covered all kinds of switching functions and operations on them, including the mentioned projection functions and how to combine them with boolean operators (which will be used later). But we have not covered how to

interpret renaming like $s_2[\vec{x}' \leftarrow \vec{x}]$ as operation on switching function. We leave that bit out in the presentation, it can be straightforwardly done.

Indeed if one has convinced oneself that all the relevant notations and concepts have their counterpart in the chosen representation, the switching functions, one can work formulaically with variables, and renaming of variables, boolean formulas, etc. In the context of LTL, we have talked about products of automata resp. transition systems (let's not make a fundamental discinction here in the discussion). We covered two simple cases, the synchronous and the asynchronous production or parallel composition. Imagine one is modelling a system, consisting of a number number transitions systems running in parallel, either synchronously or asychronously depending on the kind of system. Then the CTL model checking is done of the overall combined transition system.

If now each of the processes is encoded in the described way as switching function, then one could figure out if it's possible to define paralell composition directly on switching functions. And it's possible, but we omit also that here.

1.4.3 Exploring the encoding for model checking

Now with the transition system safely encoded, we need to show how the algorithm(s) cover earlier can be made working on those encodings. We have seen different versions of the algorithm(s), some more concrete than others. The overall design does not change, the algorithms works bottom-up through the structure of the given CTL formula. That has nothing to do with whatever encoding we have chosen, here switching functions, but it could be anything, so we don't have to change anything there. So the recursive algorithmic skeleton from Listing 1.2 can remain unchanged.

Let's then discuss the individual cases of the algo from Listing 1.2 one by one. For the temporal cases, see in particular also the corresponding cases from the characterization from Figure 1.6.

The first four cases in the switch statement are covered. In particular, $\wedge$ and $\neg$ can be interpreted on switching functions. The case for $\exists \bigcirc A$ requires to encode the pre-set calculation. With the encoding of the transition relation $\rightarrow$ and with the encoding of existential quantification and the renaming operation, the set $\{s \in S \mid \exists s'.s \rightarrow s' \wedge s' \in sat(\Phi)\}$ can be straightforwardly encoded as

$$\exists \vec{x}'.\, \underbrace{\Delta(\vec{x}, \vec{x}')}_{s \in pre(s')} \wedge \underbrace{\chi_A[\vec{x}' \leftarrow \vec{x}]}_{s' \in A} \tag{1.32}$$

Insteat of the U-case, let's discuss here how to represent $\exists\Diamond$, as we elaborated the fixpoint iteration on that slightly simpler special case. So, how to encode $\exists\Diamond$? Let's look at the formulation from Listing 1.5, the breadth-first backward exploration.

It's an iterative calculation of the predecessor sets. Of course the iteration itself needs not to be "encoded", it's still a loop. What needs to be encoded is the start set and the iterative step in each round from T_j to $T_{j+1} = T_j \cup pre(T_j)$.

The start set $T_0 = A$ can be represented by a switching function $f_0 = \chi_A$. Each T_j will likewise be representd by a switching function $f_j = \chi_{T_j}$. And with $\Diamond$ being basically

an iterated version of $\bigcirc$, the iteration step is basically what we have covered for $\exists\bigcirc$ in equation (1.32):

$$\exists \vec{x}'.\ \underbrace{\Delta(\vec{x},\vec{x}')}_{s\in pre(s')} \wedge \underbrace{f_j[\vec{x}' \leftarrow \vec{x}]}_{s'\in T_j} \tag{1.33}$$

The treatment of th until-operator is not much harder to encode (see its characterization from Figure 1.6). The resulting code is shown in Listing 1.8.

```
1  f_0(x⃗) := χ_{A_1}(x⃗);
2  j := 0;
3  repeat
4     f_{j+1}(x⃗) := f_{j+1}(x⃗) ∨ (χ_{A_2}(x⃗) ∧ ∃x⃗'.Δ(x⃗, x⃗') ∧ f_j(x⃗'));
5  until f_j(x⃗) = f_{j−1}(x⃗);
6  return  f_j(x⃗).
```

Listing 1.8: Symbolic exploration $sat(\exists(A_1\ U\ A_2))$

Finally, Listing 1.9 shows how to do the case for $\exists\Box$. The innovation here is that the greates fixpoint is calculated: in each iteration round, the current approximation gets smaller not larger, using $\wedge$, not $\vee$.

```
1  f_0(x⃗) := χ_A(x⃗);
2  j := 0;
3  repeat
4     f_{j+1}(x⃗) := f_{j+1}(x⃗) ∧ ∃x⃗'.Δ(x⃗, x⃗') ∧ f_j(x⃗'));
5  until f_j(x⃗) = f_{j−1}(x⃗);
6  return  f_j(x⃗).
```

Listing 1.9: Symbolic exploration $sat(\exists\Box A_)$

Bibliography

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

[2] Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model Checking.* MIT Press.

[3] Shannon, C. E. (1936/1940). A symbolic analysis of relay and switching circuits. Master's thesis, MIT Press.

Index

Index

binary decision tree, 23

CTL
    existential normal form, 8

ENF, 8
existential normal form, 8

negation, 9
normal form, 8, 9

progress, 6
projection function, 21

responseogress, 7