Course Script

IN 5110: Specification and Verification of Parallel Systems

IN5110, autumn 2021

Martin Steffen

Contents

Chapter 1 Formal methods

What is it about?

Learning Targets of this Chapter

The introductory chapter gives some motivational insight into the field of “formal methods” (one cannot even call it an overview over the field).

Contents

1.1 Introduction

About this document

This is the “script” or “handout” version of the lecture’s material covered on the slides.

In earlier iterations of the lecture, the document basically reproduced the slides in a more condensed way, with additional comments. On purpose, the slides used in class are kept not too full. Additional information and explanations said in class or using the whiteboard, without being shown on the slides, are shown here, as well as links and hints for further readings. In particular, *sources* and *bibliographic information* is shown mostly only here.

In the meantime, however, the script has to some extent emancipated from the slides. There is in the meantime more text and explanations. The text of the slides is often lists of bullet points or keywords. Those are no longer reproduced here, *accompagnied* by additional text. The content of the slides is turned into text (and with additional information and explanations). Likewise, it’s no longer the case that the header of each individual slide shows up here as sub-sub-section. Still the text here follows in structure exactly the sequence of slides, including the sectioning structure, also the figures, formulas etc., appear here and in the same order. So, the two variants of the material are still closely coupled, though the script now is more like an standalone text rather than commented slides.

It’s also best seen as “working document”, which means it will probably evolve during the semester.

Organizational matters

The lecture typically has not too many particpants, which means it becomes a bit more of a seminar. With a small audience, we strongly encourage active participation.

Exam: The exam will be oral.
Plan:

- see the homepage https://www.uio.no/studier/emner/matnat/ifi/IN5110
  - check for updates
  - only the master-webpages will be kept up-to date (not the PhD version)
- I might also be from time to time tangetially connected texts in the blogposts at https://martinsteffen.github.io/programverification/.

1.1.1 Motivating example

Let's start with a small example. It's atypical for the lecture, in that it's not about specifically concurrent or distributed systems but a numerical computation. Thanks to César Muñoz (NASA, Langley) for he example (which is taken from "Arithm'etique des ordinateurs" by Jean Michel Muller. See http://www.mat.unb.br/ayala/EVENTS/munoz2006.pdf or https://hal.archives-ouvertes.fr/ensl-00086707.

A simple computational problem

$$\begin{aligned} a_0 &= \tfrac{11}{2} \\ a_1 &= \tfrac{61}{11} \\ a_{n+2} &= 111 - \frac{1130 - \frac{3000}{a_n}}{a_{n+1}} \end{aligned} \tag{1.1}$$

The definition or specification from equation (1.1) seems so simple that it does not even look like a "problem", more like a first-semester task.

Real software, obviously, is mostly (immensely) more complicated. Nonetheless, certain kinds of software may rely on subroutines which have to calculate some easy numerical problems like the one sketched above (like for control tasks or signal processing).

You may easily try to "implement" it yourself, in your favorite programming language. If your are not a seasoned expert in arithmetic programming with real numbers or floats, you will come up probably with a small piece of code very similar to the one shown below (in Java).

```
public class Mya {

    static double a(int n) {
      if (n==0)
         return 11/2.0;
      if (n==1)
         return 61/11.0;
      return 111 - (1130 - 3000/a(n-2))/a(n-1);
    }

    public static void main(String[] argv) {
      for (int i=0;i<=20;i++)
         System.out.println("a("+i+") = "+a(i));
    }
}
```

Listing 1.1: A straightforward implementation

The example is not meant as doing finger-pointing towards Java, so one can program the same in other languages, for instance here in ocaml, a functional language.

```
(* The same example, in a different language *)

let rec a(n: int) : float =
  if n = 0
  then  11.0 /. 2.0
  else (if n = 1
        then 61.0 /. 11.0
        else (111.0 -. (1130.0 -. 3000.0 /. a(n-2)) /. a(n-1)));;
```

Listing 1.2: A different straightforward implementation

One can easily test the program for a given input of 20. In the output shown below, every second line is omitted for shortness. Stabilization at more or less 100 is also not a feature of Java. For instance, a corresponding ocaml program shows “basically” the same behavior; the exact numbers are slightly off.

The solution (?)

```
$ java mya
a(0)  = 5.5
a(2)  = 5.5901639344262435
a(4)  = 5.674648620514802
a(6)  = 5.74912092113604
a(8)  = 5.81131466923334
a(10) = 5.861078484508624
a(12) = 5.935956716634138
a(14) = 15.413043180845833
a(16) = 97.13715118465481
a(18) = 99.98953968869486
a(20) = 99.99996275956511
```

Alright, that’s what the program spits out and it’s not unplausible. We may vaguely remember that there is such a thing as a mathematical *series*. That’s sequences of numbers summed up, where a number at a given position is calculated by some formula from the value(s) of the previous one(s). Equation (1.1) is an example of (the specification of) a series of rationals resp. reals. One may additionally remember that math concerns itself with in particular *infinite* series and for those, the infinite sum may or may not stabilize or converge to a finite number. That’s called the limit of the series. Looking at the printed output from the implementation, it shows clear signs of convergence, stabilizing at 100, and already being quite close to it after 20 iterations.

Let’s consult out knowledge or a textbook about behavior of series, and theory tells that a_n for any $n \geq 0$ may be computed by using the following expression.

$$a_n = \frac{6^{n+1} + 5^{n+1}}{6^n + 5^n} \ .$$

With that alternative, non-recursive representation, it's clear to which number the series converges, and it's not 100, not even close.

$$\lim_{n \to \infty} a_n = 6 \ . \tag{1.2}$$

So, after 20 iterations, we should expect $a_{20} \approx 6$, not what the program actually calculates.

The example should cause concern for various reasons. The obvious one is that a seemingly correct program shows weird behavior. Of course, what is "seemingly" correct may lay in the eyes of the beholder.

One could shrug it off, making the argument that even the not so experienced programmer should be aware that floats in a programming language is most often different from "mathematical" real numbers and therefore the implementation is not to be expected to be 100% correct anyway. Of course in this particular example, the numbers are not just a bit off due to numerical imprecision, the implementation behaves completely different from what one could expect, the result of the implementation for the higher numbers seems to have nothing to do *at all* with the expected result.

But anyway, one conclusion to draw might be "be careful with floats" and accumulation of rounding errors. And perhaps take an extra course or two on computer arithmetic if you are serious about programming software that has to do with numerical calculations (control software, etc.). That's a valid conclusion, but this lecture will not follow the avenue of getting a better grip on problems of floats and numerical stability, it's a fields of its own.

The example can also be discussed from a different angle. The slides claim that the implementation is wrong insofar that the result should really be something like 6. One can figure that out with university or even school level knowledge about real analysis, series, and limits, etc. However, the problem statement is really easy. Actual problems are mostly much more complex even if we stick to situations, when the problem can be specified by a bunch of numerical equations, maybe *modelling* and representing some physical environment that needs to be monitored and controlled. It's unlikely to encounter a software problem whose "correct" solution can be looked-up in a beginner's textbook. What's correct anyway? In the motivational example, "math" tells us the correct answer should be approximately 6, but what if the underlying math is too complex to have a simple answer as to what the result is supposed to be (being unknown or even unobtainable as closed expression).

When facing a complex numerical (or computational) problem, many people nowadays would simply say *"let's use a computer to calculate the solution"*, basically assuming "what the computer says *is* the solution". Actually, along that lines, one could even take the standpoint that in problems like the ones from the example, the (Java) program is not the *solution* but the *specification* of the task.

That's not so unrealistic: the program uses recursion and other things, which can be seen as quite high-level. Then the task would be, to implement a piece of hardware, or firmware or some controller, that implements the specification, given by some high-level recursive description in some other executable format.

One can also imagine that the Java program is used for *testing* whether the more low-level implementation does the right thing, like comparing results or use the Java program to monitor the results in the spirit of *run-time verification*. The cautioning about "beware of numerical calculations" still applies, but the point more relevant to our lecture would be, that sometimes *specifications are not so clear either*, not even if they are "computer-aided". Later in the introduction, we say a program is correct only relative to a (formal) specification, but also the specifications themselves may be problematic and that includes the checking, even the automatic one, whether the specification is satisfied.

And we still may draw another parallel. The example shows in a way that software is *brittle*, in the sense that seemingly small deviations have large effects. In the example, small numerical approximations did not result in a small deviation of the expected outcome, but resulted in a different outcome. There, the cause is numerical instability, which is a complicated thing and outside the range of our lecture. But the small-cause-big-effect property is general in software. A misplaced comma or a loop- or array index wrong +/- one when working with a loop or an array can have drastic consequences (throwing an exception or derailing the program). In the context of parallel and concurrent programs, misplaced (or forgotten) locks and synchronization may have the same show-stopping effects.

Software in general is bad in tolerating deviations. A program with the array bound or the loop-condition one-off and thereby crashing is not more or less correct. It makes no sense to say that 1000 times the code looped perfectly through the loop body, only the last 1001st iteration went wrong, that's more than 99% correct...

This differentiates software and systems driven by software from traditional machines and constructions. Material constructions, like tools, houses, engines, bridges etc. are never perfect, there are all constructed with some tolerances, and each piece has its own tolerance, where tolerance is the accepted deviation from a piece's ideal and perfect measure or shape. Different constructions and different materials and different manufacturing methods have different tolerances, and for a particular kind of product, the lower the tolerances, the higher the quality and typically the price. The word "tolerance" says it already: real things can never perfect, but to some extent, deviations are acceptable. Each piece may have an individual tolerance, perhaps a cylinder in a combustion engine must be near perfect in its surface and wrt. its diameter, the plastic sealing ring for the exaust outlet at the same motor may have higher tolerances (already from the fact that its produced from a different material). Altogether, engineering science and enginering experience tells us that as long as the parts keep their (perhaps standardized) tolerances, the overall engines will do its job reliably (within its own tolerance and given side conditions, like it should not be used below -40 degrees and only oil or grease of some physical and chemical quality have to be use, and changed after a while etc.). And the producer is confident enough that it offers a couple of years guarantee on the engine or dishwasher, resp. the laws may enforce such a guarantee, so the producer better makes products that survive at least the guarantee period on average. Quality assurance is intended to make sure that products leaving the factory are in good shape. That will include checking said tolerances during the process (perhaps not on every single screw but *sampling*) and there will perhaps a quality end control of the finished product. If the engine shows signs of tearn and wear, one can do some maintentance, fastening some screws, cleaning the valves, replacing the sealing rings, and the machine is good for the next 2 years, the recommended maintenance period. Often, physical constructions don't have this "small-deviations-huge-effect" brittleness. If one single screw is not manufactured within its tolerances, it will probably overall be ok. Maybe even a missing screw is fine, and one beam slightly too short might degrade perhaps a construction in terms of long-lasting-ness or robustness to some extent. Of course, at some point too much is too much, and the whole system collapses or does not work, but it's typically a gradual thing. And engineers counter that with "redundancies" like beams are designed more stongly and with more steel than would be precisely needed not to collapse, and perhaps 10 screws are used, if 2 could do the job if guaranteed than none is forgotten or bad.

For software, the situation is rather different. There's no concept of tolerances, the "parts" like routines, libraries, modules or subcomponents are perfect replica from each other (ignoring the issue of deviations of versions and upgrades). There is never something like "yesterday I bought a piece of software, but it was not really good, one if the parts, the multiplication they used, was carelessly done, I brought it back and they fixed it and exchanged the rotten multiplication routine be a new one, and now it works". Sure, software can be and routinely is patched. . .

There's also typically no tear and wear in the material sense or maintenance intervals (again, the evolving software enviroment, upgrades of routines and libraries used by a piece of software may make it feels is if a piece if software "degrades", since this or that feature suddenly does not work any more as it used to).

And as far as producer guarantees are concerned, they often read like that:

```
<ACME> LICENSES THE LICENSED SOFTWARE "AS IS," AND MAKES NO EXPRESS OR
IMPLIED WARRANTY OF ANY KIND. <ACME> SPECIFICALLY DISCLAIMS ALL INDIRECT
OR IMPLIED WARRANTIES TO THE FULL EXTENT ALLOWED BY APPLICABLE LAW,
INCLUDING WITHOUT LIMITATION ALL IMPLIED WARRANTIES OF, NON-INFRINGEMENT,
MERCHANTABILITY, TITLE OR FITNESS FOR ANY PARTICULAR PURPOSE. NO ORAL OR
WRITTEN INFORMATION OR ADVICE GIVEN BY <ACME>, ITS AGENTS OR EMPLOYEES
SHALL CREATE A WARRANTY.
```

It's not really warranty, it's a legal formulation to assure as much as possible the absence of warranty, a *disclaimer*.

1.1.2 How to guarantee correctness?

Formal methods is about to establish and assure that software works properly with mathematical rigor, resp. using tools baseed mathematical and logical principles. And the lecture will look at some of those principles and logics and, to some extent, a few tools.

Working "properly" can mean many things. It can include that it meets user expectations, that it's user-friendly (which is difficult to achieve and difficult to determine), that it does the job efficiently etc. Often, one differentiates between *validation* and *verification*. Verification means to establish that a program or system does what is expected from it. The expectations are layed down as *specification*. The specification as well as the notion of conformance to the specification and process of establishing conformance are *formal*, i.e., based on mathematical principles, in particular logics. A program that meets its requirements or *satisfies* a the given *specification* is said to be *correct* (wrt. to that specification).

So correctness in that sense is on the one hand rather strict and precise: a formal connection between a specification and a system, carried out with mathematical rigor, ideally with the help of a tool. On the other hand, it's also restricted. Correctness for instance is not absolute, it's correct relative to a specification. Things not mentioned in the specification are not covered, and if the specification is "wrong" or ill-considered, same for the implementation.

Another thing to keep in mind is the following. We stated that formal verification is about a mathematical argument of logical proof that the specification is satisfied. A mathematical proof operates on "mathematical objects"

Correctness

- A system is **correct** if it meets its "requirements" (or specification)

Examples:

- **System:** The previous program computing a_n **Requirement:** For any $n \geq 0$, the program should conform with the previous equation

(incl. $\lim_{n\to\infty} a_n \approx 6$)

- **System:** A telephone system
- **Requirement:** If user A wants to call user B (and has credit), then *eventually* A will manage to establish a connection
- **System:** An operating system **Requirement:** A deadly embrace (nowaday's aka *deadlock*) will never happen

The "specifications" here are obviously anything else but formal. A "deadly embrace" is the original term for something that is now commonly called *deadlock*. It's a classical error condition in concurrent programs. In particular something that cannot occur in a sequential program or a sequential algorithm. It occurs when two processes try to gain access to two mutually dependent shared resources and each decide to wait indefinitely for the other. A classical illustration is the "dining philosophers".

The requirements, apart from the first one and except that they are unreasonable small or simple, are characteristic for "concurrent" or "reactive" system . As such, they are typical also for the kind of requirements we will encounter often in the lecture. The second one uses the word "eventually" which obtains a precise meaning in *temporal logics*. More accurately, it depends even on what kind of temporal logic one chooses and also how the system is modelled. Similarly for the last requirement, using the word "never".

Remarks The examples are taken from Holzmann's Ch. 1. See also Cousot's slides; K. Schneider's, Peled's, Clarke's and Holzmann's Chapter 1.

How to guarantee correctness?

- not enough to show that it **can** meet its requirements
- show that a system **cannot fail** to meet its requirements

Dijkstra's dictum "Program testing can be used to show the presence of bugs, but never to show their absence"

A lesser known dictum from Dijktra (1965) On proving programs correct: "One can never guarantee that a proof is correct, the best one can say is: 'I have not discovered any mistakes'. "

- *automatic* proofs? (Halting problem, Rice's theorem)
- any *hope*?

Dijksta's well-known dictum comes from [13]. The statements of Dijkstra can, of course, be debated, and have been debated. What about automatic proofs? It is impossible to construct a general proof procedure for arbitrary programs. It's a well-known fact that only programs in the most trivial "programming languages" can be automatically analysed (i.e., programming without general loops or recursion or if one assumes bounded memory). For clarity, one should perhaps be more precise, what can't be analysed. First of all, the undecidability of proplems refers to properties concerning the behavior or semantics of programs. Syntactic properties or similar may well be analyzed. Questions referring to the program text are typically decidable. A parser *decides* whether the source code is syntactically correct, for instance, i.e., adheres to a given (context-free) grammar. In most programming languages, type correctness is decidable (and the part of the compiler that decides on that is the type checker). What is not decidable are semantics properties of what happens when running the code. The most famous of such properties is the question whether the program terminates or not; that's known as the *halting problem*. The halting problem (due to Alan Turing) is only one undecidable property, in fact, *all* semantical questions are undecidable: every single semantical property is undecidable, with the exception of only two which are decidable. Those 2 decidable one are the 2 trivial ones, known as *true* and *false*, which hold for all programs resp. for none. The general undecidability of all non-trivial semantical properties is known as *Rice's theorem*.

As second elaboration: undecidability refers to analysis programs *generally*. Specific programs may well be analysed, of course. For instance, one may well establish for a particular program, that it terminates. It may even be quite easy, if one has only for-loops or perhaps no loops at all. After all, verification is about establishing properties about programs. It's only that one cannot make an algorithmic analysis for all programs.

The third point is on the nature of what decidability means. A decision procedure is a *algorithm* which makes a decision in a binary manner: yes or no. And that implies that the decision procedure terminates. There is no "maybe", and there is no non-termination in which case one would not know either. A procedure that can diverge in some cases is not a decision-procedure by a *semi-decision* procedure and the corresponding problem is only semi-decidable (or partially recursive).

Validation & verification

- In general, **validation** is the process of checking if something satisfies a certain criterion
- Do not confuse validation with *verification*

Validation "Are we building the right product?", i.e., does the product do what the user requires

Verification: "Are we building the product right?", i.e., does the product conform to the specification

The terminology and the suggested distinction is not uncommon, especially in the formal methods community. It's not, however, a universal consensus. Some authors define verification as a validation technique, others talk about validation & verification as being complementary techniques. However, it's a working definition in the context of this lecture, and we are concerned with *verification* in that sense.

Approaches for validation

testing
- check the actual system rather than a model
- focused on sampling executions according to some coverage criteria
- not exhaustive ("coverage")
- often informal, formal approaches exist (MBT)

simulation
- A model of the system is written in a PL, which is run with different inputs
- not exhaustive

verification "[T]he process of applying a manual or automatic technique for establishing whether a given system satisfies a given property or behaves in accordance to some abstract description (specification) of the system" [27].

1.1.3 Software bugs

Sources of errors

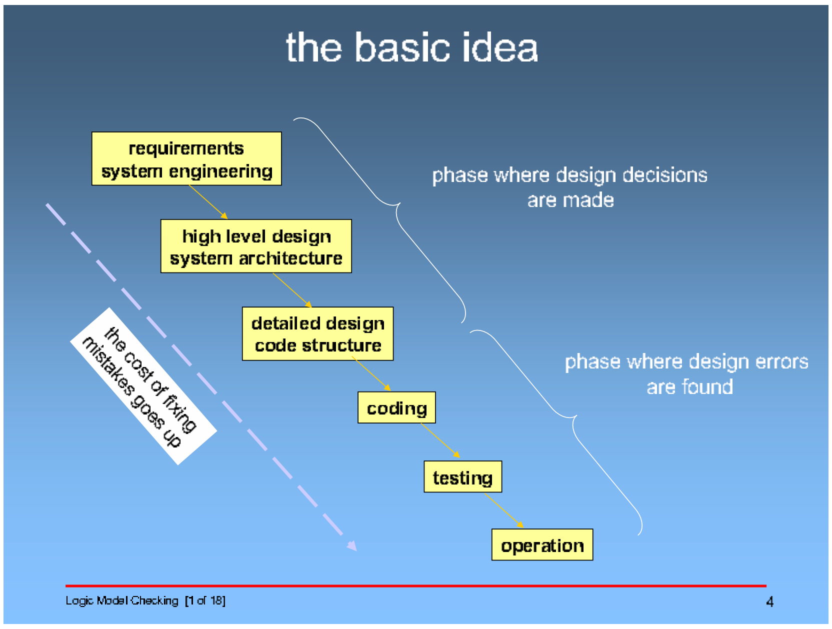
Errors may arise at different stages of the software/hardware development:

- specification errors (incomplete or wrong specification)
- transcription from the informal to the formal specification
- modeling errors (abstraction, incompleteness, etc.)

- translation from the specification to the actual code
- handwritten proof errors
- programming errors
- errors in the implementation of (semi-)automatic tools/compilers
- wrong use of tools/programs
- . . .

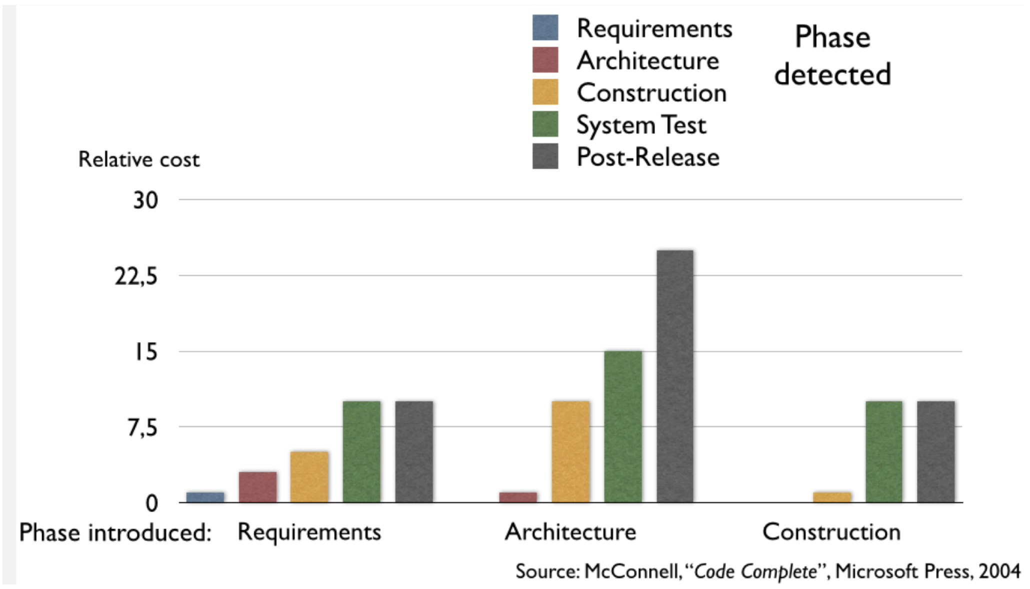
The list of errors is clearly not complete, sometimes a “system” is unusable, even if it’s hard to point with the finger to an error or a bug (is it “a bug or a feature”?). Different kinds of validation and verification techniques address different kinds of errors. Also testing as one (huge) subfield is divided in many different forms of testing, trying to address different kinds of errors.

Errors in the SE process


the basic idea
requirements system engineering
high level design system architecture
detailed design code structure
coding
testing
operation
phase where design decisions are made
phase where design errors are found
the cost of fixing mistakes goes up
Logic Model Checking [1 of 18]
4

The picture borrowed from G. Holzmann’s slides. Most software is developed according to some process with different phases or activities (and by different teams and with specific tools); often, the institution of even the legal regulators insist on some procedures etc. Many of such software engineering practices have more or less pronounced “top-down” aspects (most pronounced in a rigid waterfall development, which, however, is more an academic abstraction, less pronouced in agile processes). No matter how one organizes the development process, “most” errors are detected quite late on the development process, at least that’s what common wisdom, experience, and empirical results show. The figure (perhaps unrealistically simplifying) shows a top-down process and illustrates that certain kinds of errors (like design errors) are often detected only later. It should be clear (at least for such kind of errors), that the later the errors are detected, the more costly they are to repair.

Costs of fixing defects



The book the figures are taken from is [26] (a quite well-known source). The book itself attributes the shown figures to different other sources.

Hall of shame

- July 28, 1962: Mariner I space probe
- 1985–1987: Therac-25 medical accelerator
- 1988: Buffer overflow in Berkeley Unix finger daemon
- 1993: Intel Pentium floating point divide
- June 4, 1996: Ariane 5 Flight 501
- November 2000: National Cancer Institute, Panama City
- 2016: Schiaparelli crash on Mars

The information is taken from [15]. See also the link to that article.

July 28, 1962: Mariner I space probe The Mariner I rocket diverts from its intended course and was destroyed by mission control. A software error caused the miscalculation of the rocket's trajectory. *Source of error*: wrong transcription of a handwritten formula into the implementation code.

1985-1987: Therac-25 medical accelerator A radiation therapy device delivers high radiation doses. At least 5 patients died and many were injured. Under certain circumstances it was possible to configure the Therac-25, so the electron beam would fire in high-power mode but with the metal X-ray target out of position. *Source* of error: a *race condition*.

1988: Buffer overflow in the Berkeley Unix finger daemon An Internet worm infected more than 6000 computers in a day. The use of a C routine *gets*() had no limits on its input. A large input allows the worm to take over any connected machine. *Kind of error*: Language design error (Buffer overflow).

1993: Intel Pentium floating point divide A Pentium chip made mistakes when dividing floating point numbers (errors of 0.006%). Between 3 and 5 million chips of the unit have to be replaced (estimated cost: 475 million dollars). *Kind* of error: Hardware error.

June 4, 1996: Ariane 5 Flight 501 Error in a code converting 64-bit floating-point numbers into 16-bit signed integer. It triggered an overflow condition which made the rocket to disintegrate 40 seconds after launch. *Error:* exception handling error.

November 2000: National Cancer Institute, Panama City A therapy planning software allowed doctors to draw some "holes" for specifying the placement of metal shields to protect healthy tissue from radiation. The software interpreted the "hole" in different ways depending on how it was drawn, exposing the patient to twice the necessary radiation. 8 patients died; 20 received overdoses. *Error:* Incomplete specification / wrong use.

2016: Schiaparelli crash on Mars "[..] the GNC Software [..] deduced a *negative altitude* [..]. There was no check on board of the plausibility of this altitude calculation"

The errors on that list are quite known in the literature (and have been analyzed and re-analyzed and discussed). Note, however, that in some cases, the cause of the error is controversial, despite of lengthy (internal) investigations and sometimes even hearings in the US congress or other external or political institutions. The list is from 2005, other (and newer) lists certainly exists. A well-known collection of computer-related problems, especially those which imply societal risks and often based on insider information is Peter Neumann's Risk forum (now hosted by ACM), which is moderated and contains reliable information (in particular speculations are called speculations when the issues and causes are unclear). Not all one finds on the internet is reliable, there are many folk tales on "funny" software glitches.

Many errors may never see the public light or are openly analyzed, especially when they touch security related issues, or military or financial institutions. Of course, when a space craft explodes moments after lift-off or crash-lands on Mars on live transmission from ground control, it's difficult to swipe it under the rug. But not even then, it's easy to nail it down to the (or a) causing factor not to mention when it comes to put the blame somewhere or find ways to avoid it the next time. For instance, if it's determined that the ultimate cause was a missing semicolon (as some say was the case for the failure of the Mariner mission, but see below), then what does that mean and how to react? Tell all NASA programmers to double-check semicolons next time, and that's it? Actually, looking more closely, one should not think of the bug as a "syntactic error" and it's short-sighted to think of it as a typo.

Indeed, in the Mariner I case, the error is often attributed to a "hyphen", sometimes a semicolon. Other sources (who seem well-informed) speak of an overbar, see the IT world article, which refers to a post in the risk forum. Ultimately, the statement that it was a "false transcription" is confirmed by those sources. It should be noted that "transcription" means that someone had to punch in patterns with a type-writer like machine into punch cards. The source code (in the form of punch cards) was, obviously, hard to "read" by humans, so *code inspection* or *code reviews* were hard to do at that level. To mitigate the problem of erronous transcription, machines called *card verifiers* where used. Bascially, it meant that two people punched in the same program and verification meant that the result was automatically compared by the verifier.

1.1.4 On formal methods

Sources

The slides are inspired by introductory material of the books by K. Schneider and the one by D. Peled ([30, Section 1.1] and [27, Chapter 1]).

What are formal methods?

FM "Formal methods are a collection of notations and techniques for describing and analyzing systems" [27]

- **Formal**: based on "math" (logic, automata, graphs, type theory, set theory . . .)
- formal **specification** techniques: to unambiguously describe the system itself and/or its properties
- formal **analysis/verification**: techniques serve to verify that a system satisfies its specification (or to help finding out why it is not the case)

Terminology: Verification

The term *verification*: used in different ways

- Sometimes used only to refer the process of obtaining the formal correctness proof of a system (deductive verification)
- In other cases, used to describe any action taken for finding errors in a program (including *model checking* and maybe *testing*)

Formal verification (reminder) Formal verification is the process of applying a manual or automatic *formal* technique for establishing whether a given system satisfies a given property or behaves in accordance to some abstract description (*formal* specification) of the system

Saying *'a program is correct'* is only meaningful w.r.t. a given spec.!

The term "verification" is used (by different people, in different communities) in different ways, as we hinted at already earlier. Sometimes, for example, testing is not considered to be a verification technique.

Limitations

- Software verification methods do not guarantee, in general, the correctness of the code itself but rather of an abstract *model* of it
- It cannot identify fabrication faults (e.g. in digital circuits)
- If the specification is incomplete or wrong, the verification result will also be wrong
- The implementation of verification tools may be faulty
- The bigger the system (number of possible states) more difficult is to analyze it (*state space explosion problem*)

For a discussion of issues like these, one may see the papers “Seven myths of formal methods” and “Seven more myths of formal methods” ([18] [5]).

Any advantage?

be modest Formal methods are not intended to guarantee absolute reliability but to *increase* the confidence on system reliability. They help minimizing the number of errors and in many cases allow to find errors impossible to find manually.

- remember the *VIPER* chip

Parnas has a more dim view on formal methods. Basically he says, no one in industry is using them and the reason for that is that they are (basically) useless (and an academic folly). Parnas is a big name, so he’s not nobody, and his view is probably shared explicitly by some. And implicitly perhaps shared by many, insofar that formal methods has a niche existance in real production software.

However, the view is also a bit silly. The argument that *no one* uses it is certainly an exaggeration. There are areas where formal methods are at least encouraged by regulatory documents, for instance in the avionics industry. One could make the argument that high-security applications (like avionics software) is a small niche, therefore formal method efforts in that direction are a folly for most progammers. Maybe, but that does not discount efforts in areas where one thinks it’s worth it (or is forced by regulators to do it).

Secondly, even if really no one in industry would use such methods, that would not discount a research effort, including academic research. The standpoint that the task of academic research is to write papers about what practices are currently profitably employed in mainstream industry is a folly as well.

Maybe formal methods also suffer a bit from similar bad reputation as (sometimes) artificial intelligence has (or had). Techniques as investigated by the formal method community are opposed, ridiculed and discounted as impractical until they “disappear” and then become “common practice”. So, as long as the standard practicioner does not use something, it’s “useless formal methods”, once incorporated in daily use it’s part of the software process and quality assurance. Artificial intelligence perhaps suffered from a similar phenomenon. At the very beginning of the digital age, when people talked about “electronic brains” (which had, compared to today, ridiculously small speed and capacity), they trumpeted that the electronic brains can “think rationally” etcetc., and the promised that soon they would beat humans in games that require strategic thinking like tic-tac-toe.

The computers very soon did just that, with "fancy artifical intelligence" techniques like back-tracking, branch-and-bound or what not (branch-and-bound comes from operations research). Of course the audience then said: Oh, that's not intelligence, that's just brute force and depth-first search, and nowadays, depth-first seach is taught in first semester or even school. And Tic-tac-toe is too simple, anyway, the audience said, but to play chess, you will need "real" intelligence, so if you can come up with a computer that beats chess champions, ok, then we could call it intelligent. So then the AI came up with much more fancy stuff, heuristics, statistics, bigger memory, larger state spaces, used faster computers, but people would still state: well, actually a chess playing computer is not intelligent, it's "just" a complex search. So, the "intelligence" those guys aim at is always the stuff that is not yet solved. Maybe the situation is similar for formal methods.

Perhaps another parallel which has led to negative opinions like the one of Parnas is that the community sometimes is too big-mouthed. Like promising an "intelligent electronic brain" and what comes out is a tic-tac-toe playing back-tracker... For the formal methods, it's perhaps the promise to "guarantee 100% correctness" (done based on "math") or at least perceived as to promise that. For instance, the famous dictum of Disjktra that testing cannot guarantee correctness in all cases is of course in a way a triviality (and should be uncontroversial), but it's perhaps perceived to mean (or used by some to mean) that "unlike testing, the (formal) method can guarantee that". Remember the Viper chip (a "verified" chip used in the military, in the UK). See [29] for a short historical account of the rise and the fall of the Viper chip and the (legal) battles around it. Interestingly an important part of the battle was about the question, discussed in course "what is a mathematical proof". Also the book [24] discusses that.

Another netfind: "bitcoin" and formal methods :-)

Cardano seems to be one of quite many bitcoin-style proposals based on block-chain, marketed under the mysterious slogan "Making the world a work better for all"... As far a cryptocurrencies goes, it's not too esotheric. According to a current ranking of market capitalization of different "coins", it's ranked number 5.

provably correct

Using formal methods

Used in different stages of the development process, giving a classification of formal methods

1. We describe the system giving a *formal specification*
2. We can then *prove some properties* about the specification
3. We can proceed by:
   - Deriving a program from its specification (formal synthesis)
   - *Verifying* the specification wrt. implementation

Formal specification

- A specification formalism must be unambiguous: it should have a *precise syntax and semantics*
  - Natural languages are not suitable
- A trade-off must be found between expressiveness and analysis feasibility
  - More expressive the specification formalism more difficult its analysis

Do not confuse the specification of the system itself with the specification of some of its properties

- Both kinds of specifications may use the same formalism but not necessarily. For example:
  - the system specification can be given as a program or as a state machine
  - system properties can be formalized using some logic

Proving properties about the specification

To gain confidence about the correctness of a specification it is useful to:

- Prove some properties of the specification to check that it really means what it is supposed to
- Prove the equivalence of different specifications

Example a should be true for the first two points of time, and then oscillate. The example is taken from [30]

- some attempt:

$$a(0) \land a(1) \land \forall t.\ a(t+1) = \neg a(t)$$

One could say the specification is *INCORRECT!* and/or incomplete. The error may be found when trying to prove some properties. Implicitly (even if not stated), is the assumption that t is a natural number. If that is assumed, then the last conjuct should apply also for $t = 0$, but that contradicts the first two conjucts.

So perhaps a correct (?) specification might be

$$a(0) \land a(1) \land \forall t \geq 0 . a(t+2) = \neg a(t+1)$$

Formal synthesis

- it would be helpful to automatically obtain an implementation from the specification of a system
- difficult since most specifications are *declarative* and not *constructive*
  - They usually describe **what** the system should do; not **how** it can be achieved

Example: program extraction

- specify the operational semantics of a programming language in a constructive logic (calculus of constructions)
- prove the correctness of a given property wrt. the operational semantics (e.g. in Coq)
- extract (*ocaml*) code from the correctness proof (using Coq's extraction mechanism)

Verifying specifications w.r.t. implementations

Mainly two approaches:

- Deductive approach ((automated) theorem proving)
  - Describe the specification φ_{spec} in a formal model (logic)
  - Describe the system's model φ_{imp} in the same formal model
  - Prove that $\varphi_{imp} \implies \varphi_{spec}$
- Algorithmic approach
  - Describe the specification φ_{spec} as a formula of a logic
  - Describe the system as an interpretation M_{imp} of the given logic (e.g. as a finite automaton)
  - Prove that M_{imp} is a "model" (in the logical sense) of φ_{spec}

A few success stories

- Esterel Technologies (synchronous languages – Airbus, Avionics, Semiconductor & Telecom, ...)
  - Scade/Lustre
  - Esterel
- Astrée (Abstract interpretation – used in Airbus)
- Java PathFinder (model checking – find deadlocks on multi-threaded Java programs)
- verification of circuits design (model checking)
- verification of different protocols (model checking and verification of infinite-state systems)
- Z3 (and other) constraint solver

...

Classification of systems

Before discussing how to choose an appropriate formal method we need a classification of systems

- Different kindd of systems and not all methodologies/techniques may be applied to all kind of systems
- Systems may be classified depending on
  - *architecture*
  - *type of interaction*

The classification here follows Klaus Schneider's book "Verification of reactive systems" [30]. Obviously, one can classify "systems" in many other ways, as well.

Classification of systems: architecture

- Asynchronous vs. synchronous hardware
- Analog vs. digital hardware
- Mono- vs. multi-processor systems
- Imperative vs. functional vs. logical vs. object-oriented software
- Concurrent vs. sequential software
- Conventional vs. real-time operating systems
- Embedded vs. local vs. distributed systems

Classification of systems: type of interaction

- **Transformational** systems: Read inputs and produce outputs – These systems should always terminate
- **Interactive** systems: Idem previous, but they are not assumed to terminate (unless explicitly required) – Environment has to wait till the system is ready
- **Reactive** systems: Non-terminating systems. The environment decides when to interact with the system – These systems must be fast enough to react to an environment action (real-time systems)

Taxonomy of properties

Text Many specification formalisms can be classified depending on the kind of properties they are able to express/verify. Properties may be organized in the following categories

Functional correctness The program for computing the square root really computes it

Temporal behavior The answer arrives in less than 40 seconds

Safety properties (*"something bad never happens"*): Traffic lights of crossing streets are never green simultaneously

Liveness properties (*"something good eventually happens"*): process A will eventually be executed

Persistence properties (stabilization): For all computations there is a point where process A is always enabled

Fairness properties (some property will hold infinitely often): No process is ignored infinitely often by an OS/scheduler

Remarks K. Schneider does not mention "Functional correctness" nor "temporal behavior". I think I took them from Peled's book (?)

When and which formal method to use?

Examples:

- Digital circuits . . . (BDDs, model checking)
- Communication protocol with unbounded number of processes. . . . (verification of infinite-state systems)
- Overflow in programs (static analysis and abstract interpretation)
- . . .

Open distributed, concurrent systems ⇒ *Very difficult!*

Need the combination of different techniques

It should be clear that the choice of method depends on the nature of the system and what kind of properties one needs to establish. The above lists basically states the (obvious) fact that the more complex (and unstructured) systems get, the more complex the application of formal method becomes (hand in hand with the fact that the development becomes more complex). The most restricted form perhaps is digital circuits and hardware. The initial successes for model checking were on the area of hardware verification. Ultimately, one can even say: at a certain level of abstraction, hardware is (or is supposed to be) a finite-state problem: the piece of hardware represents a finite-state machine built up of gates etc, which work like boolean functions. It should be noted, though, that this in itself is an *abstraction*: the physical reality is not binary or digital and it's a hard engineering problem to make physical entities (like silicon, or earlier tubes or magnetic metals) to actually behave as if they were digital (and to keep it stable like that, so that it still works reliably in a binary or finite-state fashion after trillions of operations. . .) In a way, the binary (or finite-state) abstraction of hardware is a *model* of the reality, one one can check whether this model has the intended properties. Especially useful for hardware and "finite state" situations are *BDDs* (binary decision diagrams) which were very successful for certain kinds of model checkers.

1.1.5 Formalisms for specification and verification

Some formalisms for specification

- Logic-based formalisms
  - Modal and temporal logics (E.g. LTL, CTL)
  - Real-time temporal logics (E.g. Duration calculus, TCTL)
  - Rewriting logic
- Automata-based formalisms
  - Finite-state automata
  - Timed and hybrid automata
- Process algebra/process calculi
  - CCS (LOTOS, CSP, ..)
  - π-calculus . . .
- Visual formalisms
  - MSC (Message Sequence Chart)
  - Statecharts (e.g. in UML)

  - Petri nets

It should go without saying that the list is rather incomplete list. The formalisms here, whether they are "logical" or "automata-like" are used for specification of more reactive or communicative behavior (as opposed to specifying purely functional or input-output behavior of sequential algorithms). By such behavior, we mean describing a step-wise or temporal behavior of a system ("first this, then that. . . ."). Automata with their notions of states and labelled transitions embody that idea. Process algebras are similar. On a very high-level, they can partly be understood as some notation describing automata; that's not all to it, as they are often tailor-made to capture specific forms of interaction or composition, but their behavior is best understood as having states and transitions, as automata. The mentioned logics are likewise concerned with logically describing reactive systems. Beyond purely logical constructs (and, or), they have operators to speak about steps being done (next, in the future . . .). Typical are *temporal* logics, where "temporal" does not directly mean refering to clocks, real-time clocks or otherwise. It's about specifying steps that occur one after the other in a system. There are then real-time extensions of such logics (in the same way that there are real-time extensions of programming language as well as real-time extensions of those mentioned process calculi).

Whether one should place the mentioned "visual" formalisms in a separate category may be debated. Being visual refers just to a way of representation (after all also automata can be (and are) visualized, resp. "visual" formalisms have often also "textual" representations.

Some techniques and methodologies for verification

- algorithmic verification
  - Finite-state systems (model checking)
  - Infinite-state systems
  - Hybrid systems
  - Real-time systems
- deductive verification (theorem proving)
- abstract interpretation
- formal testing (black box, white box, structural, . . .)
- static analysis
- constraint solving

1.1.6 Summary

Summary

- **Formal methods** are useful and needed
- which FM to use depends on the problem, the underlying system and the property we want to prove
- for real complex systems, only part of the system may be formally proved and no single FM can do the task
- our course will concentrate on
  - temporal logics as a specification formalism

  - safety, liveness and (maybe) fairness properties
  - Spin (LTL model checking)
  - few other techniques from student presentation (e.g., abstract interpretation, CTL model checking, timed automata)

Ten Commandments of formal methods

From “Ten commandments revisited” [6]

1. Choose an appropriate notation
2. Formalize but not over-formalize
3. Estimate costs
4. Have a formal method guru on call
5. Do not abandon your traditional methods
6. Document sufficiently
7. Do not compromise your quality standards
8. Do not be dogmatic
9. Test, test, and test again
10. Do reuse

Further reading

Especially this part is based on many different sources. The following references have been consulted:

- Klaus Schneider: Verification of reactive systems, 2003. Springer. Chap. 1 [30]
- G. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming, 2000. Addison Wesley. Chap. 2 [1]
- Z. Manna and A. Pnueli: Temporal Verification of Reactive Systems: Safety, Chap. 0 This chapter is also the base of lectures 3 and 4. [25]

Chapter 2 Logics

What is it about?

Learning Targets of this Chapter

The chapter gives some basic information about “standard” logics, namely propositional logics and (classical) first-order logics (and maybe more).

Contents

2.1 Introduction

What’s logic?

As discussed in the introductory part, we are concerned with formal methods, verification and analysis of systems etc., and that is done relative to a *specification* of a system. The specification lays down (the) desired properties of a system and can be used to judge whether a system is correct or not. The requirements or properties can be given in many different forms, including informal ones. We are dealing with *formal* specifications. Formal for us means, it has not just a precise meaning, that meaning is also fixed in a mathematical form for instance, in the form of a “model”[1] We will mostly not deal with informal specifications nor with formal specifications that are unrelated to the *behavior* in a broad sense of a system.

For example, a specification like

the system should cost 100 000$ or less, incl. VAT

could be seens as being formal and precise. In practice, such a statement is probably not precise enough for a legally binding contract (what’s the exchange rate, if it’s for Norwegian usage? Which date is taken to fix the exchange rate, the date of signing the contract, the scheduled delivery date, or the actual delivery date? What’s the “system” anyway, the installation? The binary? Training? Warranty, etc.) All of that would be “formalized”

[1]The notion of model will be variously discussed later resp. given a more precise meaning in the lecture. Actually, it will be given a precise mathematical meaning in different technical ways, depending on which framework, logics, etc. we are dealing with; the rough idea remains the “same”, though.

in a legal contract not even readable for mathematicians, but only for lawyers, but that's not the kind of formalization we are dealing with.

For us, properties are expressed in "logics". That is a very broad term, as well, and we will encounter various different logics and "classes" of logics.

This course is not about fundamentals of logics or philosophical questions, like "is logic as discipline a subfield of math, or is it the other way around". We are also mostly not much concerned with fundamental questions of logical *meta-theory*. If one has agreed on a logic (including notation and meaning), one can use that to fix some "theory" which is expressed *inside* the logic. For example, if one is interested in formally deriving things about natural numbers, one could first choose first-order logic as general framework, then select symbols proper for the task at hand (getting some grip on the natural numbers), and then try to axiomatize them and formally derive theorems inside the chosen logical system. As the name implies meta-theory is *not* about things like that, it's about what can be said *about* the chosen logic itself: Is my logic decidable? How efficient can it be used for making arguments? How does its expressivity compares to that of other logics? ... Such questions will pop up from time to time, but are not at the center of the course. For us, logic is more of a tool for validating programs, and for different kind of properties or systemd, we will see what kind of logics fits.

Still, we will introduce basic vocabulary and terminology needed when talking *about* a logic (on the meta-level, so to say). That will include notions like formulas, satisfaction, validity, correctness, completeness, consistency, substitution ..., or at least a subset of those notions.

When talking about "math" and "logics" and what their relationship is: some may have the impression that math as discipline is a formal enterprise and formal methods is kind of like an invasion of math into computer science or programming. It's probably fair to say, however, that for the working mathematician, math is *not* a formal discipline in the sense the formal methods people or computer scientists do their business. Sure, math is about drawing conclusions and doing proofs. But most mathematicians would balk at the question "what's the logical axioms you use in your arguments?" or "can go give me the grammar of the syntax you use in your reasoning?". That only bothers mathematicans (to some extent) who prove things *about* logical systems, i.e., who take logics as object of their study. But even those will probably not write their arguments *about* a formally defined logic *inside* a(nother?) logical system. That formal-method people are more obsessed with such nit-picking questions has perhaps two reasons. For one is that they want not just clear, elegant and convincing arguments, they want that the *computer* makes the argument or at least assist in making the argument. To have a computer program do that, one needs to be 100% explicit what the syntax of a formal system is and what it means, how to draw arguments or check satisfaction of a formula etc.

Another reason is that they are often not dealing with clear and elegant things, the objects of study for formal-method people are, mathematically seen, often "dirty stuff". One tries to argue for the corrrectness of a program, an algorithm, maybe even an implementation. That often means one does not deal with an elegant mathematical structure but some specific artifact. It's not about "in principle, the idea of the algorithm is correct" (although that is also important); whether the code is correct or not not depends also on special corner cases, uncovered conditions, or other circumstances. There is no such argument like “the remaining cases work analogously. . . ”: A mathematician might get away with that, but a computer program would not.

Additionally, when making proofs about software, it’s often not about “the remaining five analogous cases”. Especially, in concurrent program or algorithms, one has to cover a huge amount of possible *interleavings* (combinations of orderings of executions), and a incorrectness, like a race condition, may occur only in some very seldom specific interleavings. So it’s more like there are “5 fantastillion more cases”. . . Proving that a few exemplary interleavings are correct (or test a few) will simply not do the job. That’s indeed one problem with *testing* concurrent systems. Even if one tests a huge amount of interleavings (and coaching an implementation to do check particular interleavings as opposed to rely on chance is also non-trivial), this normally pales in comparison to the astronomical number of possible interleavings. For sequential systems or a sequential, deterministic procedures, the possible behaviors is also astronomical, because typically there are many possible inputs, conceptually infinitely many often. However, being deterministic, at least the same input should lead to the same reaction. That’s a big help, in particular when it comes to reproducability. Furthermore, there are techniques that allow to tailor the input in such a way that *corner cases* are taken, for instance, feeding input that for all branches both the positive as well as the negative branch is executed by at least one test. That’s known as a form of test *coverage*.

For concurrent systems, what happens in one particular run is *non-deterministic*. In particular there is no part of the program code responsible that this or that decision is take (as is the case for a conditional statement in the sequential case). The different interleavings are done by the scheduler or the timings of the parallel processes.

General aspects of logics: truth vs. provability We will see different logics, with different syntax’es, different purposes etc. But the way they are defined and introduced will share some common pattern.

They their syntax are given as a formal *language* fixed by grammar. Mostly they will have the familar boolean propositional logic connectives as core (“and”, “not” etc.), and there will be various additional syntax on top.

Expressions in the logical syntax are generally called *formulas*. Though some more specific names exists, like formulas of propositional logic may be called *propositions*, or formulas without free variables in first-order logic are called *sentences* by some. We mostly use φ (and φ_1, ψ', χ . . .) for formulas independent of the current setting we discuss.

More interesting is the question what formulas respresent or specify. That’s about the *semantics* or meaning of the syntax. The question about what some syntax means cannot be asked just for formulas, the same is relevant for *programs*: the program code, in the form of its (abstract) syntax tree has a meaning, the programs semantics, that’s what the program does at run-time.

For logics, the question about the meaning of a formula is the question about its “truth status”, like is the formula true or not. That true-or-false is, however a bit too simplistic, in that it’s too categorical or unconditional.

The fact whether a formula is true or not is influenced by additional "factors". Other terminology for being true is, the formula *holds*.

...

If a formular is unconditionally true, that's called the formula is *valid* or in the case of propositional formulas, it's a (propositional) tautology. So whether a formula

truth vs. provability:

when does a formula *hold*, is *true*, is *satisfied*, valid, satisfiable

syntax vs. semantics/models, model theory vs. proof theory

- $\sigma \models \varphi$ (or $\models_\sigma \varphi$):
  - σ satisfies φ
  - σ models φ (σ is a model of φ)
- $[\![\varphi]\!]^\sigma = \top$:
  - with σ as propositional variable assignment, φ is true or φ holds
  - the semantics of φ under σ is $\top$ ("true")

Of course, there are formulas whose truth-ness does *not* depend on particular choices, being unconditionally true (or other unconditionally false). They deserve a particular name like (propositional) "tautology" (or "contradiction" in the negative case).

Another name for a generally true formula or a formula which is true under all circumstances is to say it's *valid*. For propositional logic, the two notions (valid formula and tautology) coincide.

If we got to more complex logics like first-order logics, things get more subtle (and the same for modal logics later). In those cases, there are more "ingredients" in the logic that are potentially "non-fixed", but "movable". For example, in first-order logic, one can distinguish two "movable parts". First-order logic is defined relative to a so-called signature (to distinguish them from other forms of signatures, it's sometimes called first-order signature). It's the "alphabet" one agrees upon to work with. It contains functional and relational symbols (with fixed arity or sorts). Those operators define the "domain(s) of interest" one intends to talk about and their syntactic operators. For example, one could fix a signature containing operators `zero`, `succ`, and `plus` on a single domain (a single-sorted setting) where the chosen names indicate that one plans to interpret the single domain as natural numbers. We use in the discussion here `typewriter` font to remind that the signature and their operators are intended as *syntax*, not as the semantical interpretation (presumably representing the known mathematical entities 0, the successor function, and $+$, i.e., addition). There are also syntactic operators which constitute the logic itself (like the binary operator $\wedge$, or maybe we should write `and`...), which are treated as really and absolutely fixed (once one has agreed on doing classical first-order logic or similar).

The meaning of the symbols of a chosen signature, however, are generally *not* a priori fixed, at least when doing "logic" and meta-arguments about logics. On the other hand, when doing program verification, typically one is not bothered about that, one assumes a fixed interpretation of a given signature. Anyway, the elements of the signature are not typically thought of as *variables*, but choosing a semantics for them is one of the non-fixed, variable parts when talking about the semantics of a first-order formula. That part, fixing the functional and relational symbols of a given signature is called often an *interpretation*. There is, however, a second level of "non-fixed" syntax in a first-order formula, on which the truthness of a formula depends: those are (free) *variables*. For instance, assuming that we have fixed the interpretation of `succ`, `zero`, `leq` (for less-or-equal) and so on, by the standard meaning implied by their name, the truth of the formula `leq(succ x, y)` depends on the choices for the free variables `x` and `y`.

To judge, whether a formula with free variables holds or not, one this needs to fix two parts, the interpretation of the symbols of the alphabet (often called the interpretation), as well as the choice of values for the free variables. Now that the situation is more involved, with two levels of choices, the terminolgy becomes also a bit non-uniform (depending on the text-book, one might encouter slightly contradicting use of words).

One common interpretation is to call the choice of symbols the *interpretation* or also *model*.

2.2 Propositional logic

2.2.1 Introduction

A quite basic form of logic is known as *propositional* or also *boolean* logic (in honor of George Boole).[2] Other names are statement logic or sentential logic. It's also underlying binary hardware, binary meaning "two-valued". The two-valuedness is the core of *classical* logics in general, the assumption that there is some *truth* which is either the case or else not (true or else false, nothing in between or "tertium-non-datur"). This is embodied by classical propositional logic.

In the following, we introduce the three ingredients of a mathematical logics, its *syntax*, its *semantics* (or notion of models, its semantics, its interpretation) and its *proof theory*. We don't go too deep into any of those, especially not *proof theory*.

Model theory is concerned with the question of when formulas are "true", what structure satisfies a formula (its model). Proof theory, on the other hand, is about when formulas are *provable* by a formal procedure or derivation system. Those questions are not independent. A provable formula should ideally be semantically true, that's the question of *soundness* of the proof system. And vice versa: all formulas which are actually "true" should ideally be provably true as well (a question of *completeness*). Notationally, one often uses the symbol $\vdash$ when referring to proof-theoretical notions and $\models$ for model-theoretical, mathematical ones. $\vdash \varphi$ thus would represent φ is derivable or provable, and $\models \varphi$ for the formula being "true" (or valid etc.) referring to its semantics.

[2] Like later for first-order logic and other logics, there are variations of that, not only syntactical, some also essential. We are dealing with *classical* propositional logics without without being too dogmatic which selection of operators we use. One can also study intuitionistic versions. One of such logic is known as *minimal* intuitionistic logic, that has implication $\rightarrow$ as the only constructor.

Non-classical logics The following remarks are not part of the technical content of the lecture, but it may be interesting to at least be aware of alternatives of the somehow classical path we followed. As stated, classical logics is based on the two valued interpretation of "truth": if not true, it's false, if not false, it's true, there is no middle ground ("tertium-non-datur" in Latin or "the law of the excluded middle). This way of thinking underlies also classical first-order logic (see later), which is somehow the "standard" garden-variety all-purpose logic for many (not all). First-order logic emerged (ca .100 years ago) qas *the* underlying logic in the attempt for "formalize math" once and for all. In a way that attempt "failed". By formalize something (like math), it's not just meant to syntactically represent the domain of interest (like math or parts of it), but to use deductive systems to *derive* in a prescribed manner statements that hold (the theorems) and thereby answering, in a formalistic manner, in principle the truth or falsehood of hopefully all possible mathematical statements (after which one could fire all mathematicians, as their issues can be solved without them. . .).

A calculatorial, step-by-step approach to derive true facts is, as we would say nowadays, the attempt to find an *algorithm* or a *program* for that task. That "grand challenge", however to formalize and solve ("algorithmically") mathematics once and for all failed, it was ultimately shot down by Kurt Gödel who proved (among other things) the futility of this attempt: derivability in logics like first-order logics (and related ones) is undecidable. Turing's famous undecidability of the halting problem is closely related, which may be plausible in that are both concerned with *deciding* results of a *computational* procedure ("can I derive step-by-step a theorem in this logic", "can I do a step-by-step excecution of that program and will it halt"). The word *decision* is meant one expects a clear-cut "yes-or-no" answer. So that corresponds to a classical interpretation. The undecidability of some logics means that being *true* and being *provable* are different things and cannot be made the same. Being true does not necessarily mean being provably, some statements are neither provable not disprovable (in the sense that their negation is provable).

Intuitionistic (or constructive) approaches to logics basically take the stand-point that logic is not about "truth" in the classical sense but about "provability". In the light of the above discussion, it also leads to drop the principle of tertum-non-datur, i.e. $P \vee \neg P$ is *no longer* generally true. For mathematicin, The question whether they do their job in "classical" or "intuitionistic" thinking is a bit outlandish for most (perhaps an intuitionistic thinker would try to avoid proofs by contradiction). For computer scientists, the intuitionistic approach has a certain appeal, since one motivation of it is that logic is about proving things in an algorithmic manner (i.e., with a computer).

The ultimate cause of troubles is that one tries to prove things about *infinite* entities with *finite* means, like with a finite number of steps of a halting program or deduction process (already the natural numbers as a simple mathematical structure are ideally infinite). Boolean logic, which we start with, is not expressive enough to encounter troubles of infinite, everything is finite.

Nonetheless, one could study intutitionistic propositional logic. It is also interesting that such things are *also* relevant in connection with hardware. A logic circuit is seen a binary device (voltage high or else volrage low, which corresponds to the boolean values true or false), but actually that's an abstraction. In reality, for physical reasons, hardware may have intermediate states, maybe at a certain point in time, or may be instable due to oscillation etc). Without going into details, there is a connection between "stable" boolean circuits and intuitionistic logics.

2.2.2 Syntax: propositions as formulas of propositional logic

Logics express themselves syntactically by so-called *formulas*. Different logics used different formulas. Propositional formulas, the formulas of propositional logics, are commonly also called just *propositions*. One can stumble also over texts which call them *sentences* (hence the word sentential logic as alternative name for propositional logic). Later we we also look at formulas of first-order logics and of other logics. The syntax of propositions we use is given as follows:

$$
\begin{array}{lllll}
\varphi & ::= & & \text{propositions} & (2.1)\\
 & & p \mid \top \mid \bot & \text{atomic propositions} & \\
 & \mid & \varphi \land \varphi \mid \neg\varphi \mid \varphi \rightarrow \varphi \mid \ldots & \text{compound propositions} &
\end{array}
$$

One can distinguish between atomic propositions and compound ones. There are two "constant" atomic propositions, called $\top$ and $\bot$ (or "true" and "false"). Constant in the sense that their meaning is fixed, resp. will be fixed in the obvious manner, indicated by the "names" of the symbols, as soon as we come to the semantics of formulas. Not fixed is the interpretation of p, representing *propositional variables*.[3] For propositional variable, we use p_1, p', q ... as typical elements.

Compound propositions are constructed using logical operators or connectives. We assume that they are a familiar. They include binary connectives like conjunction and disjunction $\land$ and $\lor$, negation $\neg$ as unary one, implication $\rightarrow$ as another one.

One could include more, like "exclusive or" $\oplus$ or equivalence or bi-implication $\leftrightarrow$, etc., but we are here not overly obsessed here by fixing the exact selection of logical operators. The grammar from equation (2.1) contains "...". One may also find presentations leaving out syntax for operators that can be expressed by others, so that the syntax becomes *minimal*: no operator can be removed without crippling the expressiveness of the logic. Out shown syntax is not minimal. For example $\land$ can be expressed using $\neg$ and $\land$, and actually also the atomic propositions $\top$ and $\bot$ are not strictly needed and could be expressed likewise by $\neg$ and $\land$, for instance.

Especially in HW, for logical circuits, one may base the hardware one just *one* logical operator, "NAND" ("NOR" works likewise). That has advantages for the manufacturing process, and NAND can express all other operators, including the propositional constants $\top$ and $\bot$. For the purpose, we consider that as minor details resp. we rely on that the reader knows such basic things and can handle it. As long as we have prositional constants and

[3]One can stumble also over texts, where the p's are called propositional constants, or also propositional symbols. That's just terminology and does not change a thing. Even if called propositional constants, they are less constant than $\top$ and $\bot$, insofar that the truth value of $\top$ and $\bot$ is absolutely fixed, whereas for the p's, the truth-status needs to chosen. To call the p's propositional constants can also be justified, it's not unreasonable, but as said, it's just terminology and let's not loose sleep over that.

operators like the ones from equation (2.1) and their standard interpretation as semantics), it's propositional logic.

As common, at least in computer science, the syntax is given by a grammar. More precisely here, the context-free grammar from equation (2.1) uses BNF-notation. It's a common compact and precise format to describe (abstract) syntax, basically syntax trees. The word "abstract" refers to the fact that we are not really interested in details of actual concrete syntax for text files used in a computer. There, more details would have to be fixed to lay down a presise computer-parseable format. Also things like associativity of the operators and their relative precedences and other specifics would have to be clarified.

But that is "noise" for the purpose of a written text and human communication. A context-free grammar *is* precise, if understood as describing *trees*, and following standard convention we allow parentheses to disambiguate formulas if necessary or helpful. That allows to write $p_1 \wedge (p_2 \vee p_3)$, even if parentheses are not mentioned in the grammar. Also we sometimes rely on a common understanding of precedences, for instance writing $p_1 \wedge p_2 \vee p_3$ instead of $(p_1 \wedge p_2) \vee p_3$, relying on the convention that $\wedge$ binds stronger than $\vee$. We are not overly obsessed with syntactic details, we treat logic *formal* and *precise* but not *formalistic*. Tools like theorem provers or model-checkers would rely on more explicit conventions and *concrete* syntax.

2.2.3 Semantics: the meaning of propositions

So far we have fixed only the notation of formulas, their syntax. If one has been in contact with some form of formal logics at all, one will probably have an grip on what propositions are supposed to represent. Like that $\wedge$ represents "and" and $\vee$ represents "or". But it may be less clear, especially when studying such things for the very first time, though I assume that for participants of the course this is not the case. For example, in an English sentence "do you want tea *or* coffee?", the colloquial "or" is typically not meant as the disjunctive operator of propositional logics, written $\vee$. That sentence is probably meant as given a choice between tea or coffee but not both, and asking "do you want either tea or else coffee?" sounds borderline impolite and "do you want tea xor coffee?" incomprehensible...

At any rate, we are better off with fixing the meaning in some unambigous way. To ask what a proposition means, is to ask "is it true or else false"? So ultimately, a proposition is mapped to one of exactly two possible value, the two *truth values* or Boolean values. Let's use $\top$ for true and $\bot$ for false, and call the two-valued Boolean domain by the notation $\mathbb{B} = \{\top, \bot\}$. To rub it in: we use $\top$ and $\bot$ as propositional constants in the *syntax*, and the values $\top$ and $\bot$ as ingredient of the semantics or meaning, and unsurpisingly, two case of fixing the semantics of propositions is that $\top$ is interpreted as $\top$ and $\bot$ as $\bot$. Actually it will be an inductive definition over the syntax of propositions, and the two cases will be *base cases* in that inductive definition, dealing with two *atomic* propositions. The definition for compound proposition will be done by, obviously, the *inductive cases* of the definition.

Besides $\top$ and $\bot$, there is one base case which we have not covered, namely how to treat propositional variables p. Since we assume that there is a reservoir of propositional variables P, one should more correctly say, there's one base case per variable.

Anyway, since propositions will generally contain propositional variables the truth status of the whole proposition depends on which truth value are assumed or chosen for the propositional variables.

Choosing those is a mapping or function from propositional variables to trues values. We can call such a function a *(propositional) variable assignment* and let's use the σ for those, i.e. a variable assignment is of the following type:

$$\sigma : P \to \mathbb{B} \ . \tag{2.2}$$

With that in place, we can finally define the semantics of a proposition, namely relative to a propositional assignment. One typical notation for the semantic function are the "semantic brackets" $[\![]\!]$; the notation is also used for semantic functions in other context. Anyway, given a variable assignment σ from $P \to \mathbb{B}$ and a proposition φ from Φ, $[\![\varphi]\!]^\sigma$ is a boolean value from $\mathbb{B}$. That makes here $[\![\_]\!]^-$ a function of the following type (assuming, for no particular reason, the variable assignment as the first argument):

$$[\![\_]\!]^- : (P \to \mathbb{B}) \to \Phi \to \mathbb{B} \ . \tag{2.3}$$

When the semantic functions gives back the truth value $\top$, i.e., if

$$[\![\varphi]\!]^\sigma = \top \ , \tag{2.4}$$

we say, proposition φ is true under the variable σ, or φ holds under said variable assignment. One hears also the formulation that the variable assignment σ *satisfies* φ. Notationaly, the latter formulation would often rather written as

$$\sigma \models \varphi \tag{2.5}$$

instead of the formulation from equation (2.4).

$\models$ in the latter equation is the the (semantic) *satisfaction relation*, in this case the satifaction relation for propositional logic. Other logics have other semantic functions resp. other satisfaction relation.

Both formulaic representations and wordings are, of course, completely equivalent, and which to use is a matter of preference. We will use both, presumably.

Equation (2.5) as well as (2.4) cover the "positive" case: the formula or prosition is true, it holds, the assignment satisfies the proposition etc. The negative case means $[\![\varphi]\!]^\sigma = \bot$, resp. $\sigma \not\models \varphi$.

That there is such a clear-cut split in satisfaction or else non-satisfaction, in a proposition being true or else not, in which case the proposition is false, that is characteristic for *classical* propositional logic (or more generally also for first-order logic or classical higher-order logics). We don't cover non-classical propositional or first-order logics. The most famous and important alternatives would be *intuitionistic* variants. One thing that changes for instance for propositional logics is that is no longer the case that a proposition is false if and only if it's negation is true and vice versa. Classically, a proposition (given an choice for the propositional variables) is true or else false, there is no other thing beside true or else false. In Latin, that principle is called "tertium non datur", there is no third thing besides true or false. For intuitionistic logics, that's dropped, resp. is not the case. Without going into details that complicates the way the semantics of propositions is given.

Indeed, for our classical propositional, we have not *actually* given the semantics. We have just discussed different symbols that are typically used for it ($[\![\_]\!]_-$ and $\models$) and we have given types, in equation (2.3). We leave the exact definition as an easy exercise to the reader. We have mentioned that it ultimately it's an inductive (or "recursive") definition that needs to fix the base cases of the syntax from equation (2.1). The inductive cases have to cover all the non-atomic logical connectors, defining the truth value of a compound proposition in terms of the truth values of the sub-propositions. For example, $\varphi_1 \wedge \varphi_2$ is true of both φ_1 and φ_2 are true, and false otherwise. A definition like that would correspond to a straightforward recursive procedure.

A compact way to show all the neccessary cases for all connective is to arrange them in tabular form, a so-called *truth table*. We show the truth table for conjunction as one example in Table 2.1.

		$\wedge$.
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

Table 2.1: Truth table for conjuction

Conjuction is a binary connective, i.e., it has two "inputs". The table contains this four lines, covering the for combinatins of $\top$ and $\bot$ for the pairs of input. The last columns shows the output of the conjunction, thus fixing its semantics. As said, the remaining constructors are left out here.

2.2.4 Proof theory

In the previous section we fixed the meaning of propositions by defining the semantic function resp. defining $\sigma \models \varphi$. We called σ a variable assignment, associating truth values to propositional variables. One could also call it a *model* and use for $\sigma \models \varphi$ the words "σ models φ" instead of "σ satisfies φ". We will later encounter the notion of "model" also for first-order logic (first-order model) and for other logics. There, not surprisingly, the notion of model becomes more complex and interesting. For propositional logic, it may sound a bit too extravagant to speak of a model of a proposition. But its correct terminology.

We mention it, because it's related to "model theory". That's dealing with questions concerning the models for a logic. What are the models? If a formula has an infinite model, is there also a finite one? And more questions like that. It's likewise not a coincidence that the word model is mentioned in *model checking*. Model checking is not about general questions like the ones mentioned, it's more concretely about "given a model on the one hand and a formula on the other, does the model satisfies the formula?". For propositional logic, to check

$$\sigma \models^{?} \varphi \tag{2.6}$$

is trivial and can be done efficiently; as indicated, the semantic function $[\![\_]\!]$ is a recursive procedure, so one just have to plug in the arguments, calculate $[\![\varphi]\!]^{\sigma}$ and look at the outcome. So model checking is a non-problem for propositional logic and actually, the more general question of what can be said about models *in general* ("model theory") is boring, as well. Model theory get's more hairy for first-order logic or logics more expressive than propositional logic, but we will not cover that. We will also don't do model checking for first-order logic, but later for modal logics.

In connection with propositional logic, another question is of relevance and is a challenge, that's the question if there *exists* a model or variable assignmnt that satisfies a proposition. One could formulaically represent that as

$$? \models \varphi \tag{2.7}$$

and that's a question of *satisfiability*. For propositional logic, it's the famous SAT problem and corresponding tools are sat-solvers or more generally constraint solvers. Pure sat-solving means constraint solving for Boolean constraints (and Boolean constraint is just another name for a proposition in propositional logic intended to be checked for satisfiability).

Back to the real issue in the section, which is neither model theory nor model checking, but proof theory. Proof theory is kind of like the opposite of model theory. Model theory is about what can be said about the semantics (the models), proof theory is about what can be proven in a logic in principle. That's a theoretical question, a more more practical one is how to prove things efficiently. So one may paraphrase that by saying that model theory is concerned with when something is "true" and proof theory is about when something "provable". These questions of course hang together; the whole purpose of doing a proof is to establish truth. Ideally, of course, all true things should be provably so, and likewise all false things should be demonstrably false as well (refutable). In such a situations truth and provability coincide. Howerer, that's rarely the case. That has to do with liminations of computability; performing a proof is ultimately a computation process and already for first-order logic, one runs into undecidability.

Proof theory (same for model theory) is concerned with *meta* theory, it's *about* a given logic and its proof procedures but it's not formulated *inside* the logic at hand. For instance, when doing a statement about what can be proven in propositional logic, that's not a propositional formula, it's perhaps an English sentence, mentioning a particular proof procedure, which likewise is not given as propositional formula. Propositional logic would anyway be too weak to capture any of that.

Let's get more concrete: how to do proofs for propositional logic? The question is impresise, it depends on what one intends to prove. But ultimately all plausible things one could be interested in are *decidable*.

As mentioned, the "model checking" question from equation (2.6) is efficiently decidable (and uninteresting). Finding a satifying variable assignment (equation (2.7)) is likewise decidable. We simply have to choose $\top$ and $\bot$ for all propositional variables inside the given proposition and check the result. If we find a satisfying σ, φ is satisfiably, and, after checking all combinations, we have found not, it's not satisfiable. It's like making a giant truth table. Since there are only finitely many relevant σ to check, since the proposition contains only finitle many variables, SAT is decidable. The problem with that *brute force* method is the combinatorial explosion of checking different combinations of truth values. Unfortunately, theoretically speaking, from the perspective of complexity theory, we have to live with that. There is not much one can do, Boolean satisfiability or SAT is known to be *NP-complete* (actually, it's the prototypical NP-complete problem. When claiming that not much can be done, that's from a theoretical point of view. Practically, many things can be done and have been done to make SAT-solvers and other forms of constraint solvers to tackle larger and larger problems.

Another thing one could be interested in is not satisfiability, but *validity*: is a proposition true for *all* variable assignments. Such a *valid* propositional formula is also called a propositional *tautology*. One can also solve that by checking out all possible variable assignment σ for the involved propositional variables.

Just filling out a giant truth table in a brute force manner is a proof procedure, only not a very elegant one. So, doing proofs for boolean satisfiabilty or validity can be done smarter, looking for specific strategies, or finding "short-cuts". What I mean by short-cut is pretty common sense, like if on processes a conjunction $\varphi_1 \wedge \varphi_2$ and the evaluation strategy has determined that φ_1 is $\bot$, then there is no purpose to find out all different evaluations for φ_2.

Working hard on evaluation strategies has led to different techniques and proof methods (David-Putnam and variations, resolution and on and on) and while in general, SAT is still NP-hard, such strategies (and smart data representations) have lead to quite powerful SAT solvers or other boolean prover implementations.

To round off this section, let's have a look at *satisfiability* and *validity*. Those two notions in classical forms of logics, are two sides of a coin: a formula is valid iff its negation is unsatisfiable. Formulated contra-positively, in terms of non-validity and satisfiability, it means

$$\not\models \varphi \quad \text{iff} \quad \sigma \models \neg\varphi \quad \text{for some } \sigma$$

The model σ for the negation $\neg\varphi$, refuting validity of φ, is also called *counter-example*. Model-checking is not doing satisfiability or validity, but we will encounter also there the concept of counter-examples, which is analogous to the one discussed here. It's a valuable feature of a program verification method to be able to give back counter examples. The information that a program does not satisfy its specification and requirements is not very helpful. Such a failure needs to be accompagnied with information what and where the problem is, and that's called the *counter example*. It's a model or example that shows that the negation of the property of interest is satified. Since model checking is not validity or satisfiability checking, that idea needs some porting to work with, for instance, temporal logic model checking, but the concept is similar.

2.3 Signatures, terms, and substitutions

As far as logics are concerned, we dealt so far, in Section 2 with a most simple form of logic, classical propositional logic. It's a bare form of logics, insofar it does not speak *about* anything. It is a logic in most bare form, regulating truthness and falseness, without referring to anything particular which is being true or false. So, in that sense, it is a very sterile form of logic.

First-order logic, covered later in Section 2.4, allows to speak and reason about "things". In propositional logic, one could have a propositional variable *even*, but that's just a name, like p or q. Those propositional variables, which count among atomic propositions are either true or false in classical propositional logic, but there is no "evenness" about the variable or something that is being even or not. In first-order logic or other more expressive logic, one could formulate things like $\mathit{even}(x)$, where *even* now is called a *predicate*. First-order logic is also called more explicitly first-order predicate logic. The predicate *even* is still just a name and in that sense there is also no "evenness" about it as scuh, but now it speaks *about* something (and one can try to axiomatize things about that name so that it captures "evenness"). In $\mathit{even}(x)$, x is meant as variable, plausible representing natural numbers. And depending on which natural number x represents, the even-predicate is intended to be true or false. When one intends to speak about numbers, one also need syntax for that (not just variables). I.e., one would need syntax for natural number (like 0, 1, ...), and operations on them (like addition and multiplication two be able to write expressions like $2 \times (x + 1)$. In combination with the even-predicate, one would like that $\mathit{even}(1 + 1)$ evaluates to $\top$, $\mathit{even}(2 \times 5 + 1)$ to $\bot$, and the truth status of $\mathit{even}(x + 1)$ dependent on the choice of x.

Without further arrangements, as mentioned, there is no evenness about the even-predicate $\mathit{even}(x + 1)$, it's a symbol, in the same way as the proposition called *even* would be in propositional logics. The "arrangement" could be *fixing* the interpretation of *even* and of $+$ etc. That would be a semantics, in a way a model-theoretic approach. Or one could try to specify rules one wishes to hold for a predicate called *even* in combination with other symbols for natural numbers, for instance stipulating as axiom $\forall x.\mathit{even}(2 \times x)$ or $\forall x.\mathit{even}(x + 1) \to \mathit{odd}(x)$ etc. That could provide an axiomatization. And then one could ask; does the axiomatization, the proof-theoretic approach, capture exacly the standard natural numbrs (the model-theoretic way). The answer would be *no*.

But this section is not about how to give meaning to the symbols (either by attaching a model like the natural numbers, nor by trying to axiomatize it). That will be dealt with in the subsequent Section 2.4 about first-order logic.

In this section we are concerned with the symbols themselves, the syntactic aspects of first-order logic, bare any meaning. The domain specific concept of syntactical material for first-order logic is called (first-order) signature. It consist of a choice of *functional* and *relational* symbols, each with a given *arity* (or more generally with given sorts). For instance, when working with natural numbers, the symbol $+$ represents a *binary* operation, i.e., is of arity 2, and a relation or predicate like $\leq$ is likewise of arity 2.

2.3.1 Signatures

The signature is domain-specific, since first-order logic has also logical syntax, including the operators from propositional logics like conjunction, negation, etc.

Definition 2.3.1 (Signature (single-sorted)). A *signature* $\Sigma = (\Sigma_f, \Sigma_{rel}, ar)$ consists of two finite, disjoint sets of functional and relational symbols together with a function $ar : \Sigma_f + \Sigma_{rel} \to \mathbb{N}$, fixing the *arity* for each symbol.

A signature with functional symbols only is called **algebraic** signature.

Abstractly, one could write $\Sigma_f = \{f_1^{(2)}, f_2^{(0)}, g^{(1)}, \ldots\}$ and $\Sigma_{rel} = \{R_1^{(1)}, R_2^{(2)}, Q^{(2)}, \ldots\}$, indicating the arity as superscript.

In this lecture, we often leave the signature implicit or in English, and don't operate with Definition 2.3.1 in its full glory. For instance, when using natural numbers, we might use (and have used) symbols like $+$ and $\leq$, both of assumed arity 2, without writing $\leq^{(2)}$ or $+^{(2)}$ or explicating ar as function.

Functional symbols of arity 0 are also called *constant* symbols. For the natural numbers 0 is an example of a constant symbol.

Relational symbols of arity 0 are like the propositional symbols in propositional logic (we called them mostly propositional variables). Indeed, a degenerated signature with no functional symbols and only 0-arity relational symbols corresponds to propositional logic.

The concept of a signature from Definition 2.3.1 is a bit limited, at least in practice. An arity of a symbol is the number of arguments the symbol takes. That implies that all arguments are of the same type. A type in our context is commonly not called type, but sort. Definition 2.3.1 assumes that there is only one sort (left implicit), and that's why it's called a single-sorted signature. When working with such signatures, for instance in connection with first-order logics, that's too restrictive to be useful in many cases.

One wants the possibility to work and reason about different data structures at the same time. That then leads to *many-sorted* signatures. If multi-sortedness is what's needed in practice, why does one bother with single-sorted signatures (and single-sorted first-order logics) at all and why do many texts focus on the single-sorted case? Well, it's mostly a theoretician's thing. When studying, say, first-order logics, like studying its proof theory or model theory, basically all interesting, fundamental results work the same in both settings. The many-sorted case only leads to a slightly notational overhead ("given sorts $s_1, s_2, \ldots s_n$ with $n \geq 1$, blablabla"), which one is happy do do without when exploring properties of the logics, working on the logic's meta-theory, perhaps stating that the results *carry over* straightforwardly to the many-sorted case.

In practice, of course, specifying and verifying properties of a system or a program, one wants to be able to work with natural numbers and lists and whatever is relevant in the concrete setting.

So sorts are symbols representing different domain, like `nat` or `bool` or whatever we need. We don't use (for here) the notation $\mathbb{N}$ or $\mathbb{B}$, since for now we stress that sorts like `nat` and `bool` are *syntax*, symbols form the signature, whereas $\mathbb{B}$ and $\mathbb{N}$ are the semantics, i.e., the boolean values resp. the natural numbers. The intention is of course that `nat` is interpreted by the domain $\mathbb{N}$ etc., but associating meaning to syntax is the task of the semantics.

One could then work with integers and integer list and use a symbols like `cons` in the functional part of the signature with the type $\texttt{int} \times \texttt{List}_{\texttt{int}} \to \texttt{List}_{\texttt{int}}$. We don't give a many-sorted version of the concept of signature from Definition 2.3.1. We hope it's clear enough how it works and invoke the excuse for not doing it, namely everything "just carries over".

In a many-sorted setting we then have some form of type system. Actually, already in the single-sorted case we have one. A term $\texttt{cons}\ (0,0)$ is equally ill-typed as is $+0$ in a single-sorted setting, where $+$ needs two arguments, not one. At any rate, even when working with a large number of different sorts, the type discipline is fairly trivial. Therefore we won't bother to give a formal definition of well-sorted terms. We simply assume that we only work with well-typed expressions. For a programming language, one does not waste time to think what an ill-typed program means, one cannot run it anyway, and we won't waste time here to consider formulas of ill-sorted terms.

Many-sorted signatures have many *sorts*, like `nat`, or $\texttt{List}_{\texttt{int}}$. Why are those things are not called types, but sorts? Well, here and there, one finds also the word type for those. And for all intent and purposes, they are types. Still, more common is the sort-terminology for algebraic and first-order signatures. Perhaps that has historical reasons. It has also to do with the fact that, as far as type systems are concerned, the discipline is very restricted. People dealing with type systems would laugh at it and feel ashamed (and feel better when pointing out that it's just a sort system, not a grown-up type system…). The restriction is that one has a number of sorts, which corresponds to atomic base types, like the mentioned `nat` and $\texttt{List}_{\texttt{int}}$ and the operators like $\texttt{cons}\colon \texttt{nat} \times \texttt{List}_{\texttt{int}} \to \texttt{List}_{\texttt{int}}$, where $\texttt{nat} \times \texttt{List}_{\texttt{int}} \to \texttt{List}_{\texttt{int}}$ can reasonably be called the type of `cons`. But for function symbols of the signature, only types of the following form are allowed

$$s_1 \times \ldots s_2 \to s$$

If we call that a type, theb s_i and s are *not* also types, but something else (namely sorts). This restricting disallows for instance that `cons` is of type $\texttt{int} \to (\texttt{List}_{\texttt{int}} \to \texttt{List}_{\texttt{int}})$. It some sense, that type is equivalent to $(\texttt{int} \times \texttt{List}_{\texttt{int}}) \to \texttt{List}_{\texttt{int}}$, but it's simple not within the framework of algebraic or first-order signature. Likewise on can not have a symbol for a map-like function of type $((\texttt{int} \to \texttt{bool}) \times \texttt{List}_{\texttt{int}}) \to \texttt{List}_{\texttt{bool}})$. A map-function is a *higher-order* function; it takes another function as argument. That's not doable in algebraic signature, and there are good reasons for that. Algebraic signatures are not just play a role for first-order logics, but the field of (general) algebras uses them. And the core theory and central results of (general) algebra simply rests on that syntactic restriction. Without it's no longer (general) algebra.

For the lecture, we will look not too deep at first-order logic, but don't explore "algebra" (which can be seen as a quite restricted form of first-order logic with only a functional signature and considering *equations* only. So it's a form of equational logic and in that sense, there is one and only one relational symbol, namely "=", but since that's fixed, one does not bother to introduce a Σ_{rel}-signature just for that.

When pointing out that higher-order functions like map are not allowed in algebraic and first-order signatures, we should avoid a misconception that could suggest itself at this point. Namely that higher-order functions would be covered by higher-order logic, which goes beyond first-order logic. That's a misconception. Higher-or-not-order-ness of predicate logic refers to about what can be quantified over. For instance, second-order predicate logic is allowed to quantify over predicates, first-order logic cannot. Third order predicate logic can quanify over predicates over predicates or sets of setc. etc. That's not the same as the question what the predicates ultimately speak over. In logics with algebraic signatures of the form introduced here, predicates range over elements from the domains of the given sort, and they don't range over functions.

Of course there are logics that can deal with higher-order function and are higher-order wrt. to the quantification over predicates. Often that are logics in connection with typed λ-calculi and phrased often as (intuitinistic) dependent type theory. There are also quite a number of theorem provers based on dependent type theories, like Isabelle/Hol, Coq, and others. Anyway, higher-order aspects won't be covered in our lecture, mostl probably.

2.3.2 Terms

Closely related to (algebraic) signatures is the notion of *terms*. That's nothing else than expressions formed with the syntactic material from a given signature, i.e., constructed using the given function symbols. Additonally, one has variables available, i.e., a given countably infinite setn X (and we assume typical elements like $x, y', \ldots$ as variables). For terms, the relational part of a first-order signature is not used, only the functional part Σ_f (also called an algebraic signature).

Definition 2.3.2 (Terms (single-sorted))**.** Given an algebraic signature Σ and a (countably infinite) set of *variables* X, then the set of *terms* over Σ and X, written $T_\Sigma(X)$ is given by the following grammar.

$$\begin{array}{rcll} t & ::= & x & \text{variable} \\ & \mid & f(t_1, \ldots, t_n) & f \text{ of arity } n \end{array} \tag{2.8}$$

The set of *ground* terms over Σ is given as $T_\Sigma(\emptyset)$ (also written as T_Σ).

The terms from Definition 2.3.2 are *single-sorted* and based on the single-sorted version of signatures from Definition 2.3.1. The definition requires that terms must be "well-typed" or "well-sorted" in that a function symbol that expects a certain number of arguments (as fixed by the signature in the form of the symbol's arity) must be a applied on exactly that number of arguments. The number n can be 0, in which case the function symbol is also called a *constant* symbol. In the straightforward many-sorted generalizations, terms, by definition would likewise be required to respect the sorts. As mentioned earlier, the multi-sorted setting is not really different, it does not pose fundamentally more complex challenges (neither syntactically nor also what proof theory or models or other questions are concerned).

As a simple example: with the standard interpretation in mind, a symbol `zero` would be of arity 0, i.e., represents a constant, `succ` would be of arity 1 and `plus` of arity 2. For clarity we used here (at least for a short while) `typewriter` font to refer to the symbols of the signature, i.e., the syntax, to distinguish them from their semantic meaning. Often, as in textbooks, one might relax that, and just $+$ and 0 for the *symbols* as well.

In practical situations (i.e., tools), one could allow *overloading*, or other "type-related" complications (sub-sorts, for example) for the sake of convenience. Also, in concrete syntax supported by tools, there might be questions of *associativity* or *precedence* or whether one uses *infix* or *prefix* notations. For us, we are more interested in other questions, and allow ourselves notations like x **`plus`** y or $x + y$ instead and similar things, even if the grammar seems to indicate that it should be **`plus x y`**. Basically, we understand the grammars as *abstract syntax* (i.e., as describing trees) an assume that educated readers know what is meant if we use more conventional concrete notations.

2.3.3 Substitutions, in particular term substitutions

A central notion in connection with terms is the concept of *substitution*. Actually, it's not just important for terms, but plays a role in many situations which involve syntactic constructions containing variables. But let's focus for now on terms.

To substitute means to replace something. Substitutions are meant to *replace* variables occurring in a term by terms. At its core, a substitution is defined as mapping from variables to terms, expressing said replacment.

Definition 2.3.3 (Term substitution (single-sorted))**.** Given terms $T_\Sigma(X)$ over a signature Σ and using variables from X, a *substitution* θ is a mapping of type

$$X \to T_\Sigma(X)\ . \tag{2.9}$$

In abuse of notation we use substitutions also on terms, i.e. as mapping of type $T_\Sigma(X) \to T_\Sigma(X)$.

We write θt or just $\theta(t)$ for applying the substitution θ in term t. In the literature, the post-application notation like $t\theta$ is also very common. Note that Definition 2.3.3 does not spell out how to *lift* the substitution function on variables from Equation (2.9) to work on terms. In abuse of notation (as is common), given a mapping θ on variables, we use the same symbol also for terms. In programming-language terms we make use of *overloading*.

Substitution, especially in practice, are *finite* functions in that they replace only finitely many variables. Terms contain finitely many variables, if any. So applying a subtitution only affects the variable occuring in the terms under consideration, so from that point of view, there is no real use for substitutions affecting inititely variables. For notation for finite substitution, we use $[t/x]$ for replaceing x by t, or $[t_1, t_2/x_1, x_2]$ for replacing x_1 and x_2, etc. As an example, $[1+z/x]((x+(y*x)))$ gives $(1+z)+(y*(1+z))$.

That's enough for the moment concerning substitutions, it should be sufficiently clear and/or sufficiently known. We have defined substitutions on terms. We will later use substitution also on first-order formulas (actually, the concept of substitution makes sense everywhere if one has "syntactic expression" with "variables"): formulas will contain, besides logical constructs and relational symbols also variables and terms. The substitution will work the same as here, with one technical thing to watch out for (which is not covered right now): Later, variables can occur *bound* by quantifiers. That will have two consequences: the substitution will apply only to not-bound occurrences of variables (also called *free* occurrences). Secondly, one has to be careful: a naive replacement could suffer from so-called *variable-capture*, which is to be avoided (but it's easy enough anyway).

2.3.4 First-order signature (with relations)

So far we have focused on *algebraic* signatures, the part Σ_f of the signature from Definition 2.3.1 used for terms. In first-order logic, the signature, besides function symbols, contains also *relational* symbols Σ_{rel}. Those are intended to be interpreted "logically". For instance, in a single -sorted case, if one plans to deal with natural numbers, one needs relational symbols on natural numbers, like the binary relation `leq` (less-or-equal, representing $\leq$) or the unary relation `even`. One can call those relations also **predicates** and they form later then the *atomic* formulas of the first-order logic (also called (first-order) predicate logic). When unspecific and talking generally, we use letters like P, Q' etc. as typical elements of Σ_{rel}. standard binary symbol: $\doteq$ (equality)

Further side issues and remarks

Remark 2.3.4 (Sort for Booleans)**.** The above presentation is for the single-sorted case again. The multi-sorted one, as mentioned, does not make fundamental trouble.

In the hope of not being confusing, I would like to point out the following in that context. If we assumed a many-sorted case (maybe again for illustration dealing with natural numbers and a sort `nat`), one can of course add a second sort intended to represent the booleans, let's call it `bool`. Easy enough. Also one could then think of relations as boolean valued function. I.e., instead of thinking of `leq` as relation-symbol, one could attempt to think of it as a *function symbol* namely of sort `nat` $\times$ `nat` $\rightarrow$ `bool`. Nothing wrong with that, but one has to be careful not confuse oneself. In that case, `leq` is a function symbol, and `leq(5,7)` (or `5 leq 8`) is a term of type `bool`, presumably interpreted same as term `true`, but it's not a predicate as far as the logic is concerned. One has chosen to use the surrounding logic (FOL) to speak about a domain intended to represent booleans. One can also add operator like `and` and `or` on the so-defined booleans, but those are *internal* inside the formalization, they are *not* the logical operators $\wedge$ and $\vee$ that part part of the logic itself. □

Remark 2.3.5 (0-arity relation symbols)**.** In principle, in the same way that one can have 0-arity function symbols (which are understood as constants), one can have 0-arity relation symbols or predicates. When later, we attach meaning to the symbols, like attaching the meaning $\leq$ to `leq`, then there are basically only two possible interpretations for 0-arity relation symbols: either "to be the case" i.e., true or else not, i.e., false. And actually there's no need for 0-arity relations, one has fixed syntax for those to cases, namely "true" and "false" or similar which are reserved words for the two only such trivial "relations" and their interpretation is fixed as well (so there is no need to add more alternative such symbols in the signature).

Anyway, that discussion shows how one can think of propositional logic as a special case of first-order logic. However, in boolean logic we assume many propositional symbols, which then are treated as *propositional variables* (with values true an false). In first order logics, the relational symbols are not thought of as variables, but fixed by choosing an interpretation, and the variable part are the variables inside the term as members of the underlying domain (or domains in the multi-sorted case).[4] $\square$

Remark 2.3.6 (Equality)**.** The equality symbol (we use $\doteq$) plays a special role (in general in math, in logics, and also here). One could say (and some do) that the equality symbol is one particular binary *symbol.* Being *intended* as equality, it may be captured by certain laws or axioms, for instance, along the following lines: similar like requiring `x leq x` and with the intention that `leq` represents $\leq$, this relation is *reflexive*, one could do the same thing for equality, stating among other things `x eq x` with `eq` intended to represent equality. Fair enough, but equality is *so central* that, no matter what one tries to capture by a theory, equality is at least *also part of the theory*: if one cannot even state that two things are equal (or not equal), one cannot express anything at all. Since one likes to have equality anyway (and since it's not even so easy/possible to axiomatise it in that it's really the identity and not just some equivalence), one simply says, a special binary symbol is "reserved" for equality and not only that: it's agreed upon that it's *interpreted* semantically as equality. In the same way that one always interprets the logical $\wedge$ on predicates as conjuction, one always interprets the $\doteq$ as equality.

As a further side remark: the status of equality, identity, equivalence etc is challenging from the standpoint of *foundational* logic or math. For us, those questions are not really important. We typically are not even interested in alternative interpretations of other stuff like `plus`. When "working with" logics using them for specifications, as opposed to investigate meta-properties of a logic like its general expressivity, we just work in a framework where the symbol `plus` is interpreted as $+$, end of story. Logicians may ponder the question, whether first-order logic is expressive enough that one can write axioms in such a way that the only possible interpretation of the symbols correspond to the "real" natural numbers and plus thereby is really $+$. Can one get an axiomatization that characterizes the natural numbers as the *only* model (the answer is: *no*) but we don't care much about questions like that. $\square$

[4]Strictly speaking, unless one has an empty signature, one one not even *need* true and false a pre-arranged standard atomic proposition, as one could define the true case as `x leq x` $\vee\neg$ `x leq x` (if one had for example `leq`). Of course, no one wants to do like that (though for reasons of economy, some representations only introduce maybe true, knowing that false can be represented by the negation, same for omitting $\vee$ if one has $\wedge$ and $\not\leq$).

2.4 First-order logic

In this section, we briefly cover first-order logics. It's not an in-depth discussion, since the lecture is mostly concerned with other forms of logics, like temporal logic. Still, like propositional logics, predicate logic is basic knowledge, so we at least cover a possible syntax and discuss how that logic is interpreted, i.e., speak shortly about semantics and a bit of proof theory, as well.

As said, we are mostly interested temporal logics or other logic. But that's actually a bit orthogonal. Temporal logic will have *temporal* connectives, but also standard, non-temporal logical operators, like conjunction or implication. Taking propositional logic as core, adding temporal or modal operators, leads to *propositional* temporal or modal logic, but one can add such operators to first-order logic, as well ... So it can't hurt to have a short look at or get a short reminder of first-order logics.

2.4.1 Syntax

Definition 2.4.1 (Formulas of first-order logics)**.** Given a signature Σ, the formulas of first-order logic are given by the following grammar.

$$\begin{array}{rcll} \varphi & ::= & P(t,\ldots,t) \mid \top \mid \bot & \text{atomic formulas} \\ & \mid & \varphi \wedge \varphi \mid \neg\varphi \mid \varphi \rightarrow \varphi \mid \ldots & \text{formulas} \\ & \mid & \forall x.\varphi \mid \exists x.\varphi & \end{array} \tag{2.10}$$

We use φ for formulas of first-order logic, the same symbol we used for the formulas of propositional logic (aka propositions) from equation (2.1), and we also will use the same symbol for formulas of later logics.

As before for propositional logic, we silently assume proper priorities and associativities (for instance, $\neg$ binds by convention stronger than $\wedge$, which in turn binds stronger than $\vee$ etc.) In case of need or convenience, we use parentheses for disambiguation.

The grammar, choice of symbols, and presentation (even terminology) exists in variations, depending on the textbook. It's often convenient to work with a sorted or typed variant, using for example syntax like $\forall x{:}Nat.\varphi$, which does not change much from a logical point of view.

The above definition, as we did in the propositional case, is a bit generous wrt. the offered syntax. One can be more economic in that one restricts oneself to a *minimal* selection of constructs (there are different possible choices for that). For instance, in the presence of (classical) negation, one does not need both $\wedge$ and $\vee$ (and also $\rightarrow$ can be defined as syntactic sugar). Likewise, one would need only one of the two quantification operators, not both. Of course, in the presence of negation, $\top$ can be defined using $\bot$, and vice versa. In the case of the boolean constants $\top$ and $\bot$, one could even go a step further and define them as $P \vee \neg P$ and $P \wedge \neg P$ (but actually it seems less forced to have at least one as native construct). One could also explain $\top$ and $\bot$ as propositions or relations with arity 0 and a fixed interpretation. All such representations can be found here and there, but they are inessential for the nature of first-order logic and as a master-level course we are not loosing sleep over representational questions like that. Of course, if one had to interact with a tool that supports, for instance first-order logics or a fragment or extension thereof, (like a theorem prover or constraint solver) or if one wanted to implement such a tool oneself, syntactial questions would of course matter and one would have to adhere to stricter standards of that particular tool.

2.4.2 Semantics: the meaning of first-order logic formulas (semantics, interpretation, models . . .).

After fixing the syntax, we have to address what formulas mean. It's analogous to what we did for propositional logic (and later for other logics). Conceptually, the way the semantics is defined is analogous to the situation for propositional logics. The syntax is given by a grammar, and the semantics is a mapping of syntax trees to a semantical "domain". In the propositional setting, the mapping was given by assigning truth values to the propositional syntax, and that mapping was lifted to propositions.

For first-order logics, the situation gets a bit more involved. There are two syntactic levels. There is the level of functional and relational symbols, with which one can form terms over a given signature and relations between terms or predicates on terms.

Besides that level, there's the logical level, which needs to be interpreted as well. Most of that part is the same as for the propositional level (for instance, the symbol $\land$ is conjuction, as before etc.) The quantifiers are new of course, compared to propositional logics.

In the following we assume a given signature Σ, and we focus on the single-sorted case, with the excuse that, as mentioned, it does not make a difference, in theory. A first-order structure, in the single-sorted case, is simply some set of elements together with functions and relations on it.

For illustration, let's take the familiar natural numbers $\mathbb{N}$. The set $\mathbb{N}$ in itself does not qualify as structure, for that, one needs additional constants, functions, and predicates or relations on that set or domain of the structure. A typical selection could be the following:

$$(\mathbb{N}; 0, \lambda x.x + 1, +, \times, \leq, \geq) \tag{2.11}$$

Here, the domain $\mathbb{N}$ is equipped with 4 functions (one of which is 0, which is of zero arity and this called usually a constant rather than function) and two binary relations $\leq$ and $\geq$. Never mind the λ-notation, it's just meant as a function that takes one argument from the domain and returns its successor.

Structures like that can be used to give meaning to first-order signatures. That's done by associating to each *function symbol* of a given arity a mathematical *function* of the same arity, and analogously, for each *relational symbol* or predicate symbol of a given arity. a matheamtical relation of fitting arity. The association is a mapping for which we use I, the interpretation function or interpretation for short, and a model is the pair consisisting of a domain A together with the interpretation function (and implicitly the signature, assumed given).

Definition 2.4.2 (First-order Σ-structure)**.** Assume a first-order signature Σ. A first-order *structure* M for Σ is a tuple

$$M = (A, I) \tag{2.12}$$

where A is a set (the domain) and I (the interpretation), maps functional symbols function f such that $I(f) : A^n \to A$ for each $f \in \Sigma_f^{(n)}$ and relational symbols P such that $I(P) \subseteq A^n$ (for each $P \in \Sigma_{rel}^{(n)}$).

Instead of $I(f)$ and $I(P)$, we mostly write $[\![f]\!]^I$ and $[\![P]\!]^I$ or f^I and P^I. Another name for Σ-structure is Σ-model. As a side remark: The special case of first-order signatures containing no relational symbols, but only function symbols is also called *algebraic* signature (see Definition 2.3.1). In line with that, the corresponding structures (without relations[5] or predicates) are called Σ-algebras.

As before, we introduce the concept for a single-sorted signature and consequently, the Σ-structure has one single domain. The many-sorted case is the obvious generalization. In particular, the interpretation function is required in that case not respect the sorts of the symbols (not just their arity).

Definition 2.4.2 makes as strict separation between syntax on the one hand, and the mathematical structure or model on the other. So, for the illustrative example of natural numbers from equation (2.11), which is the semantical level, one has also a syntactical layer, the signature. So, to make the split more visible, would could write the coresponding signature Σ with one sort say `Nat` and function symbols `zero`, `succ`, `plus` and `times` and relational symbols `leq` and `geq`. So, the `times` is meant as symbol, and $\times$ as the well-know mathematical function of multiplication. The interpretation function would fix $[\![\texttt{times}]\!]^I = \times$, or $[\![\texttt{leq}]\!]^I = \leq$ for one of the relations.

One could also alternatively fix $[\![\texttt{times}]\!]^{I'} = +$, or $[\![\texttt{leq}]\!]^{I'} = \geq$. Sortwise or typewise, that would be just fine, but of course it would be a non-recommended interpretation of symbols like `times` and `leq`.

For most people including mathematicians (excepts perhaps logicians doing model theory or semanticists) all of that is a bit schizophrenic or over-the-top hair-splitting. Interpreting the symbol `times` by $\times$, ok, but isn't $\times$ also a symbol? Sure, in a way yes, still talking about models, interpetations, semantics, etc. there is this split, one symbol (say `times`) is is meant as syntax, and associated with that is a mathematical function (but one needs to say which function it is, or define it, and in this case one could use the symbol $+$ and appeals to the knowledge of the reader to have an understanding what $+$ does, or defining it perhaps semantically, maybe using induction, if one feels the need).

As said, for most it's a bit over the top, and would be content with just working with a structure as in equation (2.11), not bothering to make the split in the two levels explicit.

That may sound nitpicking, but probably it's due to the fact that when dealing with "foundational" questions like model theory, etc. one should be clear what a *model* actually is (at least at the beginning). But also practically, one should not forget that the

[5]Except equality.

illustration here, the natural numbers, may be deceivingly simple. If one deals with more mundane stuff, like capturing real world things as for instance is done in ontologies, there may be hundreds or thousands of symbols, predicates, functions etc. and one should be clear about what means what. Ontologies are related to “semantics techniques” that try to capture and describe things and then query about it (which basically means, asking questions and draw conclusions from the “data base” of collected knowledge) and the underlying language is often (some fragment of) first-order logic.

Giving meaning to variables

After having introduced the notion of Σ-structure, giving an interpretation for the syntactic material of a first-order signature, are we now ready to fix the semantics of first order formulas? Formulas contain of course also logic connectives like $\wedge$ or $\neg$, but that will work analogously to the propositional case. One piece missing is that we have to deal with *variables*. Formulas can contain occurrences of (free) variables, i.e., variables which are not in the scope of a quantifier. The logical status of such an open formula obviously may depend on which values are chosen for the free variables For example, assume an atomic formula like (x `minus one`) `geq zero` and assume the obvious interpretation of the involved symbols on the domain of natural numbers, i.e., the formula represents $(x - 1) \geq 0$. For values of x larger or equal 1, that represents a truth about natural numbers, but for a value of 0 for x, it's false. That should be clear enough, so let's nail it down. Let's call a mapping that assigns values to variables plausibly a *variable assignment*.

Definition 2.4.3 (Variable assignment)**.** A *variable assignment* for variables from X and a domain A is a mapping σ of the following type:

$$\sigma : X \to A \tag{2.13}$$

In Section 2.2, we introduced already variable assignments, there for propositional logics, see equation (2.2). We also used the same symbol for it and conceptually, it's similar, anyway: giving values to variables. In the context of propositional logics, it was propositional variables, here its variables representing values from a semantic domain of values. Note that what what we called propositional variables $p \ldots$ in propositional logic correspond in first-order logics to relational symbol or predicate symbols of arity 0. If one has such 0-arity symbols in the signature, they are not considered variables and they are given their meaning by the *interpretation*, either as $\top$ or $\bot$. That actually means there is no much need for such “constant” predicate symbols or proposition symbols, two are enugh, one interpreted as $\top$ and one as $\bot$ (and even those could be *defined*, for example the one representing false-hood as negation of the one representing truth-ness and truthness as $\varphi \vee \neg\varphi$, choosing some arbitrary formula φ. Our syntax from Definition 2.3.3 provided two keywords $\top$ and $\bot$ (which are like 0-arity predicates) whose interpretation will be fixed to the obvious truth values $\top$ and $\bot$ accordingly.

A variable assignment is sometimes also called *valuation* or also *state*. The latter namely is not so much used when dealing with (first-order) logics as such, but when using such a logic for program verification. There, some variables play the dual role. The program being verified containts typical program variables. Th logic *speaks* about the program, using perhaps first-order logic or a fragment thereof. For instance, first-order formulas could express at a point in a program expectations about the values of some program variables, like *asserting* that, at *that* point, a particular variable is non-negative. This would be an example of a *assertion*. Many languages, for instance Java, offer assertions of that form (but not quantifications, that would be too expressive for the intended purposes of run-time assertions). Properties of program variables are captured by *free* variables in formulas. In a logical setting, values of variables are fixed by variable assignments or valuation, when speaking about variables in a program, we think of that assignment as the current *state* of the program. In imperative programs, a stretch of code can of course change the values of variables by assignments, i.e., there will be state changes, so the assertion before a stretch of code will be different from the assertion afterwards. The assertion before is typically called *pre-condition* and the assertion afterwards the *post-condition* of that code piece (but of course the post-condition of a particular piece of code is the pre-condition for what follows, if anything).

Be it as it may, the discussion just explains that mappings of the form from equation (2.13) are also known as *state* in particular in the context of verification of imperative programs which deal with states and state changes. Logics like first-order logics in isolation are typically seen as *declarative*, not dealing with variable changes; thus the terminology talks about valuations rather than states (though it's the same thing). Later we will talk about modal and temporal logics, those are logics which (rather generally speaking) reason about changes, for instance changes over "time" during an execution of a system.

Now we have fixed the interpretation of function symbols and know how to assign values to variable. So everything is in place to give meaning to terms containing variables. Given an interpretation for, it's done by straightforwardly lifting σ to terms for which we write

$$[\![t]\!]^I_\sigma$$

Even if straighforward, for completeness sake, here is the definition by induction on the structure of terms:

Definition 2.4.4 (Interpretation of terms)**.** Given a signature Σ and a corresponding model $M = (A, I)$, the value of term t from $T_\Sigma(X)$, with variable assignment $\sigma : X \to A$ written $[\![t]\!]^I_\sigma$) is given inductively as follow:

$$\begin{array}{rcl} [\![x]\!]^I_\sigma & = & \sigma(x) \\ [\![f(t_1, \ldots, t_n)]\!]^I_\sigma & = & [\![f]\!]^I([\![t_1]\!]^I_\sigma, \ldots, [\![t_n]\!]^I_\sigma) \ . \end{array} \tag{2.14}$$

Variables are given their meaning by σ, function symbols are given their meaning by I (in the equation we write $[\![f]\!]^I$) and the rest is straightforward induction.

Before we tackle also the semantics of formulas, we have get one aspect concerning variables out of the way. It has to do with the quantifiers in the formulas. The quantifiers $\forall$ and $\exists$ range over variables and *bind* them. For instance, (free) occurrences of x in φ or *bound* by the quantification in $\forall x.\varphi$, so no longer free. The quantification introduces a *scope* for the bound variable, the variable can be seen as "local" to that scope.

Note that it's possible that a variable is at the same time free in some part of a formula and bound in another part, like variable x in $(\exists x.x = y + 15) \wedge y = 2 \times z$. Thus one speaks not just about free or bound variables, but more precisely of variable *occurrences*: x *occurs* bound in the left part of the conjunction and occurs free in the right-hand part. The other variables y and z occur free, only.

In the presence of variable binders, the notion of substitution needs some adaptation resp. some words of caution. Definition 2.3.3 defined substitution for terms over an algebraic signature. That implied there had been no quantifiers around, and everything was straightforward. Now, we want to make use of substitutions also for formulas. Substitutions are, as before, (see equation (2.9))... [careful with substitution, other binding, scoping mechanisms]

...

$$\varphi = \exists x.x + 1 \doteq y \qquad \theta = [x/y] \tag{2.15}$$

The satisfaction relation

Without further ado, we can now fix the meaning of first-order formulas. As mentioned in connection with propositional logics, one can do that in the form of a semantic function $[\![\_]\!]$ (like we did for terms as part, which are part of formulas)) or with a satisfaction relation $\models$ (see equation (2.16). One finds both notations, it's a matter of taste, and we will use both.

The semantics combines the interpretation of functional symbols, variables and relational symbol with that of the logical connectives, the latter defined analogously to what's been done for propositional case (except for the quantifiers, which are new, of course).

Given a signature Σ as fixed, one says that variable assignment σ makes a formula φ true in a model M, or φ *holds* in a model M and under an assignment σ, or M and σ *satisfy* φ, or similar. Notationally written as

$$M, \sigma \models \varphi \quad \text{or} \quad \sigma \models_M \varphi \ . \tag{2.16}$$

Alternatively, as said, on can base the definition on a functional formulation, for instance writing $[\![\varphi]\!]^I_\sigma$ for the truth value of φ in a given model M and under a variable assignment σ.

Definition 2.4.5 (Satisfaction relation)**.**

$$
\begin{array}{lll}
M,\sigma \models \top & & \\
M,\sigma \not\models \bot & & \\
M,\sigma \models P(t_1,\ldots,t_n) & \text{iff} & P^I(\llbracket t_1\rrbracket^I_\sigma,\ldots,\llbracket t_n\rrbracket^I_\sigma) \\
M,\sigma \models \neg\varphi & \textit{iff} & M,\sigma \not\models \varphi \\
M,\sigma \models \varphi_1 \land \varphi_2 & \textit{iff} & M,\sigma \models \varphi_1 \quad \text{and} \quad M,\sigma \models \varphi_2 \\
M,\sigma \models \varphi_1 \lor \varphi_2 & \textit{iff} & M,\sigma \models \varphi_1 \quad \text{or} \quad M,\sigma \models \varphi_2 \\
M,\sigma \models \varphi_1 \rightarrow \varphi_2 & \textit{iff} & M,\sigma \not\models \varphi_1 \quad \text{and} \quad M,\sigma \models \varphi_2 \\
M,\sigma \models \forall x.\varphi & \textit{iff} & M,\sigma' \models \varphi \quad \text{for all } x\text{-variants } \sigma' \text{ of } \sigma \\
M,\sigma \models \exists x.\varphi & \textit{iff} & M,\sigma' \models \varphi \quad \text{for some } x\text{-variants } \sigma' \text{ of } \sigma
\end{array}
\tag{2.17}
$$

Side remark 2.4.6 (Terminology)**.** In seldom cases, some books call the pair of (M,σ) a model (and the part M then called interpretation or something else). It is a terminology question (thus not so important), but it may lead to different views, for instance what "semantical implication" means. The standard default answer what that means is the following (also indepdendent form the logic). A formula φ_1 implies semantically a formula φ_2, if all models of φ_1 are also models of formula φ_2 (the satisfaction of φ_1 implies satisfaction of φ_2).

Now it depends in if one applies the word "model" to M or to the pair M,σ). That leads to different notions of semantical implications, at least if one had formulas with free variables. For closed formulas, it does not matter, so some books avoid those finer points but just defining semantical implication on closed formulas. □

Proof theory

We have fixed the semantics or model and interpretation of first-order logic formulas, we have defined validity and satisfiability. What is missing is how to *prove* that a formula or sentence is valid or satisfiable. Notions like truth and satifaction etc. are defined in reference to an external mathematical "reality", e.g., in reference to a Σ-structure. That's all very standard, though the sceptics may ask about what kind of reality are we talking about, like how real are, for instance, the natural numbers or other structures, are they somehow more fundamental than first-order or other logics that justifies that we use mathematical structures to give meaning to logical formulas? Shouldn't logics rather be the foundation of mathematics?

We don't loose sleep over such philosophical questions. But it's a concern, of course, having clarified validity and satisfaction etc., how to **establish it** as a fact or disprove it. So it's a question of mechanical procedure or algorithmic approach to **prove** or **disprove** valid formulas or derive logical consequences from a given set of sentences. That's what proof theory is concerned with.

Using the semantic definition of validity directly, one would have to check for *all* models that each of them make the formula true. There are infinitely many models and one cannot check them all. If one is after satifiabilty, not validity, one could try one model after the other to find one that satisfies the formula. That likewise would be hopelessly unpractical.

There are not only infinitely many models, the models themselves may be infinite, so even after having decided on a model, checking if a formula holds in the model or not may not be possible.

All that attempts would not qualify as proof theory anyway. We are interested how to establish the "truth-ness" of a formula or refute it. Model theory asks, is this formula "true" or valid, proof-theory asks is a formula provable. More generally, it's about how to devise proof systems for that task, and asking about limits of what can be proven in a given logic. Implementations of such calculi are called *theorem provers.* That's because a formula derivable or inferrable in a proof systems are also called a *theorem* of the calculus. There are also proof systems trying to establish if a formula is satisfiability, corresponding implementation are rather called satisfiability checkers or constraint solvers.

Above we mentioned limits of what can be proven. It's a well-known fact, that validity of first-order logic is *undecidable.* Which means there cannot be an algorithm or a proof system that can be used to *decide* that. For first-order logic, the best we can hope for is a system that is able to derive as theorems all *valid* formulas, and non-valid formulas cannot be derived as theorem. Isn't that deciding validity? No, of course not, because when a formula is not valid, non-derivability can mean that inference process does not terminate, so one never knows. *Soundness* and *completeness* is as good as it gets for first-order logics, but the logic is undecidable. Actually, it does not take much to obtain logics where also completeness is out of the window,[6] for instance second-order logics or doing first-order logics and fixing the model to be the natural numbers with the usual operations.

Since we are not too deeply interested in first-order logic in this lecture and also not so much in validity and theoremhood, but rather in model checking, we don't dig deeper here. We just say a few gegeral words about the shape of proof systems.

Proof systems There are *many* flavors of proof systems for first-order logic. We don't explore them or their differences. We just mention some aspects largely common to such systems, also for different logics.

Let's focus on validity, i.e., a system to derive only (and perhaps all) valid formulas. There are also refutation systems, basically focusing on the dual question of (non-)satisfiabilty. They are also of practical relevance, but as said, let's focus just one flavor, validity.

The purpose of such a proof system is to allow to derive all valid formulas. Those are infinitely many, so one cannot just list them all, of course. So, it's a (specification of) a way to derive or infer them. Often, the proof system is given in a non-deterministic way, leaving out which derivation steps are supposed to be explored first. Of course in an implementation, derivation strategies need to be fixed, and heuristics of how concretely make use of the proof system in terms of strategy or scheduling is very important. For the specification of a proof system, it is often simpler to leave out, at least to some extent, such questions that have to do with efficience, and focus of whether the system is sound and complete (if possible).

[6]Of course if one does not care about soundness, then completeness is obtainable, though that's a non-option typically.

The way that such inference systems are arranged, very generally, is that there is a pool of a priori given formulas together with a way to generate or derive new formulas from those and those already derivered previousy. The given formulas are often called **axioms** and the generation process is specified by derivation **rules**.

To use a standard notation, a rule looks as follows

$$\frac{\varphi_1 \quad \cdots \quad \varphi_n}{\psi} \tag{2.18}$$

Often rules are of more refined shape, involving side condictions, or more information than just formulas, but anyway.

The formulas $\varphi_1, \ldots, \varphi_n$ on top of the rule are called the **premises** and the one ψ below the **conclusion** of the rule. The rule is intended to express that if all of the premises have been established, either by virtue of being an axiom or having been derived earler in the process, the the conclusion ψ can be added as (another) theorem to the derived formulas.

If the axioms are semantically valid, and if the rules like the one shown allow only to generate valid formulas when using prior valid formulas in a derivation step, then obviously all derivable formulas are valid. In other words, one has a **sound** derivation system for the given logic. Typically, that's easy to achieve and easy to see. **Completeness** is quite a different story, that no valid formula is missed in the genration process.

The derivation of new formulas from old ones can be seen as a *sequence*, adding new formulas to older ones. Alternatively, in many presentations, it's rather a tree. Be that as it may, the derivation is also called a **proof** of the "last" formula at the end. Since the rule system does typically not specify which rules to apply when, proving a formula corresponds to **proof search**, and the choice of search strategy has a huge impact in the efficiency of a implementation of a proof system.

The way the story here is presented, deriving valid formulas is a process that generates valid formulas from axioms using rules. Also that is not the way that proving works. In practical verification tasks, one has most probably a property in mind one is instereted in to establish or check if it holds. In other words, simply generating valid formulas one after other so see if the one interesting one shows up in the process is not a useful strategy. So the derivation rules in a search are better seen *backwards*. A *conclusion* of the rule is the *goal* one tries to establish and the hypotheses of a rule are the *subgoals*.

Given a proof system one writes $\vdash \varphi$, of the formula φ can be *derived* in the system. One also says synonymously, φ is a *theorem* of the given derivation system. One writes $\nvdash \varphi$ if that's not the case. Those are proof theoretic statements about the formula and the proof system. There model theoretic counterparts are $\models \varphi$ and $\not\models \varphi$: the formula is *valid* resp. is not valid. A theory is a set of formulas, and one can also there distinguish between a proof theoretic theory: all the formulas such that $\vdash \varphi$ in a given derivation system. Or the semantic or model theoretic theory, all valid formulas $\models \varphi$.

Ideally, one has a sound and complete proof systems for the models one wants to cover. In this case, semantical concepts like validity or truth coincide with the proof theoretic ones like derivability or theoremhood. In notation

$$\models \varphi \quad \text{iff} \quad \vdash \varphi$$

Often, one is also interested not just in validity, but in "consequences". Like $\Gamma \models \varphi$ meaning that φ is a *semantical consequence* of the set of formulas Γ. It's a consequence in that every model of (all the formulas of) Γ is also a model of φ. Proof-theoretically one would write $\Gamma \vdash \varphi$. That's meant to represent: In the given proof system, φ can be derived using the given axiom plus the formulas from Γ (as so to say additional axioms). Again, in a favorable situation with soundness and completeness, one has $\Gamma \models \varphi$ iff $\Gamma \vdash \varphi$.

In some way, implication $\rightarrow$ represents a form of conseqence (as do inference rules): something follows from or is implied by something else. Thus $\Gamma \models \varphi$ may seem the same $\models \bigwedge \Gamma \rightarrow \varphi$: the conjunction of all formulas in Γ implies φ. For some logics, that's indeed the case, but one has to be careful, depending on the logic, and the exact interpretation of what $\models$ actually is supposed to mean. For instance, in first order logics when having formulas containting free variable. At any rate, that's fineprint for our interest and we don't dig deeper.

A proof system for propositional logic As said, we don't venture into introducing, comparing, and investigating different proof systems. But let's have at least one or two simple examples for concreteness sake. We don't show a proof system for first-order logic, to keep it really simple, we focus on propositional logic. A first-order logic system could contain the propositional rules as shown, but would need more to deal with quantification.

$$\frac{}{\varphi \rightarrow (\psi \rightarrow \varphi)}\ \mathrm{Ax}_1$$

$$\frac{}{(\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))}\ \mathrm{Ax}_2$$

$$\frac{}{((\varphi \rightarrow \bot) \rightarrow \bot) \rightarrow \varphi}\ \mathrm{DN}$$

$$\frac{\varphi \qquad \varphi \rightarrow \psi}{\psi}\ \mathrm{MP}$$

Table 2.2: Axioms and rules for a fragment of propositional logic

The system has three axioms and one inference rule. The name DN of the third axiom stands for *double negation*. The reason for that is, that negation can be defined or explained in terms of $\bot$ and implication by having $\neg\varphi$ defined as

$$\varphi \rightarrow \bot\ .$$

Actually, we were a bit sloppy when saying there are three axioms. We should more correctly say, that there are *3 axiom schemas*. What that means is that each axiom scheme represents infinitely many concrete axioms as *instantiation*. The “formulas” φ, ψ, and χ mentioned in the rule system are meta-variables representing formulas, but are not formulas themselves, strictly speaking.

The rule system from Table 2.2 is in a form called *Hilbert style*. Hilbert systems are an early and influential form of proof systems. It’s however mostly of historical interest, as there are better ways to automate reasoning. What exactly is a good design depends also on what one wants to achieve (and for what kind of domain), for instance whether it’s for *interactive* theorem proving or fully automated deduction. Hilbert-style systems are not good for much practical things. . .

To get at least a taste of the proof system (and a taste why it’s clumsy), let’s look at a very small example, deriving a rather trivial property.

Example 2.4.7. The following derivation establishes $\vdash \varphi \to \varphi$.

$$\begin{aligned}
&(\varphi \to ((\varphi \to \varphi) \to \varphi)) \to \\
&\quad ((\varphi \to (\varphi \to \varphi)) \to (\varphi \to \varphi)) && \text{Ax}_2 && (2.19)\\
&\varphi \to ((\varphi \to \varphi) \to \varphi) && \text{Ax}_1 && (2.20)\\
&(\varphi \to (\varphi \to \varphi)) \to (\varphi \to \varphi) && \text{MP on (2.19) and (2.20)} && (2.21)\\
&\varphi \to (\varphi \to \varphi) && \text{Ax}_1 && (2.22)\\
&\varphi \to \varphi && \text{MP on (2.21) and (2.22)} && (2.23)
\end{aligned}$$

□

The example also illustrates the concept of axiom schemas. The axioms from the derivation rules are not used “as-is” but they are instantiated. For instance, in the first two lines, it’s a rather complicated specific case of the second axiom. It’s problematic, since if we we proceed line by line, generating new theorems from previous until we get what we want, how are we supposed to come up with the specific instances of the axioms that ultimately lead to success? And that’s just one aspect why Hilbert systems are no practically useful representation. Another one is that it seems weird that in order to prove something pretty obvious like $\varphi \to \varphi$ one is forced to use some ridicously convoluted formula like $(\varphi \to ((\varphi \to \varphi) \to \varphi)) \to ((\varphi \to (\varphi \to \varphi)) \to (\varphi \to \varphi))$ as part of the argument. That’s not how it should be.

Natural deduction style proof systems Better alternatives are known as *sequent calculi* or systems of *natural deduction*. All those systems are about deriving valid formulas. There are also so-called *refutation systems*, on the other hand, which do something else, namely they try to *refute* a formula by checking if it’s negation is *satisfiable*, among them various resolution methods and tableaux method.

We don’t look here into those refutation alternatives, but we at least mention in which way sequent calculi and natural deduction systems differ from the Hilbert formulation. At at very high level, also sequent calculi and natural deduction systems are of the form described, there is a pool of axioms (resp. axioms schemas) and there are derivation rules.

The systems typically work with derivation *trees* as proofs, not with sequences of formulas, but that's more a presentational issue. One can easily use a tree like presentation for proofs in the Hilbert rules of Table 2.2: the axioms are at the leaves and the inner nodes are instances of the modus ponens rule MP. Maybe also for historical reasons, Hilbert-style proofs are mostly presented as sequences, not as trees, but that's not the real difference.

Hilbert-style formulations were "criticized" as unnatural, in that it was perceived that using those axioms and rules does not really reflect in a natural way, that logical arguments are done.

The above derivation in Example 2.4.7 should have given a feeling of that. The proof was pretty convoluted, in particular given the fact that this is really the most *trivial* valid thing that can be said about $\to$, and implication somehow lies the heart of logics. Logics is not just about describing things with formulas, it's about *drawing conclusions*, figuring out consequences from facts (as for instance in automated reasoning). Now that a formula φ is implied by itself seems *the single most obvious thing about implication* and it does not feel right that it requires so many steps to derive.

One could say, why not add that as axiom to the other ones if it's so central? One could of course do so. Of course then one has more axioms than one needs since, as the derivation shows, the "self-implication" formula can be derived.

Trying to get an axiomatization with $\varphi \to \varphi$ as additional axiom would also miss the point. The problem is not that it's hard to derive the particular case where φ is implied by φ as the most immediate consequence. One needs *generally a better way to draw conclusions.*

Gentzen in particular suggested that a more natural way to arrange logical arguments is reasoning from hypotheses. Like:

> **Assuming** φ as hypothesis, it follows that φ holds.

One could write $\varphi \rhd \varphi$ for that and use that as axiom. An as a rule one could use

$$\frac{\varphi_1 \rhd \varphi_2}{\rhd\ \varphi_1 \to \varphi_2} \to\text{I} \tag{2.24}$$

Also that expresses a simple fact about $\to$. If one assumes φ_1 as hypothesis and that allows to prove φ_2, as stipulated in the premise of the rule, then surely one can derive the implication $\varphi_1 \to \varphi_2$ in the conclusion, and that implication is derived *without* assuming φ_1 as hypothesis on the left of $\rhd$ (the hypothesis has been discharged in the derivation step).

Such kind of argumentation working with hypotheses is characteristic for natural deduction system and sequent calculi. Another general distinguishing feature of such systems, in comparison to Hilbert's formulation is that they have *more rules and less axioms.*

In the discussion here, we focus on one connective, namely $\to$. Of course, there may be more built into the logics (as opposed to be explained as macros of others), like $\wedge$, $\neg$ etc. All of them need to be covered by a proof system. Hilbert's standpoint would be: there is exactly *one derivation rule, namely modus ponens* and that's it. At least

$$\frac{\Gamma \rhd \varphi_1 \qquad \Gamma \rhd \varphi_2}{\Gamma \rhd \varphi_1 \wedge \varphi_2}\ \wedge\text{-I} \qquad \frac{\Gamma \rhd \varphi_1 \wedge \varphi_2}{\Gamma \rhd \varphi_1}\ \wedge\text{-E}_1 \qquad \frac{\Gamma \rhd \varphi_1 \wedge \varphi_2}{\Gamma \rhd \varphi_2}\ \wedge\text{-E}_2$$

Table 2.3: Introduction and elimination rules for conjunction

in propositional logic, in first-order logic one would add also some rule(s) dealing with quantification. Anyway, all the propositional connectives are covered by an approriate selection of axioms, and propositional deriviation relies solely on applying modus ponens as the one and only rule.

Natural deduction, in contrast, would cover each logical connective with a few rules, characteristic for the connective. One particular connective, say $\wedge$ is characterised by so-called *introduction* and *elimination* rules. The terminology is easily explained. An intruduction rule, for instance for $\wedge$ is a rule, where the premises don't mention $\wedge$, but the conclusion does, so applying the rule introduces that construct. An elimination rule does the converse. At least one premise mentions $\wedge$, but the conclusion does not.

Let's have a look then at the rules for conjunction for illustration. Before doing that, and looking back at equation (2.24) for implication, we realize the the format of the rules of the derivation system gets more complex compared to the ones we started out with in equation (2.18). The the premises and the conclusions don't just consist of formulas. In the example from equation (2.24), the only premise is of the form $\varphi_1 \rhd \varphi_2$. That's slightly simplified, insofar that left of the symbol $\rhd$, in general there is a *set* of formulas, not just one as in the discussion from above. So-called sequent calculi often also work with sets of formulas on the *right* of $\rhd$. Let's stick however, to a formulation with one formula on the right. Thus, the rules operate with pairs of the form $\Gamma \rhd \varphi$, with Γ a set of formulas, the hypotheses. Such a *judgement* or sequent is intended to capture that, assuming *all* formulas from Γ, φ holds.

Concerning concretely the rules for $\wedge$, see Table 2.3. For $\wedge$, there is one introduction rule and two elimnation rules. It's characteristic for so called natural deduction system, that each connective of the syntax is covered by appropriate introduction and elimination *rules*. The so-called sequent calculi work similar, however focusing on elimination rules. As mentioned, the pair $\Gamma \rhd \varphi$ is often called a sequent. Still, the rules from Table 2.3 are natural deduction rules, namely that of a natural deduction system in sequent formulation, as it's called.

A Hilbert style system would capture $\wedge$ not by rules, but by axioms like the ones from Table 2.3.

$$\varphi_1 \to \varphi_2 \to (\varphi_1 \wedge \varphi_2) \quad \wedge\text{-I} \qquad \varphi_1 \wedge \varphi_2 \to \varphi_1 \quad \wedge\text{-E}_1 \qquad \varphi_1 \wedge \varphi_2 \to \varphi_2 \quad \wedge\text{-E}_2$$

Table 2.4: Axioms for conjunction

We have said, natural deduction systems work with introduction and elimination rules. That also applies to implication $\rightarrow$. The rule modus ponens MP 2.2 is nothing else than the elimination rule for $\rightarrow$. For completeness sake, Table 2.5 shows the corresponding rules.

$$\frac{\Gamma, \varphi_1 \rhd \varphi_2}{\Gamma \rhd \varphi_1 \rightarrow \varphi_2} \rightarrow\text{-I} \qquad \frac{\Gamma \rhd \varphi_1 \rightarrow \varphi_2 \qquad \Gamma \rhd \varphi_1}{\Gamma \rhd \varphi_2} \rightarrow\text{-E}$$

Table 2.5: Introduction and elimination rules for implication

The introduction rule $\rightarrow$-I is the rule that shows how natural deduction systems (and likewise sequent calculi) work explicitly with a set of hypotheses and the set of hypotheses in the rule changes from Γ, φ in the premise to Γ in the conclusion, discharging φ. We had discussed that already in connection with the special situation for $\varphi \rightarrow \varphi$ in equation (2.24).

2.5 Modal logics

This section gives a taste of so-called modal logics. It covers some general aspects from the perspective of logics. In the verification part later, we will likewise deal with modal logics, but more from the perspective of how to do model checking and we cover a few specific modal and in particular temporal logics of interest in computer science and program verification. Still, some warm-up about modal logics in general can't hurt. After some general remarks in Section 2.5.1, we follow the same path as we did for propositional logics and first-order logics. We fix some syntax in Section 2.5.2, we address semantics, models, etc., and say a few words about proof systems in Sections 2.5.3 and 2.5.4.

Actually, there is no such thing as "the" modal logics. To some extent that is true also for propositional logics and first-order logics. Sure, one can select a few fundamental connectives or add more syntactic sugar to the syntax, there are classical versions or intuitionistic ones. And then there are many different ways how to design a proof system for the logics. But mostly, when saying "propositional logics", everyone means more or less the same thing, the rest is details. Similar for first-order logics.

Modal logics are differentiated also by the fact "modality" can mean quite different things (like time, belief, knowledge and other things). All that can be captured by modalities, leading to different modal logics.

At the face of it, logics covering beliefs, time, knowledge have not much in common, at least there seems no reason why one would expect so. It turns out, they have a lot in common, namely what conceptually constitutes a **model** for such logics. The central section for us is thus Section 2.5.3, which introduces the idea of such models. They will be called **Kripke models** and in computer science, they basically can be seen as **transition systems** (a form of graphs). The section about the syntax of modal logic will likewise be less interesting for us, since later we will work with *specific* syntax for specific logics, in particular temporal logics not with a generic syntax with modal operators. The section about proof system is less relevant, as we are concerned often with model checking, not deriving valid formulas. But the idea of Kripke structure will be important, as this is the notion of "model" when doing model(!) checking (for temporal logics).

2.5.1 Introduction

The roots of logics date back very long, and those of modal logic not less so. As far as his interests in logics was concerned, Aristotle not only wrote about syllogisms etc., like modus ponens, he also had his fingers in the origins of modal logic and discussed some kind of "paradoxes" in that context that gave food for thought for future generations of philosophers, puzzling about what to make out of such modalities.

Very generally, a logic of that kind is concerned not with *absolute* statements (which are true or else false) but *qualified* statements, i.e., statements that "depend on" something. An example for a modal statement would be "tomorrow it rains". It's difficult to say in which way that sentence is true or false, only time will tell... It's an example of a statement depending on "time", and *tomorrow* is an example of a *temporal* modality. But there are other modalities, as well (referring to *knowledge* or *belief* like "I know it rains", or "I believe it rains" or similar qualifications of absolute truth.

Statements like "tomorrow it rains" or others were long debated, often with philosphical and/or even religous connotions like: is the future deterministic (and pre-determined by God's providence), do persons have a free will, etc. Those questions will not enter the lecture. Nonetheless: determinism vs. non-determinism is meaningful distinction when dealing with program behavior, and we will also encounter temporal logics that view time as *linear* which kind of means, there is only *one* possible future, or *branching*, which means there are many. It's however, not meant as a fundamental statement about the "nature of nature", it's just a distinction of how we want to treat the system. If we want to check individual runs, which are sequences of actions, then we are committing ourselves to a *linear* picture (even when dealing with non-deterministic programs). But there are branching alternatives to that view as well, which lead to branching temporal logics.

Different flavors of modal logic lead to different axioms. Let's write $\Box$ for the basic modal operator (whatever its interpretation), and consider

$$\Box\varphi \to \Box\Box\varphi \ , \tag{2.25}$$

with φ being some ordinary statement (in propositional logic perhaps, or first-order logic). If we are dealing with a logic of belief, should the following ne a valid formula: If I believe that something is true, do I believe that I believe that the thing is true? What about "I believe that you believe that something is true"? Do I believe it myself? Not necessarily so.

As a side-remark: the latter can be seen as a formulation in *multi-modal* logic: it's not about one single modality (being believed), but "person p believes", i.e., there's one belief-modality per person; thus, it's a *multi-modal* logic. We start in the presentation with a "non-multi" modal logic, where there is only *one* basic modality (say $\Box$). Technically, the syntax may feature two modalities $\Box$ and $\Diamond$, but to be clear: that does *not* yet earn the logic the qualification of "multi": the $\Diamond$-modality is in all considered cases expressible by

$\Box$, and vice versa. It's analogous to the fact that (in most logics), $\forall$ and negation allows to express $\exists$ and vice versa.

Now, coming back to various interpretations of equation (2.25): if we take a "temporal" standpoint and interpret $\Box$ as *"tomorrow"*, then certainly the implication should not be valid. If we interpret $\Box$ differently, but still temporally, as *"in the future"* then again the interpretation seems valid.

If we take an "epistemologic" interpretation of $\Box$ as "knowledge", the left-hand of the implication would express (if we take a multi-modal view): "I know that you know that φ". Now we may ponder whether that means that then I *also* know that φ? A question like that may lead to philosophical reflections about what it means to "know" (maybe in contrast with "believe" or "believe to know", etc.).

The lecture will not dwell much on philospophical questions. The question whether equation (2.25) will be treated as *mathematical question*, more precisely a question of the assumed underlying *models* or *semantics*.

It's historically perhaps interesting: modal logic has attracted long interest, but the question of "what's the mathematics of those kind of logics" was long unclear. Long in the sense that classical logics, in the first half of the 20th century had already been superthoroughly investigated and formalized from all angles with elaborate results concerning *model theory* and proposed as "foundations" for math etc. But no one had yet come up with a convincing, universally accepted answer for the question: "what the heck is a model for modal logics?". Until a teenager and undergrad came along and provided the now accepted answer, his name is *Saul Kripke*. And models of modal logics are now called *Kripke-structures* (it's basically transition systems).

2.5.2 Syntax

Modal logics comes equipped with connectives to express the modalities the logics is interested in. One typicallly finds two modal operators, called **"necessity"** and **"possibility"**. Formulas $\Box\varphi$ and $\Diamond\varphi$ are read as "necessarily φ" and "possibibly φ" or simply "box φ" or "diamond φ".

Different flavors of modal logics interpret the modal operators differently. In a temporal setting $\Box\varphi$ can be interpreted as "always φ. And there are many other modal logics.

logics	$\Box\varphi$
temporal	always φ
doxastic	I believe φ.
epistemic	I know φ.
intuitionistic	φ is provable.
deontic	It ought to be the case that φ.

One finds logics with more operators, but those two are the classical ones. One can also work with combinations, like trying to capture the temporal evolution of knowlege, which would be a temporal-epistemic logic. We will restrict here the modal operators to $\Box$ and $\Diamond$, and in the later part of the lecture mostly work with a temporal mind-set.

Definition 2.5.1 shows a syntax for some vanilla modal logics. It is formulated as extension of propositional logics, so it's propositional modal logics. One can also use first-order logic as underlying logic, that does not change the modal part of the story.

Definition 2.5.1 (Formulas of modal logic)**.** The formulas of (propositional) modal logic are given by the following grammar.

$$\begin{array}{rcll} \varphi & ::= & \top \mid \bot & \text{atomic propositional formulas} \\ & \mid & \varphi \wedge \varphi \mid \neg\varphi \mid \varphi \rightarrow \varphi \mid \ldots & \text{propositional formulas} \\ & \mid & \Box\varphi \mid \Diamond\varphi & \end{array} \tag{2.26}$$

2.5.3 Semantics

Now to the core of the modal logic section, clarifying the semantics and the notion of models for such logics.

Kripke structures The definition below makes use of the "classical" terminology for modal logics. It's actually quite simple, based on a relation, here written R, on some set. The relation is also called *accessibility relation* and the "points" connected by that relation are called *worlds*. So: a modal model is thus just a relation, or we also could call it a *graph*, but traditionally it's called a *frame* (a Kripke frame). That kind of semantics is also called *possible world semantics* (but not graph semantics or transition system semantics, even if that would be an ok name as well).

The Kripke frame in itself is not a model yet, in that it does not contain information to determine if a modal formula holds or not. The general picture is as follows: the elements of the underlying set W are called "worlds": on some world some formulas holds, and in a different world, different ones. "Embedded" in the modal logic is an "underlying"logic. As said we mostly assume *propositional* modal logic, but one might as well consider an extension of first-logic with modal operators. For instance, in connection with *runtime verification* will make use of a first-order variant of LTL, called QTL, quantified temporal logic, but that will be later. In the propositional setting, what then is needed for giving meaning to model formulas is an interpretation of the propositional variables, and that has to be done **per world**.

In the section of propopositional logic, we introduced "propositional variable assignments" $\sigma : P \rightarrow \mathbb{B}$, giving a boolean value to each propositional variable from σ. What we now call a *valuation* does the same *for each world* which we can model as a function of type

$$W \rightarrow P \rightarrow \mathbb{B} \; .$$

Alternatively one often finds also the "representation" to have valuations of type $W \to 2^P$: for each world, the valuation gives the set of atomic propositions which "hold" in that word. Both views are, of course equivalent in being *isomorphic*.

Side remark 2.5.2 (Labelled "graphs")**.** The valuation function V associates a propositional value to each propositional value in each world. As mentioned, a Kripke frame may also be called a graph or also transition system. In the latter case, the worlds may be called less pompously just *states* and the accessibility relation is called transition relation. The individual edges in the graph are seen as *transitions* from one state to another. That terminolgy is perhaps more familiar in computer science. The valuation function can also be seen to **label** the states with propositional information. A transition system with attached information is also called *labelled transition system*.

But one has to be careful a bit with terminology, resp. aware that there are different ways of "labelling" and different names for it.

When it comes to *labelled* transition systems, additional information can be attached to transitions or states (or both). Often labelled transition systems, especially for some areas of model checking and modelling, are silently understood as *transition-labelled*. For such models, an edge between two states does not just express that one can go from one state to the other. It states that one can go from one state to the other *by doing such-and-such* as expressed by the label of the transition. In an abstract setting, the transitions may be labelled just with letters from some alphabet.

As we will see later, going from a transition system with unlabelled *transitions* to one with transition labels correspond to a generalization from "simple" modal logic to multi-modal logic. But independent on whether one considers transitions as labelled or not, there is a "state-labelling" at least, namely the valuation that is needed to interpret the propsitions per world or state.

As a side remark: classical automata can be seen as labelled transition systems, as well, with the transitions being labelled. There are also variations of such automata which deal with input *and* output (thereby called I/O automata). Two classical versions of that idea are used in describing hardware (which is a form of propositional logic as well…), both label the transitions for the input. However, one version labels the states with the output (Moore-machines) whereas another one labels the transitions with the output (Mealy-machines), i.e., in the latter representation, transitions contain input as well as output information. Both correspond to different kinds of hardware circuitry (Moore roughly correspond to synchronous hardware, and Mealy to asynchronous one).

We will encounter automata later, as well, but in a form that fits to our particuar modal resp. temporal logic needs. In particular, we will look at Büchi-automata, which are like standard finite-state automata except that they can deal with infinite words (and not just finite ones). Those automata have connections, for instance, with LTL, a central temporal logic which we will cover. □

Definition 2.5.3 (Kripke frame and Kripke model)**.**

- A *Kripke frame* is a structure (W, R) where
  - W is a non-empty set of *worlds*, and
  - $R \subseteq W \times W$ is called the *accessibility relation* between worlds.
- A *Kripke model* M is a structure (W, R, V) where
  - (W, R) is a frame, and
  - V a function of type $V : W \to (P \to \mathbb{B})$, called *valuation*.

The valuation function is isomorphic to a function $V : W \to 2^P$; we will make use of both representations.

Kripke models are also called *Kripke structures*. The standard textbook about model checking Baier and Katoen [2] does not even mention the word “Kripke structure” or “Kripke model”, it basically uses *transition systems* instead of Kripke models with worlds called states (and the concept of Kripke frame is called *state graph* there). I say, it's “basically” the same insofar that there, they (sometimes) also care to consider labelled transitions, and furthermore, their transition systems are equipped with a set of *initial states*. Whether one has initial states as part of the graph does not make much of a difference.

Also the terminology concerning the set P varies a bit (we mentioned it also in the context of propositional logics). What we call here propositional variables, is also known as propositional constants, propositional atoms, symbols, *atomic propositions*, whatever.

Example 2.5.4 (Kripke model). Assume $P = \{p, q\}$ as propositional variables. Figure 2.1 shows (the graphical representation of) a simple Kripke model. Formulaically $M = (W, R, V)$ is given by $W = \{w_1, w_2, w_3, w_4, w_5\}$, $R = \{(w_1, w_5), (w_1, w_4), (w_4, w_1), \ldots\}$, and $V = [w_1 \mapsto \emptyset, w_2 \mapsto \{p\}, w_3 \mapsto \{q\}, \ldots]$.

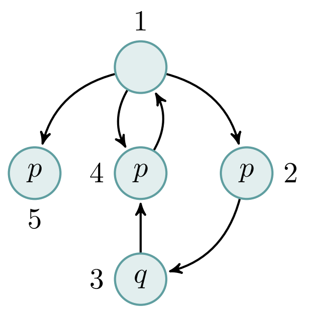


Figure 2.1: Example of a Kripke model

□

The example is slightly informal (and also later we allow ourselves these kind of “informalities”, appealing to the intuitive understanding of the reader). There are five worlds, numbered for identification. In the Kripke model, they are referred to via $w_1, w_2, \ldots$ (not as $1, 2, \ldots$ as in the figure). Later, we often call corresponding entities states, not worlds, and then we tend to use $s_1, s_2, \ldots$ for typical states. For the valuation, we use a notation of the form $[... \mapsto ...]$ to denote a *finite* mapping.

In particular, we are dealing with finite mappings of type $W \to 2^P$, i.e., to subsets of the list of atomic propositions $P = \{p, q\}$. The sets are not explicitly noted in the graphical illustration, i.e., the set-braces $\{\ldots\}$ are omitted. For instance, in world w_1, no propositional letter is mentioned, i.e., the valuation maps w_1 to the empty set $\emptyset$.

An isomporphic (i.e., equivalent) view on the valuation is, that it is a function of type $W \to (P \to \mathbb{B})$ which perhaps captures the intended interpretation better. Each propositional letter mentioned in a world or state is intended to evaluate to "true" in that world or state. Propositional letter not mentioned are intented to be evaluated to "false" in that world.

As a side remark: we used finite mappings inf the example and illustration, and in and many applications. The definition of Kripke structure, however, does *not* require that there is only a finite set of worlds, W in general is a *set*, finite or not.

Satisfaction relation Now we come to the *semantics* of modal logic, i.e., how to interpret formulas of (propositional) modal formulas. That is done by defining the corresponding *satisfaction* relation, written as $\models$, as before. After the introduction and discussion of Kripke models or transition systems, the satisfaction relation should be fairly obvious to some extent, especially the part of the *underlying logic* (here propositional logic): the valuation V is made exactly so that it covers the base cases of atomic propositions, namely give meaning to the elements of P depending on the current world of the Kripke frame. The treatment of the propositional connectives $\wedge$, $\neg$, $\ldots$ is identical to their treatment before. Remains the treatment of the real innovation of the logic, the modal operators $\Box$ and $\Diamond$.

Definition 2.5.5 (Satisfaction)**.** A modal formula φ is *true* (or it *holds*) in the world w of a model V, written $V, w \models \varphi$, if:

$$\begin{array}{lll}
V, w \models p & \text{iff} & V(w)(p) = \top \\
 & & \\
V, w \models \neg\varphi & \text{iff} & V, w \not\models \varphi \\
V, w \models \varphi_1 \vee \varphi_2 & \text{iff} & V, w \models \varphi_1 \text{ or } V, w \models \varphi_2 \\
 & & \\
V, w \models \Box\varphi & \text{iff} & V, w' \models \varphi, \text{ for all } w' \text{ such that } w\ R\ w' \\
V, w \models \Diamond\varphi & \text{iff} & V, w' \models \varphi, \text{ for some } w' \text{ such that } w\ R\ w'
\end{array}$$

As mentioned, we consider V to be of type $W \to (P \to \mathbb{B})$. If we equivalently assumed a type $W \to 2^P$, the base case of the definition would read $p \in V(w)$, instead.

For now, we prefer the former presentation for 2 reasons (but actually, it does not matter of course). One is, it seems to fit better with the presentation of propositional logic, generalizing directly the concept a boolean valuation. Secondly, the picture of "assigning boolean values to variables" fit better with seeing Kriple models more like transition systems, for instance capturing the behavior of computer programs. There, we are not so philosphically interested in speaking of "worlds" that are "accessible" via some accessibility relation R, it's more like states in a progam, and doing some action or step does a transition to another state, potentially changing the memory, i.e., the content of variable, which in the easiest case may be boolean variables. So the picture that one has a control-flow graph of a program and a couple of variables (propositional or Boolean variables here) whose values change while the control moves inside the graph seems rather straightforward and natural.

Sometimes, other notations or terminology is used, for instance $w \models_M \varphi$. Sometimes, the model M is fixed (for the time being), indicated by the words like. "Let in the following M be defined as … ", in which case one finds also just $w \models \varphi$ standing for "state w satisfies φ", or "φ holds in state w" etc. but of course the interpretation of a modal formula requires that there is alway a transition system relative to which it is interpreted.

Often one finds also notations using the "semantic brackets" $[\![\_]\!]$. Here, the meaning (i.e., truth-ness of false-ness of a formula, depends on the Kripke model as well as the state, which means one could define a notation like $[\![\varphi]\!]^M_w$ as $\top$ or $\bot$ depending on wether $M, w \models \varphi$ or not. Remember that we had similar notation in first-order logic $[\![\varphi]\!]^I_\sigma$

Side remark 2.5.6 ("Forcing")**.** We discussed (perhaps uneccessarily so) two isomorphic view of the valuation function V. Even if not relevant for the lecture, it could be noted that a third "viewpoint" and terminology exists in the literature in that context. Instead of associating with each world or state the set of propositions intended to hold in that state, one can define a model also "the other way around": then one associates with each propositional variable the set of states in which the proposition is suppoed to hold, one would have a "valuation"

$$\tilde{V} : P \to 2^W \ .$$

That's of course also an equivalent and legitimate way of proceeding. It seems that this representation is not "popular" when doing Kripke models for the purpose of capturing systems and their transitions (as for model checking in the sense of our lecture), but for Kripke models of intuionistic logics. Kripke also propose "Kripke-models" for that kind of logics (for clarity, I am talking about intuitionistic propositional logics or intuitionistic first-order logice et.c, not (necessarily) intuitionistic *modal* logics). In that kind of setting, the accessibility relation has also special properties (being a partial order), and there are other side conditions to be aware of. As for terminology, in that context, one sometimes does not speak of "w satisfies φ (in a model)", for which we write "$w \models p$", but says "world w *forces* a formula φ", for which sometimes the notation $w \Vdash \varphi$ is favored. But those are mainly different traditions for the same thing. □

For us, we sometimes use notations like $[\![\varphi]\!]^M$ to represent the set of all states in M that satisfy φ, i.e.,

$$[\![\varphi]\!]^M = \{w \mid M, s \models \varphi\} \ .$$

In general (and also other logics), $\models$- and $[\![\_]\!]$-style notations are interchangable and interdefinable. And we will freely make use of both

How to interpret "box" and "diamond" The pronounciation of $\Box\varphi$ as "necessarily φ" and $\Diamond\varphi$ as "possibly φ" are *generic*, and their meaning is given by Definition 2.5.5.

When dealing with specific modal logics, they still are mathematically given by the general definition, but how to think about that gets more specific and they will be called correspondingly more specifically, for instance: "in all futures φ" or "I know that φ" etc.

Related to the intended mindset, one imposes different restrictions on the accessibility relation R. In a temporal setting, if we interpret $\Box\varphi$ as "tomorrow φ", then it is clear that $\Box\Box\varphi$ ("φ holds in the day after tomorrow") is not equivalent to $\Box\varphi$. If, in a different temporal mind-set, we intend to mean $\Box\varphi$ to represent "now and in the future φ", then $\Box\Box\varphi$ and $\Box\varphi$ are equivalent. That reflects common sense and reflects what one might think about the concept of "time" and "days" and "future". Technically, and more precisely, it's a property of the assumed class of frames (i.e., of the relation R). If we assume that all models are built upon frames where R is *transitive*, then $\Box\varphi \to \Box\Box\varphi$ is generally true.

Validity and frame validity We should be more explicit about what it means that a formula is "generally true". We have encountered the general terminology of a formula being "true" vs. being "valid" already. In the context of modal logic, the truth-ness requires a model and a state to judge the truth-ness: $M, w \models \varphi$. A model M is of the form (W, R, V), it's a frame (= "graph") together with a valuation V. A *propositional* formula is *valid* if it's true for all boolean valuations (and the notion coincided with being a propositional tautology).

Now the situation get's more finegrained (as was the case in first-order logics). A modal formula is *valid* if $M, w \models \varphi$ for all M and all w. For that one can write

$$\models \varphi \tag{2.27}$$

So far so good. But then there is also a middle-ground, where one fixes the frame (or a class of frames), but the formula must be true *for all valuations* and all states. For that we can write

$$(W, R) \models \varphi \tag{2.28}$$

Let's abbreviate with F a frame, i.e., a tuple (W, R). We could call that notion *frame validity* and say for $F \models \varphi$ that "φ is valid in frame F". So, in other words, a formula is valid in a frame F if it holds in all models with F as underlying frame and for all states of the frame.

One uses that definition not just for a single frame; often the notion of frame-validity is applied to *sets* of frames, in that one says $F \models \varphi$ for all frames F such that ...". For instance, all frames where the relatiton R is *transitive* or *reflexive* or whatever. Those restrictions of the allowed class of frames reflect then the intentions of the modal logic (temporal, epistemic . . .), and one could speak of a formula to be "transitivity-valid" for instance, i.e., for all frames with a transitive accessibility relation.

The latter would be an ok terminology, but it's not standard. There are (for historic reasons) more esoteric names for some standard classes, for instance, a formula could be S4-valid. That refers to one particular restriction on R which corresponds to a particular set of axioms traditionally known as S4. See below for some examples.

Side remark 2.5.7 (Temporal logics and preview to LTL)**.** Coming back to the informal "temporal" interpretation of $\Box\varphi$ as either "tomorrow φ" vs. "now and in the future φ", where the accessibility relation refers to "tomorrow" or to "the future time, from now on". In the latter case, the accessibility relation would be reflexive and transitive. When thinking about such a temporal interpretation, there may also be another assumption on the frame, depending on how one sees the nature of time and future.

One way would be to see the time as *linear*. Points in time form a line, fittingly called a *timeline*, connecting the past and the future, with "now" in the middle, perhaps measured in years or seconds, or steps in an run of a program etc. With such a linear picture in mind, it's also clear that there is no difference between the modal operators $\Box$ and $\Diamond$.[7] In the informal interpretation of $\Box$ as "tomorrow", one should have been more explicit that "tomorrow" was meant "for all possible tomorrows" to distinguish it from $\Diamond$ that represent "there exist a possible tomorrow". In the linear timeline picture, there is only one tomorrow, we conventionally say "the next day" not "for all possible next days" or some such complications. Consequently, if one has such a linear picture in mind (resp. works only with such linear frames), one does not actually *need* two modal operators $\Box$ and $\Diamond$, one can collapse them into one. Conventionally, for that collapsed one, one writes $\bigcirc$. A formula $\bigcirc\varphi$ is interpreted as "in *the* next state or world, φ holds" and pronouced "next φ" for short. The $\bigcirc$ operator will be part of LTL (linear-time temporal logic), which is an important logic used for model checking and which will be covered later. When we (later) deal with LTL, the operator $\bigcirc$ corresponds to the modal operators $\Diamond$ and $\Box$ collapsed into one, as explained. Besides that, LTL will have *additional* operators written (perhaps confusingly) $\Box$ and $\Diamond$, with a *different interpretation* capturing "always" and "eventually" Those are also temporal modalities, but their interpretation in LTL is different from the ones that we haved fixed for now, when discussing modal logics in general. $\Box$

Restrictions on the frames, resp. the accessibility relation As mentioned, different classes of model logics arise by imposing restrictions on which frames one considers. Restrictions on the binary relation $R \subseteq W \times W$ include that it is reflexive, transitive, (right) Euclidian, total, that it's an order relation, and more. It's not the goal of the lecture to study those and compare different logics (and many variations have been studied indeed). Still, we at least mention some common restrictions.

[7] Characterize as an exercise what *exactly* (not just roughly) the condition the accessibility relation must have to make $\Box$ and $\Diamond$ identical.

Definition 2.5.8. A binary relation $R \subseteq W \times W$ is

- *reflexive* if every element in W is R-related to itself.

$$\forall a.\ a\, R\, a$$

- *transitive* if

$$\forall a\ b\ c. \quad a\, R\, b \wedge b\, R\, c \to a\, R\, c$$

- (right) *Euclidean* if

$$\forall a\ b\ c. \quad a\, R\, b \wedge a\, R\, c \to b\, R\, c$$

- *total* if

$$\forall a. \exists b. \quad a\, R\, b$$

The following remark may be obvious, but anyway: The quantifiers like $\forall$ and the other operators $\wedge$ and $\vee$ are not meant here to be vocabulary of some (first-order) logic, they are meant more as mathematical statements, which, when formulated in for instance English, would use sentences containing words like "for all" and "and" and "implies". One could see if one can formalize or characterize the defined concepts making use formally of a first-order logic, but that's not what is meant here. We use the logical connectives just as convenient shorthand for English words.

Example 2.5.9 shows some how some accessibiity relations are captured by different formulas, in that they are valid under the corresponding restriction.

Example 2.5.9.

- $(W, R) \models \Box\varphi \to \varphi$ iff R is reflexive.
- $(W, R) \models \Box\varphi \to \Diamond\varphi$ iff R is total.
- $(W, R) \models \Box\varphi \to \Box\Box\varphi$ iff R is transitive.
- $(W, R) \models \neg\Box\varphi \to \Box\neg\Box\varphi$ iff R is Euclidean.

2.5.4 Proof theory and axiomatic systems

We only sketch proof theory of modal logic, as we are more interested in model checking as opposed to verify that a formula is valid. There are connections between these two questions, though. As explained earlier, proof theory is about formalizing the notion of proof. That's done by defining a formal system, a proof system, that allows to *derive* or *infer* formulas from others. Formulas a priori given are also called *axioms*, and rules allow new formulas to be derived from previously derived ones (or from axiom). One may also see axioms as special form of rule, namely one without *premises*.

The style of presenting the proof system here is the plain old Hilbert-style presentation. As mentioned, there are other styles of presentations, some better suited for interactive, manual proofs and some for automatic reasoning, and in general more elegant anyway. As also mentioned, one difference between Hilbert-style and the natural deduction style presentations is that Hilbert's presentation put's more weight on the axioms, whereas the alternative downplay the role of axioms and have more deduction rules (generally speaking). That suits us fine: As discussed, different classes of frames (transitive, reflexive . . .) correspond to axioms or selection of axioms, and we have seen examples for that in Example 2.5.9.

Since we intend (classical propositional) modal logics to encompass classical propositional logic not just syntactically but also conceptually/semantically, we have all propositional tautologies as derivable. Furthermore, we have the standard rule of derivation, already present in the propositional setting, namely *modus ponens*.

That so far took care of the propositional aspects (but note that MP can be applied to all formulas, of course, not just propositional ones). But we have not really taken care of the modal operators $\Box$ and $\Diamond$. Now, having lot of operators is nice for the user, but puts more burden when formulating a proof system (or implementing one) as we have to cover more case. So, we treat $\Diamond$ as *syntactic sugar*, as it can be expressed by $\Box$ and $\neg$. Note: "syntactic sugar" is a well-established technical term for such situations, mostly used in the context of programming languages and compilers. Anyway, we now need to cover only one modal operator, and conventionally, it's $\Box$, necessitation. The corresponding rule consequently is often called the rule of (modal) *necessitation*. The rule is below called NEC, sometimes also just N or also called G (maybe historically so).

Is that all? Remember that we touched upon the issue that one can consider special classes of frames, for instance those with *transitive* relation R or other special cases, that lead them to special *axioms* being added to the derivation system. Currently, we do *not* impose such restrictions, we want **general frame validity**. So does that mean, we are done? At the current state of discussion, we have the propositional part covered including the possibility do to propositional-style inference (with modus ponens), we have the plausible rule of necessitation, introducing the $\Box$-modality. Apart from that, the two aspects of the logic (the propositional part and modal part seem conceptually *separated*. Note: a formula $\Box p \to \Box p$ "counts" as (an instance of a) propositional tautology, even if $\Box$ is mentioned. A question therefore is: are the two parts of the logic somehow further connected, even if we don't assume anything about the set of underlying frames?

The answer is, **yes**, and that connection is captured by the *axiom* stating that $\Box$ *distributes* over $\to$. The axiom is known as distribution axiom or traditionally also as axiom K. In a way, the given rules are the standard *base line* for all classical modal logics. Modal logics with the propositional part covered plus necessitation and axiom K are also called *normal* modal logics.

Side remark 2.5.10 ("Un-normal" modal logics)**.** As a side remark: there are also certain modal logics where K is dropped or replaced, which consequently are *no longer* normal logics. Note that it means they no longer have a Kripke-model interpretation either. Since our interest in Kripke-models is that we use transition systems as representing steps of programs, Kripke-style thinking is natural in the context of our course. Non-normal logics are more esoteric and "unconventional" and we don't go there. $\Box$

The distribution axiom K is written as "rule" without premises. The system focuses on the "new" aspects, i.e., the modal part. It's not explicit about how the rules look like that allow to *derive* propositional tautologies (which would be easy enough to do, and includes MP anyway). We have seen those earlier anyway.

$$\frac{\varphi \text{ is a propositional tautology}}{\varphi}\ \text{PL}$$

$$\frac{}{\Box(\varphi_1 \to \varphi_2) \to (\Box\varphi_1 \to \Box\varphi_2)}\ \text{K}$$

$$\frac{\varphi \to \psi \qquad \varphi}{\psi}\ \text{MP}$$

$$\frac{\varphi}{\Box\varphi}\ \textsc{Nec}$$

Table 2.6: Axioms for the base line propositional modal logics (“K”)

The sketched logic is is also known under the name **K** itself, so K is not just the name of the axiom. The presentation here is Hilbert-style, but could also present the same logics differently.

We show in Table 2.8 a few more axioms (with their traditional names, some of which are just numbers, like “axiom 4” or “axiom 5”). In the literature, then one considers and studies “combinations” of those axioms (like K + 5), and they are traditionally also known under special, not very transparent names like “S4” or “S5”. See Table 2.8 for some better known ones.

$$\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi) \tag{K}$$
$$\Box\varphi \to \Diamond\varphi \tag{D}$$
$$\Box\varphi \to \varphi \tag{T}$$
$$\Box\varphi \to \Box\Box\varphi \tag{4}$$
$$\neg\Box\varphi \to \Box\neg\Box\varphi \tag{5}$$
$$\Box(\Box\varphi \to \psi) \to \Box(\Box\psi \to \varphi) \tag{3}$$
$$\Box(\Box(\varphi \to \Box\varphi) \to \varphi) \to (\Diamond\Box\varphi \to \varphi)) \tag{Dum}$$

Table 2.7: Various properties of R and their axioms

The first ones are pretty common and are connected to more or less straightforward frame conditions (except K which is, as said, generally the case for a frame-based Kripke-style interpretation). Observe that T implies D.

The are many more different axiom studied in the literature, how they are related and what not. The axiom called Dum is more esoteric (the paper [17] calls it “[among the] most bizzare formulae that occur in the literature”) and actually, there are even different versions of that (Dum$_1$, Dum$_2$ . . .).

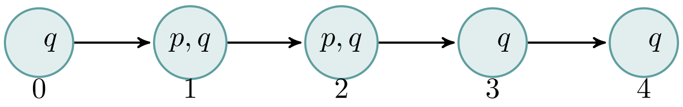
Logic	Axioms	Interpretation	Properties of R
D	K D	deontic	total
T	K T		reflexive
K45	K 4 5	doxastic	transitive/euclidean
S4	K T 4		reflexive/transitive
S5	K T 5	epistemic	reflexive/euclidean
			reflexive/symmetric/transitive
			(i.e. equivalence relation)

Table 2.8: Different flavors of modal logic

Concerning the terminology *doxastic* logic is about beliefs, *deontic* logic tries to capture obligations and similar concepts. Epistemic logic is about knowledge.

2.5.5 Exercises

Exercise 2.5.11 (Formulas holding in worlds of a model)**.** Consider the frame (W, R) with $W = \{0, 1, 2, 3, 4\}$ and $(i, i + 1) \in R$.



Let the "valuation" $\tilde{V}(p) = \{1, 2\}$ and $\tilde{V}(q) = \{0, 1, 2, 3, 4\}$ and let the model M be $M = (W, R, V)$. Which of the following statements are correct in M and why?

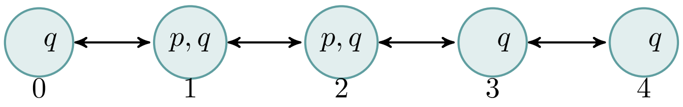
$$\begin{array}{rcl} M, 0 & \models & \Diamond\Box p \\ M, 0 & \models & \Diamond\Box p \to p \\ M, 2 & \models & \Diamond(q \land \neg p) \land \Box(q \land \neg p) \\ M, 0 & \models & q \land \Diamond(q \land \Diamond(q \land \Diamond(q \land \Diamond q))) \\ M & \models & \Box q \end{array}$$

$\square$

Solution 2.5.12 (of Exercise 2.5.11)**.** *The answers to the above questions are: yes, no, yes, yes, yes.*

Perhaps a remark concerning the status $\Diamond\Box p$ in the first situation. The frame is $\square$

Exercise 2.5.13 (Bidirectional frame)**.** A frame (W, R) is *bidirectional* iff $R = R_F + R_P$ s.t. $\forall w, w'.\ w\ R_F\ w' \leftrightarrow w'\ R_P\ w$.



Consider $M = (W, R, V)$ from before. Which of the following statements are correct in M and why?

$$\begin{array}{rcl}
M, 0 & \models & \Diamond\Box p \\
M, 0 & \models & \Diamond\Box p \rightarrow p \\
M, 2 & \models & \Diamond(q \wedge \neg p) \wedge \Box(q \wedge \neg p) \\
M, 0 & \models & q \wedge \Diamond(q \wedge \Diamond(q \wedge \Diamond(q \wedge \Diamond q))) \\
M & \models & \Box q \\
M & \models & \Box q \rightarrow \Diamond\Diamond p
\end{array}$$

Solution 2.5.14. *The solutions are: no, yes, no, yes, yes, no.*

The frame used in Exercise 2.5.13 is bi-directional. Effectively, the relation R in the example is *symmetric*, so actually the example would not need to mention the concept of being bi-directional. Later, when we present *temporal logics*, there are variants that can speak not just about the future, but also about the past. In such a temporal setting, one could interpret the formulas on bi-directional frames, the future operators being based on R_F and the operators taking about the past on R_P.

Exercise 2.5.15 (Validities)**.** Which of the following formulas are *valid* in modal logic. For those that are not, argue why and find a class of frames on which they become valid.

$$\begin{array}{l}
\Box\bot \\
\Diamond p \rightarrow \Box p \\
p \rightarrow \Box\Diamond p \\
\Diamond\Box p \rightarrow \Box\Diamond p
\end{array}$$

□

Solution 2.5.16. *1. $\Box\bot$: Valid on frames where $R = \emptyset$.*
2. $\Diamond p \rightarrow \Box p$: Valid on frames where R is a partial function.
3. $p \rightarrow \Box\Diamond p$: Valid on bidirectional frames.
4. $\Diamond\Box p \rightarrow \Box\Diamond p$: Valid on Euclidian frames.

□

2.6 Dynamic logics

2.6.1 Introduction

Dynamic logics is a so-called program logics and used in connection with a pogramming language. I.e., the formalism contains *two levels* of syntax, one the programming notation, and then the logics, both interwoven. A specified program consists of the program code plus logical annotations, sprinkled throughout the code. Hoare-logic is a typical example for that, where the code is annotated with assertions (in first-order logic or similar). Indeed, dynamic logics can be seen as a generalization of Hoare logics. Such a specification style, formulas being written as part of the code syntax, is a bit different from the way, for instance, LTL will be used. There, the logic is used *externally*, specifying the behavior of a program or process *from the outside*. It's a bit like the difference between a black box and white box view in testing.

We want to talk about programs, states of a program, and in particular *change* of the state of the program when executing programming instructions. The (perational semantics of a program is typically some form of transition system (or Kripke structure), and we know already, that logics that talk about transition systems are *modal logics*. We said, dynamc logic can also be seen as a generalization of Hoare-logics. Our exposition does not present it under that perspective by starting with Hoare-logics, we start by explainig multi-modal logic.

Connections between modal logics and FOL There are various connections between FOL and modal logics, for instance, modal logic may be seen as FOL with one free variable, but seeing it like that we loose the "beauty" of modal logics (being tailor-made as logic speaking about transition systems). Such connections between other logics and modal logics are not important for the course.

2.6.2 Multi-modal logic

It's a pretty straightforward generalization of modal logics. Instead of one relation in the transition system, we handle many, instead of having box and diamond as modal operators over one relation, we have modal operators for each of them. That's basically it.

We start by illustrating it with a Kripke frame with 2 relations R_a and R_b, both subsets of $W \times W$. An alternative and equivalent way of seeing it that one deals with a "labelled transition relation", i.e. one $R \subseteq W \times A \times W$, where A is the set of labels, here $A = \{a, b\}$.

Definition 2.6.1 (Formulas of multi-modal logic)). The formulas of a multi-modal logic are given by the following grammar.

$$\varphi \quad ::= \quad p \mid \bot \mid \varphi \rightarrow \varphi \mid \Diamond_0 \varphi \mid \Diamond_1 \varphi \mid \ldots \tag{2.29}$$

where p is from a set of propositional atoms.

Compard to the "mono"-modal syntax from Definition 2.5.1, we left out some operators. But those can easily defined by combinations of the others in the standard way. In particular multi-modal logical also has a duals to the diamind-operators, with $\Box_i$ and $\Diamond_i$ being duals of each other, of course.

$$\varphi ::= \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg \varphi \mid \Box_0 \varphi \mid \Box_1 \varphi \mid \ldots \tag{2.30}$$

Not only is the syntax extended, also the concept of model needs the corresponding obvious generalzation. So a *Kripke frame* then is of the form

$$(W, \{R_0, R_1, \ldots\}) \tag{2.31}$$

with each relation R_i a relation over W, and multi-modal Kripke-models adds a valuation V on top of that, as before. Also the semantics, i.e., satisfaction relation is effectively unchanged:

$$M, w \models \Diamond_i \varphi \quad \text{iff} \quad \exists w'.\ w\ R_i\ w' \text{ and } M, w' \models \varphi \tag{2.32}$$

Cf. also the original versions of Kripke frames and models from Definition 2.5.3 and of $\models$ from Definition 2.5.5.

The relations R_i may may overlap; i.e., their intersection need not be empty. Also infinitely many modalities are possible, so not just $\{R_a, R_b, R_c\}$ for a finite set of labels or $\{R_0, R_1, \ldots, R_n\}$. As a further remark: when discussing (a variant of) PDL and maybe TLA (temporal logic of actions), the different relations R_a arise from "programs" or "actions". So, if one sees the a as taken from a set A, then the set is not taken as unstructured and containing uninterpreted letters from an alphabet or numbers. Instead, the actions may have a syntax in itself, representing programs or single steps in a program. Since there are many *actions* or *programs*, which lead to infinitely many corresponding "modalities".

2.6.3 Dynamic logics & PDL

There are different variants of the concept of dynamic logics. As mentioned at the beginning, as a program logics, it contains logical notation and notation for logical specification. Not only can one base the idea on different underlying logics (like propositional vs. first-order logics), but also are the programming languages and notations that one can equip with a specification and verification formalism. For instance, the KeY verifier tool[8] is a theorem prover for Java programs, based on dynamic logics. Thus, the underlying program "notation" is Java.

For the presentation here, we take a more abstract view, i.e., we present dynamic logics *not* on top of a concrete programming language like Java. Capturing such a language by logic proof rules to enable verification of programs in that language is complex. Not so much because the dynamic logic as such is complex, but because programming languages themselves are complex: many special cases have to be covered, languages offer many abstractions and different features that can interfere with each other. It's not impossible, as said, KeY does just that (and implements corresponding derivation rules and other assisting techologies). But when discussing dynamic logic for a real programming language, the language details would overwhelm the logical aspects and it would become more a quite detailed study of synatcial and semantical aspects of a particular language, like for instance Java. Detailed, as it would descibe minutely the behavior of the language by logical formulas.

What we therefore do instead is using a *propositional* version of dynamic logics, PDL, on so-called **regular** programs.

[8]See `https://www.key-project.org`.

Regular programs Dynamic logics speaks about "programs", where a program is written in some "notation". in other words, programs written in some form of programming language. Here, (propositional) dynamic logic talks about **regular programs**.

Compared to programs written in real programming languages, regular programs are written in a rather more restrictive or abstract program notation, namely *regular programs*.

As a multi-modal logic, dynamic logics is interpreted on *labelled transition systems*. Before, when talking about multi-modal logics, we presented the labels as *unstructured*, just some elements from some label set A. Here, we add structure on A, basically *regular expressions*, which are seen as the underlying programming language notation. Of course, that's a very abstract way of representing programs, basically abstracting away from concrete data, like values of variables etc. That fits well with the decision to work with propositional logic as underlying formalism, as opposed to first-order logics.

Regular languages are a well-known formalism, used for various purposes. For instance they play a role for covering so-called lexical aspects of a programming language in the lexer phase of a compiler. The describe finite-state automata, they are used for (regular) pattern matching, etc.

We use regular language syntax, resp. an extension of the commonly used syntax to denote programs. In that aspects, it corresponds to the well-know view that regular expressions is a declarative notation for finite-state machines (or transition systems), which can be understood as some machine-model that can be executed, taking transition after transition.

Let's assume a set Π_0 of *atomic* programs a, b, ..., also seen as atomic *actions* or atomic labels. Besides that and s in regular expressions, there are constucts for sequences, non-deterministic chhoices ("or"), and iteration, the Kleene start. Those can be seen as abstraction to programming constructs like sequential compositions, conditionals, and loops. Of course in a programming language, conditionals are *not* non-deterministic. The non-determinism in regular programs in th presentd form is a result of *abstraction*. Since the language has abstract away from data (like the logic part based on propositional logic as core), conditionals, lacking the data to make a case-distinction, are correspondingly abstracted to non-deterministic choices. It's a level of abstraction that correponds to control-flow graphs (with the data-part abstracted away).

The discussion so far focused on constructs that are supported also by regular expressions, atoms a, choice $+$, sequencing, written $\cdot$, and the Kleene star.[9]

But regular programs contain further constructs. There are two constants, called skip and fail and written $\mathbf{1}$ and $\mathbf{0}$. The first one represents the program that does not do anything, and is guaranteed to terminate. The second one is sure to not terminate i.e., diverge or at least not terminate properly, but not doing anything else. These programs are polar opposites to each others and can be considered duals to each other (but we have introduced the concept of duality only for logics, not for program notations).

Then there is one last construct, programs of the form $\varphi?$. Such a program is kind of like a mixture of the skip program $\mathbf{1}$ and the fail program $\mathbf{0}$. Why the latter to terminate resp. fail to terminate unconditionally, a test $\varphi?$ either succeeds, i.e., behaves like $\mathbf{1}$, or

[9]In regular expressions one often uses $|$ instead of $+$, but it's the same thing.

immediately fails, i.e., behaves like **0**, *depending on* whether the formula φ evaluates at the given situation to true or false. Let's first summarize the syntax of regular programs in Definition 2.6.2. Note that the syntax given in the definition is not complete since the logical formulas φ of the tests are not yet nailed down. That will be done shortly afterward.

Definition 2.6.2 (Regular programs)**.** The syntax of *regular programs* $\alpha, \beta \in \Pi$ is given according to the grammar:

$$\alpha ::= a, \ldots \in \Pi_0 \mid \mathbf{1} \mid \mathbf{0} \mid \alpha \cdot \alpha \mid \alpha + \alpha \mid \alpha^* \mid \varphi? \,. \tag{2.33}$$

Now to the formulas φ. It's formulas from *propositional dymamic logics*, a modal logic, and besides the usual propositional connectives, the interesting one is the *modality*, of course. Let's base our syntax on the box- or necessity-modality; the diamond counterpart is definable as dual. The modality is written $[\alpha]$. So the multiplicity of the modalites comes from that they are "indexed" by (regular) programs. Note that the syntax of the programs refers to dynamic logica formulas via the tests, and the level of formulas refer back to programs via the modality operator(s): the two levels of programs and formulas are mutually recursive. So the syntax for regular programs from equation (2.33) is part of the grammar for in equation (2.34) below.

Definition 2.6.3 (PDL syntax)**.** Given a set Π_0 of *atoms* (or atomic regular programs or atomic actions), with typical elements a, b, ... The formulas φ of *propositional dynamic logic* (PDL) over *regular programs* α are given as follows.

$$\begin{array}{lcl} \alpha & ::= & a, \ldots \in \Pi_0 \mid \mathbf{1} \mid \mathbf{0} \mid \alpha \cdot \alpha \mid \alpha + \alpha \mid \alpha^* \mid \varphi? \\ \varphi & ::= & p, q, \ldots \in P \mid \top \mid \bot \mid \varphi \rightarrow \varphi \mid [\alpha]\varphi \end{array} \tag{2.34}$$

where P is a set of atomic propositions.

Some more remarks on tests Tests can be seen as special atomic programs which may have *logical* structure, but their execution properly **terminates** in the same state iff the test succeeds (is true), otherwise **fails** if the test is deemed false in the current state.

As for termination: test have two outcomes, a positive and a negative. If the test "succeeds", the test properly terminates. The qualification "properly" for this form of termination is a hint that there are also other forms of termination: When the test fails, it "terminates" in that it fails. That is sometimes called *improper termination.* In programming languages, a cause of improper termination can be raising an (uncaught) *exception.* One may interpret a failing test as divergence, i.e., definitely not terminating.

Actually, the behavior of tests is connected to *assertions* as known from many high-level programming language. For example, in Java, one can use `assert (b)` as construct, where `b` is a Boolean expression. Basic programming hygiene mandates, that `b` has no side effects, so it's a side-effect free boolean expression over the variables used in the programs (like making sure that `x >= y` or similar). Indeed, assert-statements of that form corresponds to a *simple* form of tests as supported in DL. It's a simple form, in that only propositional formulas (= boolean expressions) over program variables are allowed, whereas in dynamic logic, more complex formulas are allowed. And the formulas of dynamic logic can contain program code as part of their modalities.

Note: it's about proper or improper termination of the *test*, not the program. In general a test is used in a program, for instance a test followed by the rest of the program. In regular programs that is written as $\varphi ? {\cdot} \alpha$ (in other contexts and real programming languages, often semicolon `;` is used for sequential composition instead).

The difference between proper termination and failure is: in the first case, the test terminates without consequences (especially no side effects) and the program behaves as α. On the other hand, when the test fails, the whole program fails, i.e., terminates improperly as well. That corresponds to the situation, that an assertion `assert(b)` raises an exception which is uncaught and causes the whole program to terminate in a non-proper way.

So, *simple* Boolean tests are of the form $\varphi ?$ where φ is a propositional formula:

$$\varphi ::= \top \mid \bot \mid \varphi \to \varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \ldots .$$

and more generally, for complex tests, φ is an arbitrary formula in *dynamic logic*.

Some remarks about PDL als multi-modal logics As multi-modal logic, formulas are interpreted on a Kripke-model with multiple relations: each α is represented by a relation R_α. The relations for the basic program $a, b, \ldots \Pi_0$ are assumed given. They represent the the semantics of the atomic statements in the (abstraction of the) concrete program one models or more generally of the programming language. Operations on/composition of programs are interpreted as operations on relations.

When considering the language as such (and not just one concrete program), there are infinitely many box modalities $[\alpha]$ (same for diamond $\langle\alpha\rangle$ of course), and there are infinitely many relations R_α.

The interpretation of the two modalities we know already from the modal logic part. The $[\alpha]$ corresponds to a *universal quantification* of all α-successors, the dual $\langle\alpha\rangle$ stipulates the *existance* of an α-successor.

If we spell it out for the box-modality, $[\alpha]\varphi$ means

> If program α is started in the current state, then, *if* it terminates, then in its final state, φ holds.

Analoguously (or rather dually), $\langle\alpha\rangle\varphi$ means

> If program α is started in the current state, then, there there is a successor, i.e., there is the possibility that the program terminates, and when doing so and stepping to that possible successor, φ holds.

There is, however, a subtlety now which was not much visible in the modal logical setting (including the multi-modal one), where the actions were uninterpreted and unstructured symbols from an alphabet. Now, they *are* interpreted and structured, namely interpreted as regular programs, and that includes the possibility of non-termination. So having a pair s_1 and s_2 in a relation R_α, i.e., $s_1 R_\alpha s_2$ (for which we also write

$$s_1 \xrightarrow{\alpha} s_2$$

it's interpreted as that a program starts in state s_1 and when executing α, it reaches s_2 when α properly terminates. So, a transition $s_1 \xrightarrow{\alpha} s_2$ implies that α terminates properly. The *absence* of such a transition implies that α is not done at that point, of course, but also α is attempted but does not properly terminate. In our regular languages, that would be caused by a failed test. In real programming languages, they may be other reason for not terminating properly.

For the modalities, $[\alpha]\varphi$, being interpreted as universal quantification over all α-successors stipulates for a state s_1, that, **if** the execution of α terminates starting from s_2, then the progam will be in a state, say s_2, for which φ holds. If the program, starting at state s_1 does not terminate, then $[\alpha]$ *vacuously* holds in s_1. Programs may be non-deterministic (including regular programs). So it may be the case, that some executions of α terminate and others don't. The formula $[\alpha]\varphi$ imposes a restriction (namely φ) on the post-states for all terminating executions of α. But does not require *anything* for the others. This is connected with the notion of *partial* correctness (as opposed to total correctness).

In contrast, the dual operator $\langle\alpha\rangle\varphi$ insists on that there *exists* an execution, where α terminates after which φ holds.

The terminology if partial and total correctness may become more plausible if one would consider the restricted setting where programs are *deterministic*. Regular programs can be non-deterministic, but dynamic logic, as stated, makes sense also with other program notation, for instance, talking about sequential and deterministic programs. For example, the KeY-tool focuses on *sequential* Java programs (though also adaptations to multi-threaded Java exist).

Anyway, if the programming language happens to be deterministic, then for each s_1, there exists *at most one* s_2 such that $s_1 \xrightarrow{\alpha} s_2$: if α terminates, s_2 is *the* sucessor state, if not, there is no successor state. In that setting as before, $[\alpha]\varphi$ specifies that, if the program terminates, the successor state must satisfy φ, but if it does not terminate, it's fine too (= **partial correctness**). The dual operator $\langle\alpha\rangle\varphi$, on the other hand, *insists* that α terminates plus that φ holds afterwards (= **total correctness**).

The notions of partial and total correctness are often used in connection with Hoare logic. Indeed, Hoare logic can be seen as a special, restricted case of dynamic logic. There, one operates syntactically not with "modal operators", but one write specifications as so-called *triples*. Triples can be interpreted in a total or in a partial correctness way. Sometimes, one distiguished that notationally by writing $[\varphi_1]\ \alpha\ [\varphi_2]$ and $\{\ \varphi_1\ \}\ \alpha\ \{\ \varphi_2\ \}$, correspondingly. Note that, unlike here, the logical formulas φ_1 and φ_2 do not contain "program syntax": in dynamic logic, the two layers (programs and logic) are more intimately inverwoven (via the modalities and the tests), in Hoare-logics, the layers are more clearly separated (programs sprinkled with formulas in between). In that sense, Hoare-logic is a restricted form of dynamic logic.

Exercise 2.6.4 (Define conventional programming constructs)**.** The syntax of regular programs cannot really be called a programming notation. Now the exercise is to define conventional programming constructs with the help of the introduced vrsion of dynamic logics in combination with the constructs of regular programs. The constructs to encode are listed in equation (2.35).

$$\begin{array}{ll} \textbf{skip} & \textbf{fail} \\ \textbf{if } \varphi \textbf{ then } \alpha \textbf{ else } \beta & \textbf{if } \varphi \textbf{ then } \alpha \\ \textbf{case } \varphi_1 \textbf{ then } \alpha_1;\ \ldots\ \textbf{case } \varphi_n \textbf{ then } \alpha_n & \\ \textbf{while } \varphi \textbf{ do } \alpha & \textbf{repeat } \alpha \textbf{ until } \varphi \\ \textbf{while } \varphi_1 \textbf{ then } \alpha_1 \mid \cdots \mid \varphi_n \textbf{ then } \alpha_n \textbf{ od} & \end{array} \tag{2.35}$$

A few words to the constructs. The syntax from Definition 2.6.3 already contains **1** and **0**, which we said represent skip and fail and alongside we also explained what they supposed to mean. In the exercise, one is requested to give an "alternative" definition, using tests. Why did we then introduc **1** and **0** then in the first place? Well, the syntax of regular programs from Definition 2.6.3 makes sense *independent* from fixing the logic part. In our presentation, doing dynamic logic, the logic part can refer back to the program language level via tests. If one restrict to a simpler setting, leaving out th modalities and test, regular programs still make sense, only the logic is no longer strong enough to *encode* skip and fail (and the other constructs of this exercise).

2.6.4 Semantics of PDL

Let's nail down the semantics of PDL more precisely. The core we know already: dynamic logic is a multi-modal logic, so we need transition systems with multiple relations. We can call them also *labelled* transition systems of labelled Kripke structures. We are dealing in particular with dynamic logics for *regular* programs. In that particular setting, one can characterize the transition systems or Kripke frames further. The shape of the possible transition systems is not arbitrary. Since they result form describing the transitions of regular programs, further restrictions apply. That will lead to the definition of *regular* Kripke structures, and PDL will be interpreted over that. It's similar to the earlier discussions about modal logics, where we discussed restricted classes of frames, like with a transitive or reflexive accessibility or transition relation. Here the restriction is: the frames, resp. labelled transition systems are *regular*.

Let's start with adding "labels" to Kripke structures resp. frames.

Definition 2.6.5 (Labelled Kripke structures)**.** Assume a set of labels Σ. A *labelled Kripke structure* is a tuple (W, R, Σ) where

$$R = \bigcup_{l \in \Sigma} R_l \tag{2.36}$$

is the union of the relations indexed by the labels of Σ.

The set Σ is also called the *alphabet*. Alternatively we can define R as a ternary (= three-argument) relation:

$$R \subseteq W \times \Sigma \times W\ . \tag{2.37}$$

For labels in general, we may use $l, l_1 \dots$ but also $a, b, \dots$ or others. Besides that, is nicer to use $\xrightarrow{a}$ as notation instead of R_a, and write more suggestively

$$w_1 \xrightarrow{a} w_2 \quad \text{or} \quad s_1 \xrightarrow{a} s_2$$

instead of $(w_1, w_2) \in R_a$ resp. $(s_1, s_2) \in R_a$ (or instead of $(w_1, a, w_2) \in R$ resp. $(s_1, a, s_2) \in R$ if we define R as in equation (2.37), which is of course equivalent).

Regular Kripke structures Now, as promised, we look at the consequences of working with regular programs on the shape of the labelled transition systems. The labels are thought of as representing regular programs and thus having structure, there the different relations R_a are related, reflecting the intended semantics of regular programs. The interpretation of interpretation of certain programs or labels fixed. For instance $\mathbf{0}$ represents the failing program and $\alpha_1 \cdot \alpha_2$ is fixed to represent sequential composition. Thus relations like $\mathbf{0}$, $R_{\alpha_1 \cdot \alpha_2}, \dots$ must obey side-conditions reflecting the intended meaning of those constructs. The interpretation of the "atoms" of regular programs a is assumed given, so also the relations R_a are assumed given, and we then have to fix the interpretation semantics of the remaining constructs of regular programs on top of that. That leads to the definition of regular Kripke structures.

Definition 2.6.6 (Regular Kripke structures (no tests)). A *regular Kripke structure* is a Kripke structure labelled as follows. For all atomic programs $a \in \Pi_0$, choose some relation R_a. For the remaining syntactic constructs (except tests), the corresponding relations are defined inductively as follows.

$$\begin{array}{lcl}
R_{\mathbf{1}} & = & Id \\
R_{\mathbf{0}} & = & \emptyset \\
R_{\alpha_1 \cdot \alpha_2} & = & R_{\alpha_1} \circ R_{\alpha_2} \\
R_{\alpha_1 + \alpha_2} & = & R_{\alpha_1} \cup R_{\alpha_2} \\
R_{\alpha^*} & = & \bigcup_{n \geq 0} R_\alpha^n
\end{array}$$

In the definition, Id represents the identity relation, $\circ$ relational composition, and R^n and the n-fold composition of R.

Actually, there is one piece missing, namelely how to represent $R_{\varphi?}$. PDL is conceptually more complex than plain multi-modal logics in that there is a mutual dependence of the programs and the formulas. Formulas contain programs via the modalities and programs contain formulas via the tests. Thus, there is also a mutual dependence on the semantics of formulas, i.e., the definition of the satisfaction relation $\models$, and the conditions on the transition relation of regular Kripke structure, on which the formulas are interpreted. So in principle, $\models$ and regular Kripke-structures would have to be defined "at the same time" (Definitions 2.6.6 above and 2.6.7 below). After defining regular Kripke structures omitting the tests in Definition 2.6.6, we continue here by definining $\models$ first, and only afterwards we add the missing piece to regular Kripke structures.

Interpreting PDL formulas We will next define the satisfaction relation $\models$ in the usual manner by induction on the structure of the formulas.

Definition 2.6.7 (Satisfaction relation)**.** A PDL formula φ is *true* in the world w of a regular Kripke model $M = (W, \rightarrow, V)$, written $M, w \models \varphi$, if:

$$\begin{array}{lll}
M, w \models p & \text{iff} & p \in V(w) \text{ for all propositional atoms} \\
M, w \not\models \bot & & \\
M, w \models \top & & \\
M, w \models \varphi_1 \rightarrow \varphi_2 & \text{iff} & \text{whenever } M, w \models \varphi_1 \text{ then also } M, w \models \varphi_2 \\
M, w \models [\alpha]\varphi & \text{iff} & M, w' \models \varphi \text{ for all } w' \text{ such that } w \xrightarrow{\alpha} w' \\
M, w \models \langle\alpha\rangle\varphi & \text{iff} & M, w' \models \varphi \text{ for some } w' \text{ such that } w \xrightarrow{\alpha} w'
\end{array} \tag{2.38}$$

The semantics should contain no surprises: it's the standard Kripke interpretation in a multi-modal setting. One ingredient, as said, is missing, though, that's how to the semantics for *tests* (coming next).

To give the exact interpretation of test programs $\varphi?$ as relations, we need to know how to interpret formulas φ.

- *omitted* so far: what relationship corresponds to

$$\varphi?$$

- remember the intuitive meaning (semantics) of tests

Test programs Tests interpreted as subsets of the identity relation.

$$R_{\varphi?} = \{(w, w) \mid w \models \varphi\} \subseteq Id \tag{2.39}$$

Some special cases:

- $R_{\top?} = Id$
- $R_{\bot?} = \emptyset$
- $R_{(\varphi_1 \wedge \varphi_2)?} = \{(w, w) \mid w \models \varphi_1 \text{ and } w \models \varphi_2\}$
- $[\alpha]\varphi$ is like looking into the *future* of the program and then deciding on the action to take...

The slides show some special cases of the definition from equation (2.39). One may also compare that with the "exercises" slide from earlier. As discussed earlier, the intuition of "executing" a test in a particular state is that it either succeeds (it terminates properly, by doing nothing) or it "fails" which blocks the program from continuing. In the informal discussion back then concerning proper and non-proper termination, we drew a parallel to raising exceptions in programming languages. That parallel is adequate in particular for deterministic programs; for non-determistic ones it only goes so far. So, in connection with the non-deterministic choice operator, a few more words may be in order. See below, after finishing the discussion of equation (2.39).

Testing $\top$ always, i.e., in each state succeeds. That means that $R_{\top?}$ is the identity relation itself. It can also be interpreted that executing $\top?$ corresponds to executing `skip` (later, in a different context like LTL and programs there, "do-nothing" steps are also called *stuttering*...). Testing for $\bot$ "fail" in each state when executed, which means $R_{\bot}?$ is the empty relation. The corresponding program, the dual to `skip`, is also called `fail`. The standard Boolean operators are interpreted as usual. For instance the $\wedge$ as logical conjuction (or as *intersection* of R_{φ_1} and R_{φ_2}, which is the same).

Finally, as the most complex case, the interpretation for $[\alpha]\varphi?$: what kind of relation does that corresponds to, resp. what does it mean to execute such a test? Well, we know what $[\alpha]\varphi$ means. It specifies the "post-condition" for all executions of α. If that evaluates to true, the test succeeds and the program can proceeds, alternatively, it fails. So, this allows to "look into the future" of an execution, for instance, if one would write $[\alpha]\varphi? \cdot \alpha$ as regular program. So, the rest of the program is only executed, if it assured that all possible outcomes of α satisfy φ. Note that α in turn may contain further tests. Of course, one can speak about "other" programs than the rest $[\beta]\varphi? \cdot \alpha$ where $\beta \neq \alpha$, but still the rest α is only executed if the execution of β would satify the given property φ. Referring to future outcomes of programs, be it α or β is a very powerful mechanism.

Earlier, we compared tests to assertions like `assert(b)` in standard programming languages. Those assertions are very much weaker, in particular one cannot use modal operators to speak about complex properties that may involve specifyng the future of program and make the execution of the program depend on that. The propositional assertions are more intended for being checked at run-time, without involving complex properties referring to future continuations (or other complications).

Non-deterministic choice, tests, and exceptions As mentioned, the parallel between `fail`, i.e., $\bot?$ and exceptions works convincingly for *deterministic* programs, for non-determistic ones less so. That can be illustrated, of course, when combining tests with non-deterministic choices. As in regular expressions, the non-deterministic choice operator may be read as "or" (and written $+$ for regular programs, for regular expression $|$ is more common). So, $\alpha_1 + \alpha_2$ may be read as "do α_1 or α_2". All fine and good, but let's combine that with tests, and consider

$$\varphi? \cdot \alpha_1 + \neg\varphi? \cdot \alpha_2 \tag{2.40}$$

For the discussion here, the form of the assertion φ does not matter, is it may be a simple predicate (like $x = 0$ resp $x \neq 0$ for the negated case). The semantics starts, make a choice between the left-hand or the right-hand side. The left-hand side will be execute if the test $\varphi?$ succeeds. Alternatively choose the right hand side, which "then" will be executed if $\neg\varphi?$ succeeds, otherwise not.

What is slightly dubious seem the explanation

> "first execute the choice and then check the test; if successful, continue, if non-successful, don't do anything".

It's in particular dubious, if we intuitively think that a failed test amounts to raising an exception. The program from equation (2.40) is interpreted as "if-then-else": depending on which of the two tests evaluates to true, either the left-hand side or the right-hand side of the conditional is executed.

But if we think that *first* a choice is made, and *afterwards*, the corresponding test is executed to see if one can continue, that is a *different* way of thinking about the program. In particular, when the test is interpreted as potential exception, that's an unplausible illustration. In a conventional program involving choices, the semantics makes a choice, and if that leads to an exception, then that's what happens. Here, the failing alternative is ignored: having a transition corresponding to a non-propertly terminating program is interpreted the same way as not having a transition at all (remember $R_{\perp ?} = \emptyset$). It's unplausible to say, a program that throws an exception (or will through one in all possible futures is the same as having no program at all).

Here, the interpretation is, that a choice is made selecting among those alternatives that terminate properly, where branches that lead to failures (= non-proper termination) *are ignored* as if there were not even there. This form of choosing among "successful" alternatives while ignoring the ones that fail is called *angelic* choice and the form of non-determinism *angelic nondetermism.*[10] It miracously picks an alternative that will turn out "positively" in that it terminates properly, if such an alternative exists. Note that the choice makes not just a decision "right now" as in equation (2.40), which is used for illustration. Also in a situtation $\alpha_1 + \alpha_2$, where the two branches are not immediately "guarded" by a test, which can be used to make a decision now, the choice pics one of the "successful", i.e., properly terminating outcomes of α_1 or α_2. To have such an angelic interpretation of non-determinism is not part of a standard behavior of a programming language (neither is the dual, choising the worst outcome, which is known as *demonic* choice). These notions appear in connection with how to *interpret* the occurence of non-determistic choices in a program when it comes to verify properties about it. And, as it is, regular programs have an *angelic* intepretation of the choice operator.

Axiomatic System of PDL

Further reading On dynamic logic, a book nicely written, with examples and easy presentation: David Harel, Dexter Kozen, and Jerzy Tiuryn: [19]. Chap. 3 for beginners, a general introduction to logic concepts. This lecture is based on Chap. 5 (which has some connections with Chap. 4 and is strongly based on mathematical notions which can be reviewed in Chap. 1)

Exercises

The exercises have been placed on a separate sheet.

[10]We will encounter the concept of angelic and daemonic non-determinism again.

Exercises: Play with binary relations

- Composition of relations distributes over union of relations.
  $$R \circ (\bigcup_i Q_i) = \bigcup_i (R \circ Q_i) \qquad (\bigcup_i Q_i) \circ R = \bigcup_i (Q_i \circ R)$$
- $R^* \triangleq I \cup R \cup R \circ R \cup \ldots \cup R^n \cup \ldots \triangleq \bigcup_{n \geq 0} R^n$

Show the following:

1. $R^n \circ R^m = R^{n+m}$ for $n, m \geq 0$
2. $R \circ R^* = R^* \circ R$
3. $R \circ (Q \circ R)^* = (R \circ Q)^* \circ R$
4. $(R \cup Q)^* = (R^* \circ Q)^* \circ Q^*$
5. $R^* = I \cup R \circ R^*$

Exercises: Play with programs in DL

- In DL we say that two programs α and β are equivalent iff they represent the same binary relation $R_\alpha = R = R_\beta$.

Show:

1. Two programs α and β are equivalent iff for some arbitrary propositional constant p the formula $\langle \alpha \rangle p \leftrightarrow \langle \beta \rangle p$.
2. The two programs below are equivalent:

```
while $\varphi_1$ do                    if $\varphi_1$ then
    $\alpha$;                               $\alpha$;
    while $\varphi_2$ do $\beta$             while $\varphi_1 \vee \varphi_2$ do
                                            if $\varphi_2$ then $\beta$ else $\alpha$
```

   Hint: encode them in PDL and use (1) or work only with relations

Exercises: Play with programs in DL Use a semantic argument to show that the following formula is valid:

$$p \wedge [a^*]((p \to [a]\neg p) \wedge (\neg p \to [a]p)) \leftrightarrow [(a \cdot a)^*]p \wedge [a \cdot (a \cdot a)^*]\neg p$$

What does the formula say (considering a as some atomic programming instruction)?

Chapter 3 LTL model checking

What is it about?

Learning Targets of this Chapter

The chapter covers LTL and how to do model checking for that logic, using Büchi-automata.

Contents

3.1 Introduction

In this chapter, we leave behind a bit the classical "logical" treatment of logics like asking for validity etc., i.e., asking $\models \varphi$, but proceed to the question of *model checking*, i.e., when does a concrete model satisfies a formula $M \models \varphi$ (more precisely, when does a state s in a model satisfies a formula, written $M, s \models \varphi$). We do that for a specific modal logic, indeed, a specific temporal logic. It's one of the most prominent ones and the first one that was taken up seriously in computer science (as opposed to studied in logics, mathematics or philosophy). We will also cover a central way of doing model checking of such temporal logics, namely *automata-based* model checking.

3.1.1 Temporal logic?

Temporal logic is a modal logic of "time". There is not just one temporal logic, there are in fact many. Time in that context is mostly not understood as the real-world time, like time of the day in hours and seconds, though there are temporal logics to deal with that.

Time is understood more abstractly, capturing more *changes* in a system or situation but abstracting away from when exactly that happens. Even if one abstracts away from that, there aare different ways of modelling time. One inportant distinction is that of **linear** vs. **branching** time.

The linear picture is probably more how one informally thinks about time. Each day is followed by the next one, time flows from the past to the future, that's the linear picture behind a *time line*. Systems, in particular concurrent systems, are *non-deterministic*: running it multiple times will result in different outcomes. That can come fron internal reasons, like different scheduling decisions or other sources of internal non-determinism, But it can also be cause by interacting with the outside world. That's typical for reactive or interactive systems and different interactions leads to different reactions as well (external non-determinism). Whatever reasons for the non-determinism, if one repeats running a system, the run or execution, seen as a linear sequence of steps, will be different. And typically, there are very many different runs.

But still, that's a *linear* picture. The behavior of a system is characterised by a set of runs or executions (typically a huge number, maybe infinite). *Branching* time takes a more complex view. In that view, the non-determinism of the system is not just captured by the fact that there are many different linear runs, but by modelling that a each point in time (or a least some intermediate points in time) the system can behave non-deterministically in that it could continue in different ways: the continuating branches. Repeating that pattern leads to a tree-like model, which underlies branching time. We see both models and logics for both models.

Besides that, there are other aspects how to treat time. For instance, *discrete* time vs. *continuous* time. We mostly deal with discrete time. Systems proceed step by step, from one state to other, and that's discrete behavior. Time can be see as time *instances* or points in time (as we mostly do), but there are also logics and models what work with time *intervals*.

Another distinguishing characteristic is the following. Most logics will be able to express properties concerning the *future*. Like: "never will there be a deadlock", meaning "never in the future". But there also also versions that can (additionally) talk about the past.

The notion of *time* here, in the context of temporal logics in general and LTL in particular, is kind of abstract. Time is handled in a similar way as we did when introducing modal logics in general, i.e., as "relation" between states (or worlds): proceeding from one state to another via a transition means a "temporal step" insofar that the successor state is "after" the first state. But the time is not really measured, i.e., there is no notion of how long it takes to do a steps. So, the systems and correspondingly the logics talking about their behavior are not *real-time* systems or real-time temporal logics. There exists, however, variants of temporal logics which handle real-time, including versions of real-time LTL, but they won't (probably) occur in this lecture.

3.2 Linear-time temporal logics

In **linear temporal logic (LTL)**, also called *linear-time temporal logic*, is one of the most prominent temporal logics used in model checking. It's supported notably by Spin, and other model-checkers. It was the first, or one of the first temporal logics taken up in computer science, and there are variations of that logics.

With it, we can describe properties like, for instance, the following: assume time is a *sequence* of discrete points i in time, then: if i is *now*,

- p holds in i and every following point (the future)
- p holds in i and every preceding point (the past)

$$\begin{tikzcd}
\ldots \arrow[r] & \bullet^{p}_{i-2} \arrow[r] & \bullet^{p}_{i-1} \arrow[r] & \bullet^{p}_{i} \arrow[r] & \bullet^{p}_{i+1} \arrow[r] & \bullet^{p}_{i+2} \arrow[r] & \ldots
\end{tikzcd}$$

Time here is *linear* and *discrete*. One can consequently just use ordinary natural numbers (or integers) to index the points in time. We will mostly be concerned with properties referring to the future, i.e., we won't go much into past-time LTL resp. versions of LTL that allow to speak about the future *and* the past.

3.2.1 Syntax

As before, we start with the syntax of the logic at hand, it's given by a grammar, as usual. We assume some underlying "core" logic. Focusing on the temporal part of the logic, we don't care much about that underlying core. Practically, when it comes to automatically checking, the choice of the underlying logic of course has an impact. But we treat the handling of the underlying logic as *orthogonal*. We will mostly we just assume *propositional* logic as core logic, but the LTL-part of the story would not change if we used first-order logic or another logic.

The first thing to extend is the syntax: we have formulas ψ of said underlying core, and then we extend it but the temporal operators of LTL, adding $\Box$, $\Diamond$, $\bigcirc$, U, R, and W. So the syntax of (a version of) LTL is given by the grammar of Table 3.1.

ψ			propositional/first-order formula
φ	$::=$	ψ	formulas of the "core" logics
	$\mid$	$\neg\varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \ldots$	boolean combinations
	$\mid$	$\bigcirc\varphi$	next φ
	$\mid$	$\Box\varphi$	always φ
	$\mid$	$\Diamond\varphi$	eventually φ
	$\mid$	$\varphi\ U\ \varphi$	"until"
	$\mid$	$\varphi\ R\ \varphi$	"release"
	$\mid$	$\varphi\ W\ \varphi$	"waiting for", "weak until"

Table 3.1: LTL syntax

As in earlier logics, one can ponder, whether the syntax is *minimal*, i.e., do we need all the operators, or can some be expressed as syntactic sugar by using others? The answer is: the syntax is *not minimal*, some operators can be left out and we will see that later. For a robust answer to the question of minimality, we need to wait until we have clarified the meaning, i.e., until we have defined the semantics of the operators.

Remark on the syntax

We had mentioned it already earlier, in the section covering modal logics. The operators always and eventually from LTL are written often as $\Box$ and $\Diamond$, but they are interpreted slightly different from the interpretation of box and diamond in conventional modal logic. Since we are working in a *linear* structure, the $\bigcirc$-operator corresponds to both box and diamond over the transition relation, both collapsed. One can also interpret $\Box$ and $\Diamond$ from LTL to correspond to the traditional modalities over the **transitive closure** of the one-step successor relation.

3.2.2 Semantics

For the semantics, we need to define a satisfaction relation $\models$ between "models" and LTL formulas. In principle, we know how that works, having seen similar definitions when discussing modal logics in general (using Kripke frames, valuations, and Kripke models).

Now, that we are dealing with a *linear* temporal logic, the Kripke frames are of linear structure. As usual, what kind of *valuations* we employ would depend on the underlying logics. For example for propositional LTL, one needs an interpretation of the propositional atoms per world, for first-order LTL, one needs a choice of the free variables in the terms and formulas (the signature and its interpretation does not change when going from one world to another, only, potentially, the values of the variables).

That's also what we do next, except that we won't use explicitly the terminology of *Kripke frame* or Kripke model. We simply assume a sequence of discrete time points, indexed by natural numbers. So the numbers i, $i+1$, etc. denote the worlds, and the accessibility relation simply connects a "world" i with its successor world $i+1$.

As was done with Kripke models, we then need a valuation per world, i.e., per time point. In the case of propositional LTL, it's a mapping from propositional variables to the boolean values $\mathbb{B}$. To be consistent with common terminology, we call such a function of type $P \to \mathbb{B}$ here not a valuation as we did mostly before, but a **state** (but see also the side remarks about terminology below). Let's use the symbol s to represent such a state or valuation. A *model* then provides a state per world, i.e., a mapping

$$\mathbb{N} \to (P \to \mathbb{B}) \ . \tag{3.1}$$

This is equivalently represented as an infinite sequence of the form

$$s_0 s_1 s_2 \ldots \tag{3.2}$$

where s_0 represents the state at the zero'th position in the infinite sequence, s_1 at the position or world one after that, etc. Such an infinite sequence of states is called **path**, and we use letters π, π' etc. to refer to paths. It's important to remember that paths are *infinite*.

Some remarks on terminology: paths, states, and valuations The notions of states and paths ... are slightly differring in the general literature. It's not a big problem as the used terminology is not incompatible, just sometimes not in complete agreement.

For example, there is a notion of path in connection with graphs. Typically, a path in a graph from a node n_1 to a node n_2 is a sequence of nodes that follows the edges of the given graph and that starts at n_1 and ends in n_2. The length of the path is the number of *edges* (and with this definition, the *empty* paths from n to n contains one node, namely n). There maybe alternative definitions of paths in "graph theory" (like sequences of nodes instead of edges). In connection with our current notion of paths, there are 3 major differences. Our paths are *infinite*, whereas when dealing with graphs, a path normally is understood as a *finite sequence*. There is no fundamental reason for not considering (also) infinite paths in graphs (and some people of course do), it's just that the standard case there is finite sequences, and therefore the word *path* is reserved for those. LTL, on the other hand, deals with ininite sequences, and consequently uses the word paths for those.

The other difference is that a path here is not defined as "a sequence of nodes connected *by edges*". It's simply an infinite sequence of valuations (and the connection is just by the position in the sequence), there is no question of "is there a transition from state at place i to that of at place $i+1$ (or one may see it as implicitly given by the "underlying" implict linear Kripke-frame, where there is an edge from i to $i+1$). Later, when we connect the current notion of paths to "path through a transition system", then the states in that infinite sequence need to arise by connecting transistions or edges in the underlying transition system or graph.

Finally, of course, the conventional notion of path in a graph does not speaks of valuations, it's just a sequence of nodes. If N is the set of nodes of a graph, and $\mathbb{N}_n$ the finite set $\{i \in \mathbb{N} \mid i < n\}$, then a traditional path (of length n) in graphs is a function $\mathbb{N}_n \to N$ such that it "follows the edges".

There are other names as well, when it comes to linear sequences of "statuses" when running a program. Those include *runs*, *executions* (also traces, logs, histories etc.). Sometimes they correspond to sequences of edges (for instance, containing transition labels only). Sometimes they correspond to sequences of "nodes" (containing "status-related" information like here), sometimes both.

Anyway, for us right now and for propositional LTL), a path π, as given in Definition 3.2.1 is an infinite sequence of states (or valuations).

Paths and computations

Definition 3.2.1 (Path)**.** A *path* is an infinite sequence

$$\pi = s_0, s_1, s_2, \ldots$$

of states. It can be seen as a mapping of type $\mathbb{N} \to (P \to \mathbb{B})$.
π^k denotes the suffix path $s_k, s_{k+1}, s_{k+2}, \ldots$. π_k denotes the state s_k.

It's intended that (later) paths represent behavior of programs resp. "going" through a transition system. A transitions system is a graph-like structure (and may contain cycles), and a path can be generated following the graph structure. In that sense it corresponds to the notion of *paths* as known from graphs (remember that the mathematical notions of graph corresponds to Kripke frames). Note, however, that we have defined path *independent* from an underlying program or transition system. It's not a "path through a transition system", but it's simply an infinite sequence of state (maybe caused by a transition system or maybe also not).

Now, what's a *state* then? It depends on what kind of LTL we are doing, basically propositional LTL or first-order LTL. A state basically is the interpretation of the underlying logic in the given "world", i.e., the given point in time (where time is the index inside the linear path). In propositional logic, the state is the interpretation of the propositional symbols (or the set of propositional symbols that are considered to be true at that point). For first-order logic, it's a valuation of the free variables at that point. When one thinks of modelling programs, then that corresponds to the standard view that the state of an imperative program is the value of all its variables (= state of the memory).

The satisfaction relation $\pi \models \varphi$ is defined inductively over the structure of the formula. We assume that for the formulas of the "underlying" core logic, we have an adequate satisfaction relation $\models_{\mathsf{ul}}$ available, that works on *states*. Note that in case of first-order logic, a signature and its *interpretation* is assumed to be fixed.

Definition 3.2.2 (Satisfaction)**.** A path π satisfies an LTL formula φ, written $\pi \models \varphi$, under the following conditions:

$$
\begin{array}{lll}
\pi \models \psi & \text{iff} & \pi_0 \models_{\mathsf{ul}} \psi \text{ with } \psi \text{ from the underlying core language} \\
\pi \models \neg\varphi & \text{iff} & \pi \not\models \varphi \\
\pi \models \varphi_1 \wedge \varphi_2 & \text{iff} & \pi \models \varphi_1 \text{ and } \pi \models \varphi_2 \\
\pi \models \bigcirc\varphi & \text{iff} & \pi^1 \models \varphi \\
\pi \models \varphi_1 \; U \; \varphi_2 & \text{iff} & \pi^k \models \varphi_2 \text{ for some } k \geq 0, \text{ and} \\
 & & \pi^i \models \varphi_1 \text{ for every } i \text{ such that } 0 \leq i < k
\end{array}
$$

The definition of $\models$ covers $\bigcirc$ and U as the *only* temporal operators. It will turn out that these two operators are complete insofar that they can express the remaining operators from the syntax, at least the remaining temporal ones. Those other operators are $\Box$, $\Diamond$, R, and W, according to the syntax we presented earlier. That's a common selection of operators for LTL, but there are sometimes even more added for the sake of convenience and to capture commonly encountered properties a user may wish to express.

We could explain those missing operators as syntactic sugar, showing how they can be macro-exanded into the core operators. What we (additionally) do first is giving a *direct* semantic definition of their satisfaction. As mentioned already earlier, the two important temporal operators "always" and "eventually" are written symbolically like the modal operators necessity and possibility, namely as $\Box$ and $\Diamond$, but their interpretation is slightly different from them. Their semantic definition is straightforward, referring to *all* resp. for *some* future point in time.

The release operator is the dual to the until operator, but is also a kind of "until" only with the roles of the two formulas exchanged. Intuitively, in a formula $\varphi_1 \; R \; \varphi_2$, the φ_1 "releases" φ_2's need to hold, i.e., φ_2 has to hold up until and *including* the point where φ_1 first holds and if φ_1 never holds (i.e., never "realeases φ_2), then φ_2 has to hold forever. If there a point where φ_1 is first true and thus releases φ_2, then at that "release point" both φ_1 and φ_2 have to hold. Furthermore, it's a "weak" form of a "reverse until" insofar that it's not required that φ_1 ever releases φ_2.

Definition 3.2.3 (Satisfiability of further operators)**.**

$$\begin{aligned}
&\pi \models \Box\varphi && \text{iff } \pi^k \models \varphi \text{ for all } k \geq 0\\
&\pi \models \Diamond\varphi && \text{iff } \pi^k \models \varphi \text{ for some } k \geq 0\\[1em]
&\pi \models \varphi_1\ R\ \varphi_2 && \text{iff for every } j \geq 0,\\
& && \quad \text{if } \pi^i \not\models \varphi_1 \text{ for every } i < j \text{ then } \pi^j \models \varphi_2\\[1em]
&\pi \models \varphi_1\ W\ \varphi_2 && \text{iff } \pi \models \varphi_1\ U\ \varphi_2 \text{ or } \pi \models \Box\varphi_1
\end{aligned}$$

Validity and semantic equivalence

Now with the semantics nailed down, we can transport other semantical notions to LTL. Validity, as usual captures "unconditional truth-ness" of a formula. In this case, it thus means, that a formula holds for all paths.

Definition 3.2.4 (Validity and equivalence)**.**

- φ is *(temporally) valid*, written $\models \varphi$, if
$$\pi \models \varphi \text{ for all paths } \pi.$$
- φ_1 and φ_2 are *equivalent*, written $\varphi_1 \sim \varphi_2$, if
$$\models \varphi_1 \leftrightarrow \varphi_2 \text{ (i.e. } \pi \models \varphi_1 \text{ iff } \pi \models \varphi_2\text{, for all } \pi\text{).}$$

Example 3.2.5. $\Box$ distributes over $\wedge$, while $\Diamond$ distributes over $\vee$.

$$\Box(\varphi \wedge_1 \varphi_2) \sim (\Box\varphi_1 \wedge \Box\varphi_2)$$
$$\Diamond(\varphi_1 \vee \varphi_2) \sim (\Diamond\varphi_1 \vee \Diamond\varphi_2)$$

In some way, especially from the perspective of model checking, valid formulas are "boring". They express some universal truth, which may be interesting and gives insight to the logics. But a valid formula is also *trivial* in the technical sense in that it does not express any interesting properties. After all, it's equivalent to the formula $\top$. In other words, it's equally useless as a specification as a contradictory formula (one that is equivalent to $\bot$), as it holds for all systems, no matter what.

Valid formulas may still be useful. If one knows that one property implies another (resp. that $\varphi_1 \to \varphi_2$ is valid), one could model-check using formula φ_1 (which might be easier), and use that to establish that also φ_2 holds for a given model. But still, unlike in logic and theorem proving, the focus in model checking is not so much on finding methods to derive or infer valid formulas.

The following illustrations are for propositional LTL, where we use p, q and similar for propositional atoms. We also indicate the states by "labelling" the corresponding places in the infinite sequence by mentioning the propositional atoms which are assumed to hold

$$\begin{tikzcd}
\pi \models \Box p & \bullet_0^p \arrow[r] & \bullet_1^p \arrow[r] & \bullet_2^p \arrow[r] & \bullet_3^p \arrow[r] & \bullet_4^p \arrow[r] & \dots \\
\pi \models \Diamond p & \bullet_0 \arrow[r] & \bullet_1 \arrow[r] & \bullet_2 \arrow[r] & \bullet_3^p \arrow[r] & \bullet_4 \arrow[r] & \dots \\
\pi \models \bigcirc p & \bullet_0 \arrow[r] & \bullet_1^p \arrow[r] & \bullet_2 \arrow[r] & \bullet_3 \arrow[r] & \bullet_4 \arrow[r] & \dots \\
\pi \models p\ U\ q & \bullet_0^p \arrow[r] & \bullet_1^p \arrow[r] & \bullet_2^p \arrow[r] & \bullet_3^{q,(p)} \arrow[r] & \bullet_4 \arrow[r] & \dots \\
\pi \models p\ R\ q & \bullet_0^q \arrow[r] & \bullet_1^q \arrow[r] & \bullet_2^q \arrow[r] & \bullet_3^{p,q} \arrow[r] & \bullet_4 \arrow[r] & \dots \\
\pi \models p\ W\ q & \bullet_0^p \arrow[r] & \bullet_1^p \arrow[r] & \bullet_2^p \arrow[r] & \bullet_3^p \arrow[r] & \bullet_4^p \arrow[r] & \dots
\end{tikzcd}$$

Figure 3.1: Illustration of LTL formulas

at that point (and leaving out those which are not). However, those are *illustrations*. For instance, when illustrating $\pi \models \bigcirc p$, the illustration shows that p holds at the second point in time (the one indexed with 1). The absence of p for $i = 0$ in the picture is *not* meant to say that it's required that $\neg p$ must hold at $i = 0$ etc. Similar remarks apply to the other pictures.

For the last three examples from Figure 3.1, we should remark the following. For the case of U, the q is required to show up after a **finite** initial sequence of p's. For the R, the sequence of q's can be infinite, and in this case the p does not show up. Also for W, the weak form of until, the sequence of p can be infinite, which in this case is obvious since we have defined $p\ W\ q$ as $p\ U\ q \vee \Box p$.

3.2.3 The Past

The LTL presentation so far focuses on "future" behavior, and the "future" will also be the focus when dealing with alternative logics (like CTL or the μ-calculus (not 2021)). In the section we shortly touch upon switching perspective in that we use LTL to speak about the past; similar switches could be done also for the mentioned other logics, among others. We don't go too deep.

In a way, there is not much new here, if we just talk about the past instead of the future. If we take a transition system (or graph or Kripke structure), in a way it's just "reversing the arrows" (i.e., working with the reverse graph etc.). It corresponds to "run the program in reverse", and then future and past swap their places, obviously. Basically, the same conceptual picture can be done for LTL, considering the linear paths "backwards". Of course, instead of talking about the next state, but backwards (and using reverse paths as models), it's probably clearer if we leave paths as model unchanged, but speak about the *previous* state instead. In general, introduce *past* versions of other temporal operators: eventually (in the future) becomes sometime earlier in the past, etc. In this way, we can also get a logic which allows to express properties that mix requirements about the future and the past.

It may seem as if the future and the past were basically the same. However, it's actually not true that future and past are 100% symmetric (as we perhaps implied by the above discussion about reversing the perspective). What is asymmetric is the notion of *path*. It is an infinite sequence (or a function $\mathbb{N} \to (P \to \mathbb{B})$, but that's asymmetric insofar it has a start point, but no end. That will require a quite modest variation the way the satisfaction relation $\models$ is defined for the past operators. Apart from that, there is not really much new.

As for the semantics now, we cannot simply use paths, we need pairs (π, j) of paths and positions, where the position indicates the point of "now" [25]. Let's write $\Box^{-1}$, $\Diamond^{-1}$ etc. for **past operators**. The definition of the satisfaction relation is straightforward

$$\begin{array}{lll} (\pi, j) \models \Box^{-1}\varphi & \text{iff} & (\pi, k) \models \varphi \text{ for all } k,\ 0 \le k \le j \\ (\pi, j) \models \Diamond^{-1}\varphi & \text{iff} & (\pi, k) \models \varphi \text{ for some } k,\ 0 \le k \le j \end{array}$$

However, it can be shown that for any formula φ, there is a *future-formula*, a formula without past operators, ψ such that

$$(\pi, 0) \models \varphi \quad \text{iff} \quad (\pi, 0) \models \psi$$

Example 3.2.6 (Past and future LTL). Let's consider as example the property

$$\Box(\varphi \to \Diamond^{-1}\psi) \tag{3.3}$$

mixing future and past operators. It expresses that if a φ occurs at a point, there must have been a point before, where ψ held, and that implication holds always.

That property can be expressed equivalently without past operators, using the (future) release operator:

$$\psi\ R\ (\varphi \to \psi)\ . \tag{3.4}$$

I.e., $(\pi, 0) \models \Box(\varphi \to \Diamond^{-1}\psi)$ iff $(\pi, 0) \models q\ R\ (p \to q)$. $\Box$

Figure 3.2 illustrates the two equivalent formulations of the property.

$$\begin{tikzcd}
\bullet^{p\to\Diamond^{-1}q} \arrow[r] & \bullet^{p\to\Diamond^{-1}q} \arrow[r] & \bullet^{p\to\Diamond^{-1}q} \arrow[r] & \bullet^{p\to\Diamond^{-1}q} \arrow[r] & \bullet \arrow[r] & \dots \\
\bullet^{p\to q} \arrow[r] & \bullet^{p\to q} \arrow[r] & \bullet^{p\to q, q} \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \dots
\end{tikzcd}$$

Figure 3.2: Illustration of the formulas from Example 3.2.6

3.2.4 Some LTL examples

Let's have a look at a few temporal properties and how they can be captured by LTL. LTL has a formal syntax and semantics, one can capture thus temporal properties unambiguously and precisely. Often, though, one starts with informal requirements and it can be difficult to correctly capture informally stated requirements in temporal logic.

Let's try to capture the vaguely formulated property

"when p then q."

It's not really clear that that is supposed to mean. Here are some more or less plausible formalizations:

$\varphi \to \psi$	$\varphi \to \psi$ holds in the initial state.
$\Box(\varphi \to \psi)$	$\varphi \to \psi$ holds in every state.
$\varphi \to \Diamond\psi$	φ holds in the initial state, ψ will hold in some state.
$\Box(\varphi \to \Diamond\psi)$	(*"response"*)

The last formulation is also called a **response** property (and sometimes one uses a special notation for that, namely $\varphi \rightsquigarrow \psi$). It is not obvious, which one of them (if any) is necessarily what is intended.

Let's do a few more examples.

Example 3.2.7. $\varphi \to \Diamond\psi$: If φ holds initially, then ψ holds eventually.

$$\begin{tikzcd}
\bullet^{\varphi} \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet \arrow[r] & \dots
\end{tikzcd}$$

This formula will also hold in every path where φ does not hold initially.

$$\begin{tikzcd}
\bullet^{\neg\varphi} \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \dots
\end{tikzcd}$$

□

Example 3.2.8 (Response). $\Box(\varphi \to \Diamond\psi)$

Every φ-position coincides with or is followed by a ψ-position.

$$\begin{tikzcd}
\bullet \arrow[r] & \bullet^{\varphi} \arrow[r] & \bullet \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet \arrow[r] & \bullet^{\varphi,\psi} \arrow[r] & \dots
\end{tikzcd}$$

This formula will also hold in every path where φ never holds.

$$\begin{tikzcd}
\bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \dots
\end{tikzcd}$$

$\square$

Example 3.2.9 (∞). $\Box\Diamond\psi$ There are infinitely many ψ-positions.

$$\begin{tikzcd} \bullet^{\psi} \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet \arrow[r] & \ldots \end{tikzcd}$$

$\square$

Note that this formula can be obtained from the previous one, $\Box(\varphi \to \Diamond\psi)$, by letting $\varphi = \top$: $\Box(\top \to \Diamond\psi)$.

Before we continue with some more examples, let's pause for a while and look at the previous three examples, and compare them with a simpler one, like $\Box p$. It's an example of an *invariant property* or just an invariant, requiring that p always holds.

The previous examples, maybe especially the last one about about infinitely many occurrences of something, feel more complex, not just because there are the formula is bigger. Here we discuss and give arguments what's distinguishes the formulas of the examples from, for instance, the invariance $\Box p$.

Let's assume we have some system and we want to check whether those properties from the examples hold. Let's assume we don't do model checking, but something simpler like *monitoring* or *run-time verification*. That means, we are dealing with a *running* system and we keep an eye on the execution path and the temporal formula to see whether it holds or not. Of course, it's a program, not us, that keeps an eye on the situation, and that's the run-time *monitor*. The task is simpler than model checking insofar run-time verification deals with one execution, whereas model checking attempts to systematically explore all of them. Checking only one execution, one cannot hope for full verification of a program. One cannot even hope to establish that the monitored execution satisfies the property. Why's that? That's because paths are *infinite*. The only thing one can hope for is that the monitor detects a *violation* of the specification, and then flags an alarm or takes corrective actions. That is possible for the an invariance property $\Box p$, but not for the previous examples. For instance, the property from Example 3.2.9 states that there are infinitely points in time where the property hold. Monitoring can neither confirm that, it would take an infinite amount of time, nore can it detect a violation. It's a property one cannot monitor. Technically, that distinction can be captured by classifying properties as safety properties on the one hand, the ones that can be monitored, and *liveness* properties, those that can't.

Obviously, properties can contain both aspects, like in a conjunction of $\Box\varphi$ and one of the three previous examples. However, it can be shown that every LTL formula can be equivalently expressed by a conjunction of a pure safety property and a pure liveness properties. We will talk more precisely about safety vs. liveness later.

The discussion here reasoned that liveness properties like the one from Example 3.2.9 are intuitively more complex, by pointing out that they inuitively cannot be monitored, unlike the safety properties. LTL model checking can check all of LTL, safety and liveness, at least for finite-state systems, but also there, liveness properties are more tricky to deal with. Model-checking safety properties are actually almost a non-brainer. Take the simplest safety property $\Box p$. Given a finite-state system, how does one do that? Just exploring at all possible reachable states, see if any of them violates p. If so, the property does not hold, if one finds no violation, the property holds. So that's a straightforward graph search. Of course the state-graph may be immensely huge. So it requires clever representation techniques and exploration strategies to do that in practice. But the problem itself is a graph search, something one learns in the first or second semester, if not earlier.

It's not obvious at all how one can check liveness properties, like the one from Example 3.2.9 by a finite system. We will see that later, when we address the algorithmic problem to LTL model checking.

Let's continue with some more examples- One can check one's intuition also on the following examples, whether they are safety or liveness properties (or neither).

Example 3.2.10 (Permanence)*.* Eventually φ will hold *permanently:* $\Diamond\Box\varphi$.

$$\begin{tikzcd}
\bullet \arrow[r] & \bullet^{\varphi} \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet^{\varphi} \arrow[r] & \bullet^{\varphi} \arrow[r] & \bullet^{\varphi} \arrow[r] & \ldots
\end{tikzcd}$$

Equivalently formulated: there are *finitely* many $\neg\varphi$-positions. □

Example 3.2.11*.* $(\neg\varphi)\ W\ \psi$

The first φ-position must coincide or be preceded by a ψ-position.

$$\begin{tikzcd}
\bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\varphi} \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \ldots
\end{tikzcd}$$

φ may never hold

$$\begin{tikzcd}
\bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \ldots
\end{tikzcd}$$

□

Example 3.2.12*.* $\Box(\varphi \to \psi\ W\ \chi)$

Every φ-position initiates a sequence of ψ-positions, and if terminated, by a χ-position.

$$\begin{tikzcd}
\bullet \arrow[r] & \bullet^{\varphi,\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\chi} \arrow[r] & \bullet \arrow[r] & \bullet^{\varphi,\psi} \arrow[r] & \ldots
\end{tikzcd}$$

The sequence of ψ-positions need not terminate.

$$\begin{tikzcd}
\bullet \arrow[r] & \bullet^{\varphi,\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \ldots
\end{tikzcd}$$

□

Example 3.2.13 (Nested waiting-for). A *nested waiting-for formula* is of the form

$$\Box(\varphi \to (\psi_m \; W \; (\psi_{m-1} \; W \cdots (\psi_1 \; W \; \psi_0) \cdots))),$$

where $\varphi, \psi_0, \ldots, \psi_m$ in the underlying logic. For convenience, we write

$$\Box(\varphi \to \psi_m \, W \, \psi_{m-1} \, W \, \cdots \, W \, \psi_1 \, W \, \psi_0).$$

$$\begin{tikzcd}
\ldots \arrow[r] & \bullet^{\varphi,\psi_m} \arrow[r] & \bullet^{\psi_m} \arrow[r, dotted] & \bullet^{\psi_m} \arrow[r] & \bullet^{\psi_{m-1}} \arrow[r, dotted, no head] & \bullet^{\psi_{m-1}} \arrow[r] & \ldots
\end{tikzcd}$$

$$\begin{tikzcd}
\ldots \arrow[r] & \bullet^{\psi_2} \arrow[r, dotted, no head] & \bullet^{\psi_2} \arrow[r] & \bullet^{\psi_1} \arrow[r, dotted, no head] & \bullet^{\psi_1} \arrow[r] & \bullet^{\psi_0} \arrow[r] & \ldots
\end{tikzcd}$$

Every φ-position initiates a succession of intervals, beginning with a ψ_m-interval, ending with a ψ_1-interval and possibly terminated by a ψ_0-position. Each interval may be empty or extend to infinity. $\Box$

3.2.5 Dual connectives and complete sets of connectives

In logics, not just in modal logics, one finds often pairs of operators, which are each other's opposites. The two quantifiers $\forall$ and $\exists$ are an example. Opposite does not mean that one is the negation of the other. Clearly $\neg\forall\varphi$ means something else than $\exists\varphi$. But it corresponds to $\exists\neg\varphi$. This form of being in opposition with each other is called *duality*.

Definition 3.2.14 (Duals)**.** For binary boolean connectives $\circ$ and $\bullet$, we say that $\bullet$ is the *dual* of $\circ$ if

$$\neg(\varphi \circ \psi) \sim (\neg\varphi \bullet \neg\psi).$$

Similarly for unary connectives: $\bullet$ is the dual of $\circ$ if $\neg \circ \varphi \sim \bullet\neg\varphi$.

Duality is *symmetric*, i.e., if $\bullet$ is the dual of $\circ$ then $\circ$ is the dual of $\bullet$. Thus we may refer to two connectives as dual to each other.

The $\circ$ and $\bullet$ operators are meant as "placeholders". One can have a corresponding notion of duality for the unary operators $\Diamond$ and $\Box$, and even for null-ary "operators".

Concerning propositional connectives, $\wedge$ and $\vee$ are duals, and $\neg$ is (trivially) its own dual.

$$\neg(\varphi \wedge \psi) \sim (\neg\varphi \vee \neg\psi).$$

If we use $\not\leftarrow$ for the negated reverse implication, then that is the dual of $\to$?:

$$\begin{aligned}\neg(\varphi \not\leftarrow \psi) &\sim \varphi \leftarrow \psi \\ &\sim \psi \rightarrow \varphi \\ &\sim \neg\varphi \rightarrow \neg\psi \ .\end{aligned}$$

A set of connectives is *complete* (for boolean formulae) if every other connective can be defined in terms of them. Our set of connectives is complete (e.g., $\not\leftarrow$ can be defined), but also subsets of it, so we don't actually need all the connectives.

Example 3.2.15. The set of connectors $\vee$ and $\neg$ is complete. $\wedge$ is the dual of $\vee$, as mentioned. The other connectors can be expressed as follows:

$$\begin{array}{rcl}\varphi \rightarrow \psi & \sim & \neg\varphi \vee \psi \\ \varphi \leftrightarrow \psi & \sim & (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) \\ \top & \sim & p \vee \neg p \\ \bot & \sim & p \wedge \neg p\end{array}$$

□

Actually, it's not just a complete set, it's also a minimal complete selection in the sense that one cannot remove one of the two connectors without loosing completeness. As is well-known, it's not the only minimal complete choice of operators. An even smaller selection is the set consisting just of NAND (or just of NOR).

In LTL, the two operators $\Box$ and $\Diamond$ are duals:

$$\neg\Box\varphi \sim \Diamond\neg\varphi \quad \text{and} \quad \neg\Diamond\varphi \sim \Box\neg\varphi \tag{3.5}$$

As a side remark: the symbols $\Box$ and $\Diamond$ are also part of modal logics in general, with a different interpretation. But also there, both are duals of each other.

Back to LTL, $\Box$ and $\Diamond$ are not the only temporal duals. Also U and R are each other's duals.

$$\neg(\varphi\ U\ \psi) \sim (\neg\varphi)\ R\ (\neg\psi) \quad \text{and} \quad \neg(\varphi\ R\ \psi) \sim (\neg\varphi)\ U\ (\neg\psi) \tag{3.6}$$

We don't need all our temporal operators either:

Proposition 1 (Complete set of LTL operators)**.** The set of operators $\vee, \neg, U$, and $\bigcirc$ is *complete* for LTL.

Proof. The remaining temporal operators can be expressed as follows:

$$\begin{array}{rcl}\Diamond\varphi & \sim & \top\ U\ \varphi \\ \Box\varphi & \sim & \bot\ R\ \varphi \\ \varphi\ R\ \psi & \sim & \neg(\neg\varphi\ U\ \neg\psi) \\ \varphi\ W\ \psi & \sim & \Box\varphi \vee (\varphi\ U\ \psi)\end{array}$$

The completeness for the propositional connectives has been covered earlier. □

invariant	$\Box\varphi$
example of a liveness property	$\Diamond\varphi$
obligation	$\Box\varphi \vee \Diamond\psi$
recurrence	$\Box\Diamond\varphi$
persistence	$\Diamond\Box\varphi$
reactivity	$\Box\Diamond\varphi \vee \Diamond\Box\psi$

Table 3.2: Classification of LTL properties

3.2.6 Classification of properties

We have seen a couple of examples of specific LTL formulas, i.e., specific properties. Specific "shapes" of formulas are particularly useful or common, and they sometimes get specific names (like "response" or "permanence"). If we take $\bigcirc$ and U as a complete core of LTL, then already the shape $\top\ U\ \varphi$ is so useful that it does not only deserve a special name, it even has a special syntax or symbol, namely $\Diamond$. We have encountered other examples before as well (like *permanence*) and in the following we will list some more.

Another very important classification or characterization of LTL formulas is the distinction between *safety* and *liveness*. Actually, one could see it not so much as a characterization of LTL formulas, but of *properties* (of paths). LTL is a specific notation to describe properties of paths (where a property corresponds to a set of paths). Of course not all sets of paths are expressible in LTL (why not?). The situation is pretty analogous to that of regular expressions and regular languages. Regular expressions play the rule of the syntax and they are interpreted as sets of finite words, i.e., as *properties* of words. Of course not all properties of words, i.e. languages, are in fact regular, there are non-regular languages (context-free languages etc.).

Coming back to the LTL setting: it's better to see the distinction between safety and liveness as a qualification on path *properties* (= sets or languages of infinite sequences of states), but of course, we then see which kind of LTL formulas are capturing a safety property or a liveness property.

Note (again) that "safety" or "liveness" is not property of a paths, it's a property of path properties, so to say. In other words, there will be no LTL formula expressing "safety" (it makes no sense), there are LTL formulas which correspond to a safety property, i.e., expresse a property that belongs to the set of all safety properties.

There is a kind of "duality" between safety and liveness in that safety is like the "opposite" of liveness, but it's not that properties fall exactly into these to categories. There are properties (and thus LTL formulas) that are neither safety properties nor lifeness properties.

Safety properties: never anything bad happens

Table 3.2 contains a few kind of formulas, where φ, ψ are non-temporal formulas. The second one, "eventually φ", is not liveness as such, but it's an example of a liveness property, maybe the simplest one. A similar discussion was done when we drew some parallels between Hoare-logic and dynamic logic.

The *invariant* is a prominent example of a *safety* property. Each invariant property is also a *safety* property. Some even use the words synonymously, but according to the consensus or majority opinion, invariant properties are a subset of safety properties. See for instance the rather authoritative textbook Baier and Katoen [2]. It's however true that *invariants* are perhaps the most typical, easiest, and important form of safety properties and they also represent the essence of them. In particular, if one informally stipulates that safety corresponds to

> "never something bad happens",

then that translates well to an invariant (namely the complete absence of the bad thing: "always not bad"). That characterization of safety is due to Lamport. We also focus on invariance (see Definition 3.2.16), without given the slightly more complex characterization of safety.

As we mentioned earlier, there is a connection to monitoring and run-time verification: Safety-properties are those that can be monitored.

Definition 3.2.16 (Invariant)**.** An *invariant* formula or just invariant is of the form

$$\Box\varphi \tag{3.7}$$

for some propositional (or more generally, non-temporal) formula φ.

Safety formula of the (simplified) form from Definition 3.2.16 express *invariance* of some *state property* φ: that φ holds in every state of the computation. A state property is just what the name implies. It's a property of individual points in time, as opposed to properties of paths. This distinction will be more prominent later in branching time logics like CTL or similar. State properties here are captured by the underlying logic, like propositional logics. Path properties are the the temporal properties.

Example 3.2.17 (Mutual exclusion)*. Mutual exclusion* is a safety property. Let c_i denote that process P_i is executing in the critical section. Then

$$\Box\neg(c_1 \land c_2)$$

expresses that it should always be the case that not both P_1 and P_2 are executing in their respective critical section.

Observe: the negation of a safety formula is a liveness formula; the negation of the formula above is the liveness formula

$$\Diamond(c_1 \land c_2)$$

which expresses that eventually it *is* the case that both P_1 and P_2 are executing in the critical section.

□

Liveness properties

As for safety properties, also here we content ourselves with showing a straightforward formula expressing liveness, not by characterizing liveness as a property of properties.

Definition 3.2.18 (Liveness)**.** A *liveness* formula is of the form

$$\Diamond\varphi \tag{3.8}$$

for some propositional (or more generally, non-temporal) formula φ.

Liveness formulae *guarantee* that some event φ eventually happens: that φ holds in at least one state of the computation.

Connection to Hoare logic After having shed some light on safety and liveness in the context of LTL, let's take a quick detour to one famous program logic, namely Hoare logic. As with most logics, there is not just the one Hoare logic, it's more style of logics with many realization.

The lecture does not formally introduce (a specific variant of a) Hoare logic, we just do some general remarks, to point out a parallel with safety and liveness.

Hoare-logic, named by it's inventor Tony Hoare, is a so-called program logics. It's formulas, ultimately, speak about states of programs, and how that states changes. BTW: since the logic is about talking and reasoning of state changes, it's used for imperative program. To do so, it makes use of *assertions*. That's nothing else than formulas in typically first-order logic (or some variation or fragment) with free variables. Also outside of (Hoare-logic) program verification, assertions are often supported by programming languages. Java, for instance, has a `assert` command. The argument of such a command is an expression of boolean type. That expression of course can contain some variables, and also Boolean connectors. Quantifiers are not supported, so it's a fragment of first-order logic, supporting propositional logic and the possiblity to talk about variables and functions (or methods) and relations. In that programmer's use of assertions, the purpose is not so much to do program verification or model checking, it's run-time assertion checking. If in a run, an assertion at some point in the program code is violated, an exception is raised.

Hoare logics *is* concerned with verification, and a Hoare-logic proof system provide rules that capture the effect of the constructs of the programming language on the program state, when the construct is executed.

That is done by relating the "assertion" before a statement with the one afterwards. This leads to the well-known pre-condition and post-condition formulation, typical for Hoare logic., especially appropriate for sequential prog

There are two ways a Hoare-logics specification on a program can be interpreted. The two ways are called *partial correctness* and *total correctness*. The first one states, that when starting in a state that satisfies the pre-condition, should the program terminate, then it will be in a state that satisfies the post-condition. Note that under the partial correctness interpretation, a pre- and post-condition specification has no opinion on non-terminating programs.

That's different for total correctness. Total correctness states that, when starting in a state that satisfies the precondition, then it's guaranteed that the program terminates, and, upon termination, satisfies the post-condition.

Of we use *terminated* as predicate to express termination, we can express partial and total correctness of a program P with pre-condition φ and post-condition ψ as follows

$$\varphi \to \Box(\mathit{terminated}(P) \to \psi) \quad \text{resp.} \quad \varphi \to \Diamond(\mathit{terminated}(P) \land \psi). \tag{3.9}$$

The formulas are formulas in LTL notation, in Hoare logic would express pre- and post-conditions notationally as $\{\ \varphi\ \}\ P\ \{\ \psi\ \}$ (a so-called Hoare-triple of pre-condition, post-condition and the program in the middle.)

The two formulas of equation (3.9) are not of the form called safety and liveness from Definitions 3.2.16 and 3.2.18. The form is sometimes called *conditional* safety and *conditional* liveness formula, being conditioned on the precondition φ.

When independent from the precondition, resp. assume φ to be "true", partial and total correctness are directly safety and liveness conditions of the form introduced.

Partial correctness may look as a weaker version of total corrctness, already the name seems to imply that. Perhaps surprisingly it turns out partial and total correctness are dual to each other. Let's focus on the *un*-conditional version, post-condition only. Then, it should become not so surprising after all: $\Box$ and $\Diamond$ are defined refering to *all* time points in the future, resp. to *some* time point in the future. And $\forall$ and $\exists$ are dual operators.

Let's introduce the following ammbreviations:

$$PC(\psi) \triangleq \Box(\mathit{terminated} \to \psi) \quad \text{and} \quad TC(\psi) \triangleq \Diamond(\mathit{terminated} \land \psi)$$

Then

$$\neg PC(\psi) \sim PC(\neg\psi) \quad \text{and} \quad \neg TC(\psi) \sim TC(\neg\psi)$$

Other classes of formulas: Obligation, recurrence and persistence, reactivity

Here the definition of obligations.

Definition 3.2.19 (Obligation)**.** A *simple obligation* formula is of the form

$$\Box\varphi \lor \Diamond\psi$$

for propositional (or generally non-temporal) formulas φ and ψ.

Such a property can equivalently be written as

$$\Diamond\varphi \to \Diamond\psi$$

The equivalent form $\Diamond\chi \to \Diamond\psi$ states that if some state satisfies χ, then some state must satisfy ψ.

Obligations subsume safety and liveness properties (and the form presented here).

Proposition 2. Every safety and liveness formula is also an obligation formula.

Proof. It's a consequence of the following equivalences.

$$\Box\varphi \sim \Box\varphi \lor \Diamond\bot \quad \text{and} \quad \Diamond\varphi \sim \Box\bot \lor \Diamond\varphi$$

and the facts that $\models \neg\Box\bot$ and $\models \neg\Diamond\bot$. □

To be recurrent means to occur over and over again. That can be caputured as follows.

Definition 3.2.20 (Recurrence)**.** A *recurrence* formula is of the form

$$\Box\Diamond\varphi$$

for some propositional (or more generally, non-temporal) formula φ.

It implies that there are *infinitely many* positions where φ holds. A *response* formula, of the form $\Box(\varphi \to \Diamond\psi)$, is equivalent to a recurrence formula, of the form $\Box\Diamond\psi$, if we allow χ to be a past-formula.

$$\Box(\varphi \to \Diamond\psi) \leftrightarrow \Box\Diamond(\neg\varphi)\ W^{-1}\ \psi$$

Next we express some form of fairness as a recurrence property. There are two main variants of fairness, weak and strong fairness. Generally it means that given two or more processes competing repeatedly on some resource, it's not that case that one of the competing processes is neglected all the time. If applying long enough or often enough for a resource must ultimately give access to the resource. Resource can mean various things, typical is processor time or acquiring a lock and getting access to a critical region. Equation (3.10) below is formulated referring to a step τ, that is *enabled* resp. is taken.

Example 3.2.21 (Weak fairness)*.* Weak fairness can be specified as the following recurrence formula.

$$\Box\Diamond(\mathit{enabled}(\tau) \to \mathit{taken}(\tau)) \tag{3.10}$$

□

An equivalent form is

$$\Box(\Box enabled(\tau) \to \Diamond taken(\tau)),$$

Fairness is a concept from concurrent or parallel systems, and can be expressed in LTL. Later we will see, fairness is *not* expressible in CTL, another important temporal logics.

As said, fairness means, that some actions or processes are not "unduly neglected" in favor of other actions or processes. To speak of fairness, there must be an element of choosing from alternatives (at least two). Another element is that the choice is *repeated*. If one has just one choice between two alternatives a *one* point in time and one is chosen, then that is neither fair nor unfair. However, if one *repeatedly* favors one alternative over the other, then that is unfair. Repeatedly refers actually to *infinitely often*. So just selecting alternative A 500 times before choosing B is not a violation of fairness, choosing *always* A in an infinite run is.

Often, the choices being made is between actions of different processes, choosing to execute a statement of process P_1 or of P_2 (if one has a system of 2 processes). So, that makes fairness a (typically desired) property of a scheduler. It's also a very "abstract" and general notion (taking an "infinitely long" perspective before one speaks about fairness or unfairness). As hinted at above, if a scheduler assigns 5 times as many slots to P_1 compared to P_2, thus executing it five times as fast as the other, one might see that as "unfair" in some sense, but not in the technical sense of fairness defined here. There are also "bounded" versions of fairness (which we don't treat here)

For fairness one distinguishes between a *strong* and *weak* version. It's connected with scheduling as well and the notion of *enabledness* (of transitions or steps). Transitions can be enabled or not. That in particular applies to actions in connection with *synchronization*. For instance, if a process is at a point where the next action is to take a lock, that step may or may not be enabled, depending on whether the lock is free or taken. An non-enabled action that is never scheduled does not count as unfair scheduling: it's not the scheduler's fault that, for instance, the owner of the lock does not give it back, thereby enabling the other processes waiting on the lock. It may be characterized as unfairness at some higher-level of looking at the program or characterized as different form of defect (maybe caused by a deadlock), but it's not unfairness on the level of scheduling actions: only actions that are enabled and could thereby be actually selected count when thinking about fairness.

Weak fairness means, an enabled action cannot *remain enabled* forever without being chosen. Strong fairness requires that an action cannot be *not chosen* if it is enabled *infinitely often* (but not necessarily continuously enabled). For instance, in the lock-illustration: if the lock is taken and released by other processes, then for some process waiting to take it, the corresponding action *toggles* between being enabled and disabled. *Weak* fairness does not require that the process ultimately get's the lock, but *strong* fairness would ensure that.

Weak and strong fairness are be "recurrent" (sorry for the pun) themes in dealing with concurrent system. They may also show up in later parts of the lecture, and we will also see how to express the weak variant in LTL later.

The next property we look at is called *persistence* or alternatively *stabilization*.

Definition 3.2.22 (Persistence)**.** A *persistence* formula is of the form

$$\Diamond\Box\varphi$$

for some propositional (or more generally non-temporal) formula φ.

Persistence of φ means, that at some point onwards, it will from then on always φ. This is another way of saying that only *finitely* many position satisfy $\neg\varphi$. In other words, persistance is dual to the property “infinitely often”, which is also called recurrence. Recurrence and persistence are duals:

$$\neg(\Box\Diamond\varphi) \sim (\Diamond\Box\neg\varphi) \quad \text{and} \quad \neg(\Diamond\Box\varphi) \sim (\Box\Diamond\neg\varphi)$$

The last class of properties is called *reactivity*, combining persistence and recurrence.

Definition 3.2.23 (Reactivity)**.** A *simple reactivity* formula is of the form

$$\Box\Diamond\varphi \lor \Diamond\Box\psi \tag{3.11}$$

for propositional (or more generally, non-temporal) formulas φ and ψ.

A very general class of formulas are conjunctions of reactivity formulae.

An equivalent formulation of the reactivity formula is

$$\Box\Diamond\psi' \to \Box\Diamond\varphi,$$

(or also $\Box(\Box\Diamond\chi \to \Diamond\psi)$) which states that if the computation contains infinitely many χ-positions, it must also contain infinitely many ψ-positions.

3.2.7 GCD Example

Let’s look at a small example, a really small one. It mainly intended to illustrate LTL one more time, connecting it to the behavior of a program. It’s not a typical program for LTL problems or model checking, insofar that it’s effectively a sequential program.

Program: GCD

```
      P :: [ in $a, b$ : integer where $a > 0, b > 0$ ;
             local $x, y$ : integer where $x = a, y = b$ ;
             out $g$ : integer ;
$P_1$ :: [ $l_0$ : [ $l_1$ : while $x \neq y$ do $l_2$ : [
                 [ $l_3$ : await $x > y$ ; $l_4$ : $x := x - y$ ; ]
                 or
                 [ $l_5$ : await $y > x$ ; $l_6$ : $y := y - x$ ; ] ]
             $l_7$ : $g := x$ ; $l_8$ : ] ] ]
```

The code is some form of pseudo-code for concurrent processes. The code should be roughly understandable. The body of the process is the lower half of the code. Basically a big while-loop, consisting of two alternatives, each consisting by of an `await`-statement. What exatcly that is does not matter right now. Effectively, in this example the body of the loop is a two armed alternative. At no point, both branches of the alternative are enabled, as at most one condition guarding the branches can be true. Additionally the branches are executed *atomically*, and that effectively makes the program sequential. So it's a fancy version of writing the good old GCD algorithm.

The variables a and b are the *inputs* of the program, and g is the output. As is often assumed and good practice, the input variables are immutable, and the output is likewise not mutated, up until the end when the result is to be returned.

The red l's that show up in the program are *not* part of the program. They are indicated to refer to different control-flow points in the code (and l stands for location or label). One can (and should) make a difference between labels in the program and locations in the following sense. In the program, there are labels that refer semantically to the same control-flow points. For instance l_0 and l_1. So while in the syntax of the program code, l_0 and l_1 are different labels, it's best to think of them to refer to the same location. We don't make that formal nor do we make explicit how to turn a program into a control-flow graph; the locations correspond to nodes in a control flow graph. Control-flow graphs are common intermediate representations inside a compiler, and actually also model checkers like Spin use control-flow graph internally. After all, Spin compiles a programming-language notation for an imperative, concurrent language to executable code, which executes not just the code, but runs the code being checked against the LTL formula and arranging for exploring the state-space (plus doing a lot of further tricks and optimizations).

Below is a computation π of our recurring GCD program. States are of the form $\langle l, x, y, g\rangle$, consisting of the location and the values of the local variables x and y as well as the variable g for the result. One can make the argument, that also the values of a and b are part of the state, but those don't change, so we can safely ignore them as uninteresting.

States are of the form $\langle l, x, y, g\rangle$, and the execution shown below is an (infinite) sequence of states. It's infinite, since for LTL, we need to work with infinite paths (which arise form the infinite execution). The GCD program is terminating, so it does not technically continues executing forever, but the usual "trick" to handle that is obvious: when the program terminates, one assumes that it continues doing steps without changes to the status. This is known as (one form of) *stuttering*, and we will cover aspects of stuttering later (but, as said, the idea as such is rather obvious).

Let $at(l_n)$ represent the formula expressing that the program is at location l_n, i.e., at a state of the form $\langle l_n, \_, \_, \_\rangle$. Let furthermore *terminated* represent the formula $at(l_8)$.

$$\begin{array}{rllll}
\pi: & \langle l_1, 21, 49, 0\rangle \to & \langle l_2^b, 21, 49, 0\rangle \to & \langle l_6, 21, 49, 0\rangle \to \\
& \langle l_1, 21, 28, 0\rangle \to & \langle l_2^b, 21, 28, 0\rangle \to & \langle l_6, 21, 28, 0\rangle \to \\
& \langle l_1, 21, 7, 0\rangle \to & \langle l_2^a, 21, 7, 0\rangle \to & \langle l_4, 21, 7, 0\rangle \to \\
& \langle l_1, 14, 7, 0\rangle \to & \langle l_2^a, 14, 7, 0\rangle \to & \langle l_4, 14, 7, 0\rangle \to \\
& \langle l_1, 7, 7, 0\rangle \to & \langle l_7, 7, 7, 0\rangle \to & \langle l_8, 7, 7, 7\rangle \to \cdots
\end{array}$$

Now, do the following properties hold for π? And why?

$\Box \mathit{terminated}$	(safety)
$\mathit{at}(l_1) \rightarrow \mathit{terminated}$	
$\mathit{at}(l_8) \rightarrow \mathit{terminated}$	
$\mathit{at}(l_7) \rightarrow \Diamond \mathit{terminated}$	(conditional liveness)
$\Diamond \mathit{at}(l_7) \rightarrow \Diamond \mathit{terminated}$	(obligation)
$\Box(\mathrm{gcd}(x,y) \doteq \mathrm{gcd}(a,b))$	(safety)
$\Diamond \mathit{terminated}$	(liveness)
$\Diamond\Box(y \doteq \mathrm{gcd}(a,b))$	(persistence)
$\Box\Diamond \mathit{terminated}$	(recurrence)

3.2.8 Exercises

Exercises

1. Show that the following formulas are (not) LTL-valid.
   a) $\Box\varphi \leftrightarrow \Box\Box\varphi$
   b) $\Diamond\varphi \leftrightarrow \Diamond\Diamond\varphi$
   c) $\neg\Box\varphi \rightarrow \Box\neg\Box\varphi$
   d) $\Box(\Box\varphi \rightarrow \psi) \rightarrow \Box(\Box\psi \rightarrow \varphi)$
   e) $\Box(\Box\varphi \rightarrow \psi) \vee \Box(\Box\psi \rightarrow \varphi)$
   f) $\Box\Diamond\Box\varphi \rightarrow \Diamond\Box\varphi$
   g) $\Box\Diamond\varphi \leftrightarrow \Box\Diamond\Box\Diamond\varphi$
2. A *modality* is a sequence of $\neg$, $\Box$ and $\Diamond$, including the empty sequence ϵ. Two modalities π and τ are *equivalent* if $\pi\varphi \leftrightarrow \tau\varphi$ is valid.
   a) Which are the non-equivalent modalities in LTL, and
   b) what are their relationship (ie. implication-wise)?

3.3 Automata-based model checking

3.3.1 LTL model checking

Given a formula φ in some logic and a model M, appropriate for the logic, *model-checking* in general is about answering the question whether

$$M \models^? \varphi, \tag{3.12}$$

i.e., whether the model satisfies the formula. In our case of temporal logics, the model takes the form of a transition system together with a starting state, there the transitions represent steps of a program or system (resp. perhaps an abstraction of a real system).

That's a straight enough problem, but how to address it depends on the concrete form of the models and the concrete logics.

As far as the models go, and independent from the logics, a big challenge is typically the size of the model or transition system: the **state space** of the model is simply *huge* for realistic problems, maybe even infinite. At least in principle infinite: if one has programs that work with numbers, then there are infinitely many. One can make the argument that on a computer, there are only finitely many representable numbers, like up-to `MAXINT`, but even if in this case the state space is actually finite, it's "practically infinite", since it's just too many. Similarly if one tries to verify programs with *dynamic* data structures, like lists and trees, etc. Also there, the argument that computer memory is limited and this there is an upper bound on the size of the trees may be technically true, but not really helpful in practice.

Of ourse, not all problems require dynamic data structures, some programs run on firmware or hardware, and indeed, model checking hardware-close systems and algorithms is seen as one important application area; software being way more challenging.

A prime application area for model checking is also concurrent systems (hardware or software). When dealing with natural numbers and data structures, which cause huge or infinite state spaces, other (or additional) techniques may be more adequate, used in combination with model checking. For concurrent system, there is another reason why state-spaces become huge. That's because of processes running concurrently can be executed with many different interleavings. What makes it tricky is that concurrency-related bugs are often ...

3.3.2 Automata-based LTL model checking

As said, there are many variations on the theme of temporal logic model checking: different logics, different kinds of target systems, different models, different techniques, different optimizations, ... We focus in this section on one prominent technique for LTL, **automata-based** model checking.

It's a method for *finite-state* models and it an example of so-called *explicit* state-space model checking. It's called *explicit* state model checking, as the states of the system are individually represented and the model-checking processe explores the state-space by one individual state after another. An alternative, which represents *sets* of states jointly, is known as *symbolic* model checking (but that's for later).

Let's assume, the transition systems come with one specific state, the initial state. Kripke frames don't come equipped with a start state, but for model checking transition systems representing programs, it's conventional to assume one specific state. It actually does not change the problem in the least, but streamlines the following discussions a bit, by avoiding that we have to say "assume a Kripke-structure M together with a state s..." over an over again. And as said, it's conventional for model checking systems anyway. So, a model M, a transition system, is now a defined like what we introduced as Kripke structure but with additionally an initial state.

With that triviality out of the way, let's ask ourselves: in the setting of LTL, with transition systems as models, and given the model-checking problem from equation (3.12), how could one go for it?

For simple LTL-formulas like invariance properties $\Box p$, it's conceptually pretty simple. We are given the model as a transition system, for each state we have the valuation for the propositional atoms. In particular we are given the status of p for each place. The invariance from equation (3.12) holds, if all paths in M, that start from its initial state s_0, satisfy p (see Definition 3.2.2). That's in general infinitely many paths, and each path itself is infinite.

But this infinity is not a big deal: one simply has to explore the transition system, starting from s_0, checking all states *reachable* from s_0. If all reachable states satisfy p, the property holds for all paths starting at s_0, i.e., $M \models \Box p$ holds. If the exploration discovers one place that violates p, then the model-checking question is answered negatively, and one can stop the exploration (unless one is interested in all places that violate p). As we assume that the system is finite, the problem is solved. There may be practical challenges, like the size of the transition system, but *deciding* whether $\Box p$ holds in a finite-state model is conceptually simple.

Of course, $\Box p$ is arguably the simplest temporal property, an easy instance of a safety property. Things don't look quite so easy with more complex properties. Let's take $\Diamond\Box p$ as example, expressing *infinitely many occurrences* of p on all path. How to check that? If we were interested in checking that there *exists* a path in M, starting from s_0, that contains infinitely many places where p holds, one could come up with the following: starting from s_0, try to reach a state s such that, starting from s, one finds a *cycle* in the graph, leading back to s such that on the cycle, there is a s' where p holds. One could arrange that more efficiently than going on a loop-finding mission for all reachable s's one after the other, but doing those loop-searches naively would already solve the problem. Of course to *decide* the question, one needs to explore, for each place s, all loops. Of course if one has run through one particular cycle one time, one does not have to run through the same cycle again, so one can restrict the seach to simple cycles and that makes the problem finite, so it seems doable.

We made plausible how to look for *some* path that satisfies $\Box p$. The standard form of LTL are not looking whether there exists a path, it is about checking *all* paths. One could try to address that similarly, checking more or less cleverly for reachable states that are contained in cycles in the graph.

All that may be plausible but we need a *general method* that works for all of LTL. That's the main topic of this chapter, automata-based LTL model checking. We start by laying out the general ideas behind the approach, details will come later.

The big picture: automata-based LTL model checking as a form of refutation

In a bird's eye view and very generally, the method can be seen as a **refutation** proof method. In the chapter covering non-temporal logics, we touched upon proof-systems, sketching rule-systems to derive *valid* formulas. Proof by refutation works differently, so the proof systems we sketched back then are *no* refutation systems. Refutation means doing a proof by *contradiction*. Instead of proving a formula φ, one assumes the opposite $\neg\varphi$ and if, assuming the opposite, one is able to derive absurdity $\bot$ (a contradiction), then the original formula φ must be true (or valid etc). In non-intuitionistic logics, that works fine: instead of trying to establish validity of φ, one tries *non-satisfiability* of φ, as both is equivalent. Formulaically:

$$\models \varphi \qquad \text{iff} \qquad \neg\varphi \models \bot \,. \tag{3.13}$$

Remember, that the *core* of the satisfaction relation for LTL is defined as a relation between one paths and one formulas (see Definition 3.2.2). The satisfaction relation between transition systems and formulas from equation (3.12) is *derived* from that, posing a requirement on *all* paths of M starting its initial state.

Formulas describe sets of paths, namely all paths that make the formula true, but likewise transitition systems describes sets of paths, namely all the paths that start in its initial state. We can therefore view the model checking problem $M \models^? \varphi$ from equation (3.12) as entailment problem.

Let's treat the transition system M as specification of all its paths. We can write $[\![M]\!]$ for the set of all its paths starting at its initial state. While going down that road, we might as well say "a path π satisfies M", i.e. writing $\pi \models M$ for $\pi \in [\![M]\!]$. With this picture in mind, then the model-checking question $M \models^? \varphi$ is the logical entailment: every path "satisfying" M also satisfies φ.

That does not help much in practically addressing the model checking problem, but it helps to see the analogy to refutation methods. In analogy to validity from the equivalence from equation (3.13), the entailment can be refuted based on the following equivalence:

$$\Gamma \models \varphi \qquad \text{iff} \qquad \Gamma, \neg\varphi \models \bot \,. \tag{3.14}$$

The set of formulas $\Gamma, \neg\varphi$ on the left hand side of $\models$ is interpreted as *conjunction* of the formulas of Γ and $\neg\varphi$. An entailment is refuted if there is not interpretation that satisfies all formulas from Γ and $\neg\varphi$ at the same time. In our picture seeing M as well as φ as specifying a set of paths, we could write

$$M \models \varphi \qquad \text{iff} \qquad M \land \neg\varphi \models \bot \,. \tag{3.15}$$

Alternatively we can formulate entailment as subset requirement:

$$[\![M]\!] \subseteq [\![\varphi]\!] \qquad \text{iff} \qquad [\![M]\!] \cap \overline{[\![\varphi]\!]} = \emptyset \,. \tag{3.16}$$

where we use $\overline{A}$ to represent the complement of a set A. So $\overline{[\![\varphi]\!]}$ is represents the negated formula $[\![\neg\varphi]\!]$.

The big picture (2): steps of the construction

We have seen now that model checking can (also) be understood as refutation problem. And that picture is underlying the automata-based model checking approach which is the topic of this chapter. But it's still just a picture. What we need is to operationalize it, to turn in into an algorithm. So how can one solve the problem

$$[\![M]\!] \cap \overline{[\![\varphi]\!]} = \emptyset \;, \tag{3.17}$$

where M is a transition systems representing the system we want to check and φ the specification in LTL?

In the refutation-discussion we said, the M can also seen as a "logical" specification, namely specifying a set of paths. But of course M and φ are specifications of paths of quite different formalisms. M is a transition system and φ a logical formula. So a notation like $\varphi \wedge M$ we allowed ourselves (in equation (3.15)) is a bit dubious (but of course the $\wedge$ is un-dubiously explained by the intersection formulation from equation (3.16)).

To check entailment resp. the refutation formulation from equation (3.16) directly is problematic (in that it does not work..). What we would have to do is for checking the entailment is to calculate two infinite sets of infinite paths and then check that one use *included* in the other. For the refutation formulation, we need do to

1. calculate two infinite sets of infinite paths, $[\![M]\!]$ and $[\![\varphi]\!]$
2. calculate the complement of the latter,
3. calculate the intersection between then,
4. check if the intersection is empty.

Of course, when done directly, so those steps all involve infinite sets of infinite entities and that's not how to do it.

What we need instead is a **finite representation** of those mentioned infinite sets and then doing the steps not on the infinite sets but manipulating their finite representation. Fortunately, we have those finite representations already! Namely the transition system M and the formula φ.

Unfortunatly, though, they are not "on the same level", one is a transition system and the other is an LTL formula. What they have in common is that both describe infinite sets of paths.

To put them on a common level, we have to translate the formula φ to a "transition system". The above steps of the approach include actually one more, that said translation. Actually, one of the steps from above listed 4 can be (and is) omitted. That's step 2), the complementation. One does not need to figure out how to complement a transition system: if one has figured out a way to translate all LTL formulas to a transition system, then one simply translates $\neg\varphi$ to avoid complementation. So the steps, now on the finitite representations of the infinite sets of paths are

1. translate $\neg\varphi$ to $A_{\neg\varphi}$, a finite respresentation with nodes and edges,
2. calculate a representation that corresponds to the intersection, and
3. check if the intersection represents the empty set.

The above sketch of the approach has to be taken with a grain of salt. We said that one step put the $\neg\varphi$ at the same level with M by translating it to a transition system. In principle that's correct, but there is fine-print. The fine print is that the translated $A_{\neg\varphi}$ is not exactly of the same form as the transition system M. We will see that when looking in detail at the construction (but gloss over the details now, when discussing the big picture). Indeed $A_{\neg\varphi}$ is not called transition system but **automaton**. That's why the approach is called **automata-based LTL model checking**.

The difference between the definition of transition systems and the automata is not big. For instance, we consider transition systems in a form where the *nodes* of the transition system carry information or a labelled. For the automata, it's the *edges* which are labelled (as is standard for automata). Also, the automata have *accepting states*, again standard for automata, whereas M, representing a program or system, does not. In the construction later, one has to take care that one representation is transition labelled and the other one is action labelled, but it's a detail indeed. Both formats, edge-labeled and node-labelled, are interchangable, and the construction later will contain a small step, where M is massaged into a edge-labelled representation, to make A and the system representation to be really on the same level, being two automata not only in spirit, but for real. But that's for later, we ignore that in the rest here, we consider transition systems and automata as basically the same formalism.

Talking about automata and sets of sequences may ring a bell: a very well-known concept are **regular languages**. In that context, a set of sequences is called a **language** and sequences (over an alphabet) are called **words**. Languages are typically infinite sets, finite languages are trivial. It's a well–known fact that regular languages can be described finitely in two different and equivalent ways, by **regular expressions** and by **finite-state automata**.

The paralell is pretty close: we are interested in representing infinite sets of words, LTL corresponds conceptually to regular expressions and in both cases there is automaton representation as a different finite reprentation.

There are important differences as well. The most important one is that here we are dealing with infinite sets of **infinite words** (we call them paths, but that's just a name). Furthermore LTL is quite more expressive (already from the fact that it can express properties about infinite words). Indeed that's a crucial difference. It has to do with the fact that LTL can express liveness properties, whereas regular expressions in a way can only express safety properties. Finally, traditional finite-state automata are defined in such a way that they describe or accept only finite words. For LTL we need to adapt that, so that the automata, still finite-state , accepts infinite words. One way of doing that actually does not even change the definition of finite-state automaton at all, one does just change the conditions under which a word is *accepted* by an automaton. Roughly like that: a standard automaton accepts a word when hitting an accepting state. A **infinte-word automaton** accepts a word if it hits accepting states infinitely often. **Büchi-automaton** is a well-known version of an infinite word automaton, and that's the kind of automaton used for LTL model checking. There are other such automata as well, some equivalent to Büchi-automata, some not, but we just cover the Büchi-flavor which works well for LTL.

So, are we done then? Not quite. Having a finite representation of infinite sets of infinite words in the form of automata is great. But the automata have become quite more expressive thanks to the more complex acceeptance condition. One can implement such an automaton straightforwardly, in the same way that one can implement a standard finite-state automaton, actually since only the acceptences condition has changed, the implementation is unchanged except when to stop generating or acceting a word.

Indeed, since acceptnance requires to hit an accepting state infinitely often, such automata cannot be used to actually accept infinite words (in the way an ordinary finite-state automation can be used to accept finite words, maybe in a lexer). Indeed, using an automaton or something else, how can one expect to check properties of infinite words? It's related to an earlier discussion about safety vs. liveness, and about run-time verification. When dealing with infinite words or runs and their properties, the best one can do is spotting in a finite prefix violations of *safety* properties, liveness properties cannot be monitored or checked via run-time verification.

That sounds like bad news for model checking, but model checking is not about checking individual runs or individual infinite words, it's about working with their automata reprentations.

So what actually needs to be solved algorithmically is

translation: find a translation for LTL formulas to an equivalent Büchi automaton

intersection: construct and automata the represents the intersection of two automata

emptyness: check if the language of a given automata is empty

As said, a construction covering complementation can be avoided, we simply translate $\neg\varphi$.

We address the constructions in detail later, but before we do that, let's elaborate on the parallel between finite-word languages and infinite-word languages and their finite representations, looking also the constructions just mentioned.

3.4 Automata and logic

After having shed light on the concepts behind LTL model checking, it's time to get more technical. In this section we cover standard finite-state automata, and afterwards Büchi-automata, a well-known representative of finite-state automata for infinite words.

3.4.1 Finite state automata

Let's start by introducing the well-known concept of finite-state automata

Definition 3.4.1 (Finite-state automaton)**.** A *finite-state automaton* is a quintuple $(Q, q_0, , \Sigma, F, \rightarrow)$, where

- Q is a finite set of states
- $q_0 \in Q$ is a distinguished initial state
- the "alphabet" Σ is a finite set of labels (symbols)
- $F \subseteq Q$ is the (possibly empty) set of final states
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation, connecting states in Q.

What we call *alphabet* here (with a symbol Σ) is sometimes also called *label set* (maybe with symbol L) and sometimes the elements are also called *actions*. The terminology "alphabet" comes from seeing automata to define words and languages, the word "action" more when seeing the automaton as a system model that represents an (abstraction of) a program.

The notion of *finite state automata* is probably known from elsewhere. It's used directly or in variations in many different contexts. Even in its more basic forms, the concept is known under different names or abbreviations (FSA and NFA, finite automaton, finite-state machine). Minor and irrelevant variations concern details like whether one has one initial state or allows a set of initial states. Sometimes the name is also used "generically", for example, automata which carry more information than just labels on the transitions. For instance, information which is interpreted as input and output on the states and/or the transitions (also known Moore or Mealy machines). Such and similar variations are no longer *insignificant* deviations like the question whether one has one initial state or potentially a set. Nonetheless those variations are sometimes also referred to as FSAs, even if technically, they deviate in some more or less significant aspect from the vanilla definition given here. They are called finite-state machines or finite-state automata simply because they are state-based formalisms with a finite amount of states and some form of transition relation in between (and potentially labelled or interpreted in some particular way or with additional structuring principles).

Other names for related concepts is that of a (finite-state) *transition system*. And even Kripke structures or Kripke models can be seen as a variation of the theme, though in a more logical or philosphical context, the *edges* betwen the worlds may not be viewed as *transitions* or operational steps in a evolving system. In Baier and Katoen [2], they call Kripke structures *transition systems* (actually without even mentioning Kripke structures).

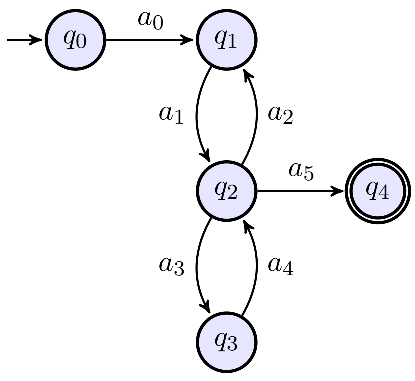
We are not obsessed with terminology. But as preview for later: In the central construction about model checking LTL, the system on the one hand will represented as a (finite) transition system where the *states* are labelled and the LTL formula on the other hand will be represented by an *automaton* whose *transitions* are labelled. The automaton will be called *Büchi*-automaton. The definition corresponds to the one just given in Definition 3.4.1. What makes it "Büchi" is not the form or data structure of the automaton itself, it the *acceptance* condition, i.e., the intepretation of the set of accepting states.

Let's illustrate the definition on a small example:

Example 3.4.2. The automaton is given by the 6-letter alphabet (or label set) $\Sigma = \{a_0, a_1, \ldots, a_5\}$, by the 5 states q_0, q_2, ..., q_4, with initial state and one final state and the transitions as given in the figure. “Technically”, one could enumerate the transitions by listing them as triples or labelled edges one by one, like

$$\rightarrow = \{(q_0, a_0, q_1), \ldots, (q_2, a_5, q_4)\} \subseteq Q \times \Sigma \times Q \ ,$$

but it does not make it more “formal” nor does it add clarity.



Renaming the letters of the alphabet, the above automaton may be interpreted as a *process scheduler*:

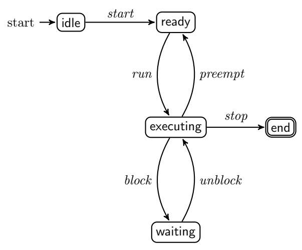


Figure 3.3: FSA scheduler

Determinism vs. non-determinism

Determinism in a system geneally means that, in a given situation or state, the next state is determined. Functions are deterministic: given an input, the function output is determined as well. In that sense, relations can be seen non-deterministic “functions”. For automata and related formalisms, determinism means, being in a state and given a letter from the alphabet, there are at most one successor states. The automata from Definition 3.4.1 can be non-deterministic, the definition is based on a transition *relation*.

Definition 3.4.3 (Determinism)**.** A finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is *deterministic* iff

$$q_0 \xrightarrow{a} q_1 \ \wedge q_0 \xrightarrow{a} q_2 \implies q_1 = q_2$$

for all q_0, q_1, and q_2 from Q and all a in Σ.

The definition of *deterministic* automaton is not 100% equivalent with requiring that there is a transition *function* that, for each state and for each symbol of the alphabet, yields *the* unique successor state. Our definition basically requires that there is *at most* one successor state (by stipulating that, if there are two successor states, they are identical). That means, the successor state, if it exists, is defined by a *partial* transition function.

Sometimes, the terminology of *deterministic* finite-state automaton *also* includes the requirement of *totality*, i.e., the transition relation is a total relation, wich makes it a *total function*.

I.e., the destination state of a transition is uniquely determined by the source state *and* the transition label. An automaton is called **non-deterministic** if it does not have this property. We prefer to separate the issue of deterministic reaction to an input in a given states ("no two different outcomes") from the issue of totality.

It should also be noted that the difference between deterministic (partial) automata and deterministic total automamata is not really of huge importance. One can easily consider a partial automaton as total by adding an extra "error" state. Absent successor states in the partial deterministic setting are then represented by a transition to that particular extra state. The reason why some presentations consider a deterministic automaton to be, at the same time, also "total" or complete is, that, as mentioned, it's not a relevant big difference anyway. Secondly, a complete and deterministic automaton is the more useful representation, either practically or also for other constructions, like *minimizing* a deterministic automaton. But anyway, it's mostly a matter of terminology and perspective: every (non-total) deterministic automaton can immediately alternatively be interpreted as total deterministic function. It's the same in that any partial function from A to B, sometimes written $A \hookrightarrow B$ can be viewed as total function $A \to B_\perp$, where $B_\perp$ represents the set B extended by an extra error element $\perp$.

The automaton from the earlier example, the process scheduler, is *deterministic*.

Runs, acceptance, and languagues of automata

Definition 3.4.4 (Run)**.** A *run* of a finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \to)$ is a (possibly infinite) sequence

$$\sigma = q_0 \overset{a_0}{\to} q_1 \overset{a_1}{\to} \ldots$$

The notation $q \overset{a}{\to} q'$ is meant as $(q, a, q') \in \to$. Each run corresponds to a **state sequence** over Q and a **word** over the alphabet Σ. Of course also the state sequence can be called a word, interpreting the set of states as alphabet.

As mentioned a few times: the terminology is not "standardized" throughout. Here, on the slides, we defined a run of a finite-state automaton as a finite or infinite sequence of *transitions*. Words which more or less means the same in various contexts (an perhaps based on transition systems or similar, not automata) include *execution*, *path*, etc. All of them are modulo details similar in that they are *linear* sequences and refer to the "execution" of an automaton (or machine, or program). The definition we have given contains "full information" insofar that it is a sequence of transitions. It corresponds to the choice of words in [20] (the "Spin-book"). The book Baier and Katoen [2], for example, uses the word run (of a given Büchi-automaton) for an infinite state sequence, starting in a/the initial state of the automaton.

For me, the definition of run as given here is a more "plausible" interpretation of the word. A run or execution (for me) should fix all details that allows to reconstruct or replay what concretely happened. Considering state sequences as run would leave out which *labels* are responsible for that sequence. Not that it perfectly possible that $q \xrightarrow{a} q'$ and $q \xrightarrow{b} q'$ (for two different labels a and b) even if the automaton is deterministic.

In a deterministic automaton, of course, a "word-run" determines a "state-run".

As a not so relevant side remark: we stressed that modulo minor variations, a commonality on different notions of *runs*, *executions*, (and histories, logs, paths, traces . . .) is that they are **linear**, i.e., they are sequences of "things" or "events" that occur when running a program, automaton, . . . When later thinking about *branching time logics* (like CTL etc), the behavior of a program is not seen as a set of linear behaviors but rather as a tree. In that picture, one execution correspond to one tree-path starting from the root, so again, one execution is a linear entity.

Side remark 3.4.5 (Other views on the concept of execution)**.** There exist, however, approaches where **one execution** is not seen as a linear sequence, but as something more complex. Typical would be a *partial* order (a sequence corresponds to a *total* order). There would be different reasons for that. They mainly have to do with modelling concurrent and distributed systems where the total order of things might not be observable. Writing down in an execution that one thing occurs before the other would, in such setting, just impose an artificial ordering, just for the sake of having a linear run, which otherwise is not based on "reality". In that kind setting, one speaks also of partial order semantics or "true concurrency" models (two events not ordered are considered "truely concurrent"). Also in connection with weak memory models, such relaxations are common. Considering partial orders (when it fits) is also a optimization technique: by avoding to explore all interleavings of all linearizations of a partial order, one can make model checking technique more efficient.

Those considerations will not play a role in the lecture: runs etc. are *linear* for us (total orderings). □

Example 3.4.6 (Run). Consider the automaton from Figure 3.3. State sequences arising from possible runs look like

$$\mathsf{idle\ ready\ (execute\ waiting)}^* \ .$$

Words in Σ that correspond to the shown state sequences are of the form

$$start\ run(block\ unblock)^* \ .$$

There are of course others as well, for instance runs involving the ready-state resp. *preempt*-steps. There are also infinite runs. In general, a single state sequence may A single state may correspond to more than one word, but in the example, the transition-label sequence determines the state sequence: the automaton is deterministic. □

Definition 3.4.7 (Acceptance)**.** An *accepting* run of a finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is a finite run $\sigma = q_0 \overset{a_0}{\rightarrow} q_1 \overset{a_1}{\rightarrow} \ldots \overset{a_{n-1}}{\rightarrow} q_n$, with $q_n \in F$.

In the scheduler example from before: a *state sequence* corresponding to an acceping run is

idle ready executing waiting executing end .

The corresponding *word* of labels is

start run block unblock stop .

An accepting run (as defined here) determines both the state-sequence as well as the label-sequence. In general, the state-sequence in isolation does not determine the label-sequence, not even for deterministic automata. But in the case of the scheduler example, it does.

The definition of acceptance is "traditional" as it is based on 1) the existance of an accepting sequence of steps which is 2) finite. The definition speaks of *accepting runs.* With that definition in the background, it's also obvious what it means that an automaton accepts a *word* over Σ or what it means to accept a state sequence. Later, when we come to LTL model checking and Büchi-automata, the second assumption, that of finite-ness will be dropped, resp. we consider *only* infinite sequences. The other ingredient, the $\exists$-flavor (there exists an accepting run) will remain.

Angelic vs. daemonic choice The $\exists$ in the definition of acceptance is related to a point of discussion that came up in the lecture earlier (in a slightly different context), namely about the nature of "or". I think it was in connection with regular expressions. Anyway, in a logical context (like in regular expressions or in LTL), the interpretation is more or less clear. If one takes the logic as describing behavior (the set of accepted words, the set of paths etc.), then disjunction corresponds to *union* of models.

When we come to "disjunction" or choice when describing an automaton or accepting machine, then one has to think more carefully. The question of "choice" pops up only for *non-deterministic* automata, i.e., in a situation where $q_0 \overset{a}{\rightarrow} q_1$ and $q_0 \overset{a}{\rightarrow} q_2$ (where $q_1 \neq q_2$). Such situations are connected to *disjunctions*, obviously. The above situation would occur where q_0 is supposed to accept a language described by $a\varphi_1 \vee a\varphi_2$. In the formula, φ_1 describes the language accepted by q_1 and φ_2 the one for q_2. The disjunction $\vee$ is an operator from LTL; if considering regular expressions instead, the notations "|" or "+" are more commonly used, but they represent disjunction nonetheless.

Declaratively, disjunction may be clear, but when thinking *operationally,* the automaton in state q_0 when encountering a, must make a "choice", going to q_1 or to q_2, and continue accepting. The definition of acceptance is based on the *existance* of an accepting run. Therefore, the accepting automaton must make the choice in such a way that leads to an accepting state (for words that turn out to be accepted). Such kind of making choices are called *angelic*, the choice supports acceptance in a best possible way.

Of course, this is also "prophetic" in that choosing correctly requires foresight (but angels can do that… Daemons can do that, as well, it's only that angels make decisions in favor of acceptance whereas daemons use their forsight to sabotage it). Of course, concretely, a machine would either have to do *backtracking* in case a decision turns out to be wrong. Alternatively one could turn the non-deterministic automaton to a deterministic one, where there are no choices to made (angelic or otherwise). It corresponds in a way a precomputation of all possible outcomes and exploring them at run-time all at the same time (in which case one does not need to do backtracking). A word of warning though: Büchi automata may not be made deterministic. Furthermore, it's not clear what to make out of *backtracking* when facing *infinite* runs.

The angelic choice this proceeds successfully if *there exists* a successor state that allows succesful further progress. There is also the *dual interpretation* of a choice situation which is known as *demonic*, which corresponds to a ∀-quantification. The duality between those two forms of non-determism shows up in connection with *branching time* logic (not so much in LTL). Also the duality is visible in "open systems", i.e., where one distinguishes the systems from its environment. For instance for security, the envionment is often called *attacker* or *opponent*. This distinction is at the core also of game-theoretic accounts, where one distinguishes between "player" (the part of the system under control) and the "opponent" (= the attacker), the one that is not under control (and which is assumed to do bad things like attack the system or prevent the player from winning by winning himself). In that context, the system can try do a good choice, angelically (∃) picking a next step or move, such that the outcome is favorable, no matter what the attacker does, i.e., no matter how bad the demonic choice of the opponent is (∀).

After this digression, back to the main course of the section. Let's just define what it means that an automaton accepts a word. That also defines the *language* of an automaton as the set of all accepted words.

Definition 3.4.8 (Language)**.** The *language* $[\![\mathcal{A}]\!]$ (sometimes also written $\mathcal{L}(\mathcal{A})$ of automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is the set of words over Σ that correspond to the set of all the accepting runs of $\mathcal{A}$.

In general, there are infinitely many words in such languages. Languages of finite-state automata and with this acceptance condition correspond to regular languages, languages expressable via regular expressions.

For the given scheduler automaton from before, one can capture its language of finite words by (for instance) the following regular expression

$$start\ run\ ((preempt\ run)^* \mid (block\ unblock)^*)\ stop\ .$$

We use | for "or", we could also have used +.

In the context of language theory, *words* are finite sequences of letters from an alphabet Σ, i.e., a word is an element from Σ^*, and languagues are sets of words, i.e., subsets of Σ^*. As stressed, for LTL and related formalisms, we are concerned with *infinite words* and languages over infinite words.

Reasoning about runs

We are of course mainly interested in LTL as temporal logic in this chapter. But as a warm-up, we can use regular expressions to specify temporal properties.

Let's have a look a the following temporal property

> If first p becomes true and afterwards q becomes true, then afterwards, r can no longer become true

In the context, the desired property is also called a *correctness claim.* As we know from the big-picture discussion about *refutation*, the model checking procedure for LTL works with the *negation* of the specification. In our example we can formulate that as

> It's an **error** if in a run, one sees first p, then q, and then r.

That property can be represented by the automaton from Figure 3.4.

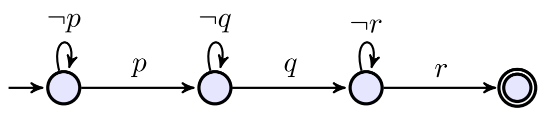


Figure 3.4: Automaton for the negated specification

The accepting states captures that the error condition has been reached. So reaching the accepting state means one has detected about a **violation** of the correctness property.

The example illustrates one core ingredient to the automata-based approach to model checking. One is given a property one wants to verify, like the informally given one from above. In order to do so, one operates with its *negation.* In the example, that negation can be straightforwardly represented as *standard* acceptance in an FSA. Being represented by conventional automata acceptance, the detected errors are witnessed by *finite words* corresponding to finite executions of a system.

Generally, one cannot expect to find a violation of the specification in a finite sequence. A property (like the one above) whose **violation** can be detected by a *finite* path is called a **safety property**. Safety properties form an important class of properties. Note: safety properties are *not* those that can be *verified* via a finite trace, the definition refers to the negation or violation of the property: a safety property can be *refuted* by the existance of a finite run.

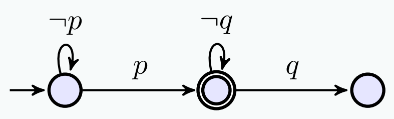
That fits to the standard informal explanation of the concept, stipulating: "that never something bad happens" (because if some bad thing happens, it means that one can detect it in a finite amount of time). The slogan is attributed to Lamport [22]. That "bad" in the sentence refers to the *negation* of the original property one wishes to establised (which is seen thus as "good"). Note one more time: the *original* desired property is the *safety property*, not its negation.

Still another angle to seeing it is: a safety property on paths is a property has the following (meta-)property: If the safety property holds *for all finite behavior, then it holds for all* *behavior* (all behavior includes *inifinite* behavior). For the mathematically inclined: this is a formulation connected to a *limit* construction or closure or a *continuity* constraint, when worked out in more detail (like: infinite traces are the limit of the finite ones etc).

How to use automata to reason about infinite run?

Let's also have a look at a livenes property, say "if p then eventually q.", respectively its negation

> It's an **error** if one sees p and afterwards never q (i.e., forever $\neg q$).



A violation of that is possible only in an **infinite** run. Consequently it cannot be expressed by the *conventional* notion of acceptance.

A moment's thought should get the "silly" argument out of the way that says: "oh, if checking the negation via an automaton does not work easily in a conventional manner, why not use the original, non-negated property. One can formulate that without referring to infinite runs and with standard acceptance.".

Ok, that's indeed silly in the bigger picture of things (why?). What we need (in the above example) to capture the negation of the formula is to express that, after p, there is *forever* $\neg q$, which means for the sketched automaton, that the loop is taken forever, resp. that the automaton stays infinitely long in the middle state (which is marked as "accepting"). What we need, to be able to accepting infinite words is a *reinterpretation* of the notion of acceptance. To be accepting is not just a "one-shot" thing, namely reaching some accepting state. It needs to be generalized to involve a notion of visting states *infinitely* often.

In the above example, acceptance could be to "stay forever in that accepting state in the middle". That indeed would capture the desired negated property. The definition of "infinite acceptance" is a bit more general than that ("staying forever in an accepting state"), it will be based on "visiting an accepting state infinitely often, but it's ok to leave it in between". That will lead to the original notion of **Büchi acceptance**, which is one flavor of formalizing "infinite acceptance" and thereby capturing infinite word languages.

There are alternatives to that particular definition of acceptance. In the lecture we will encounter a slight variation called *generalized Büchi acceptance*. It's a minor variation, which does not change the power of the mechanism, i.e., generalized or non-generalized Büchi acceptance does not really matter. However, the GBAs are more convenient when translating LTL to a Büchi-automaton format. It may be (very roughly) compared with regular languages and standard FSAs. For translating regular expressions to FSAs, one uses a variation of FSAs with so-called ϵ-transitions (silent transitions), simply because the construction is more straightforward (compositional). Generalizing Büchi automata to GBAs does not involve ϵ-transitions but the spirit is the same: use a slight variation of the automaton format for which the translation works more straightforwardly.

3.4.2 Büchi automata

As mention, Büchi-automata are defined as finite state automata from Definition 3.4.1. What makes them "Büchi" is the different acceptance condition, that can handle infinite words or runs. Infinite runs are often often called *ω-runs* ("omega runs"), and the corresponding acceptance conditio consequently ω-acceptance. The are different versions of the idea (Büchi, Muller, Rabin, Streett, parity etc.,), but here we present **Büchi acceptance** [8] [7].

Definition 3.4.9 (Büchi acceptance)**.** An *accepting ω-run* of the finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is an infinite run σ such that some $q_i \in F$ occurs infinitely often in σ.

Automata with this acceptance condition are called **Büchi automata**.

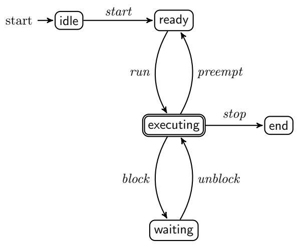


Figure 3.5: Scheduler (2)

Example 3.4.10 (Infinite runs)*.* Consider the automaton from Figure 3.5. It almost the same than the one from Figure 3.3 earlier except that the accepting state has been changed.

One accepted infinite state sequence is the following:

$$\mathsf{idle}\ (\mathsf{ready\ executing})^{\omega}$$

A corresponding ω-word is:

$$start\ (run\ preempt)^{\omega}$$

□

The automaton is meant to illustrate the notion of Büchi-acceptance; it's not directly meant as some specific logical property (or a negation thereof). Nor do we typically think that transition systems that present the "program" we like to model check work as language acceptors and thus have specific accepting states they have to visit infinitely often. Program run, but the Büchi-automata that results from the translation of an LTL formula do have accepting states meant to check the (negation of the) property of interest.

The symbol ω often stands for "infinity" (here and elsewhere). Actually ω in general stands specific infinity as one can have different forms and levels of infinities. Those mathematical fine-points may not matter much for us. But it's the "smallest infinity larger than all the natural numbers", which makes it an *ordinal number* in math-speak and being defined as the "smallest" number larger than $\mathbb{N}$ makes this a *limit* or *fixpoint* definition. It's connected to the earlier, perhaps cryptic, side remark about safety and liveness, where it's important that infinite traces are the *limit* of the finite ones).

For instance, $(ab)^*$ stands for finite alternating sequences of a's and b's, including the empty word ϵ, starting with an a and ending in a b. The notation $(ab)^\omega$ stands for *one* infinite word of alternating a's and b's, starting with an a (and not ending at all, of course). Given an alphabet Σ, Σ^ω represents all infinite words over Σ. As a side remark: for non-trivial Σ (i.e., with more than 2 letters), the set Σ^ω is no longer enumerable (it's a consequence of the simple fact that its cardinality is larger then the cardinality of the natural numbers).

Sometimes, one finds the notation Σ^∞ (or $(ab)^\infty \ldots$) to describe infinite *and* finite words. Remember in that context, that the semantics of LTL formulas is defined over *infinite* sequences (paths), only.

As mentioned shortly, there is also a variation of Büchi-acceptance, called *generalized* Büchi-acceptance. It's a minor variation of the original definition. Both flavors of acceptance conditions are of equivalent expressiveness. When translating LTL-formulas to an infinite-word automaton, the generalized Büchi-automaton format is more convenient, so we will use that one. If one prefers plain Büchi-automata, one could, in a second stage, translate the generalized version into a non-generalized one. But we don't look into that.

For a σ for of a Büchi-automaton, let's write $\mathit{inf}(\sigma)$ for the set of states that occur infinitely often in σ. With this, we can define the generalized Büchi automata and their acceptance condition as follows:

Definition 3.4.11 (Generalized Büchi automaton)**.** A *generalized Büchi automaton* is an automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$, where $F \subseteq 2^Q$.
Let $F = \{f_1, \ldots, f_n\}$ and $f_i \subseteq Q$. A run σ of $\mathcal{A}$ is *accepting* if

$$\text{for each } f_i \in F,\ \ \mathit{inf}(\sigma) \cap f_i \neq \emptyset.$$

Note that not only the acceptance has changed, but also the *format* of the format of the automaton. Büchi automata, as standard finite state automata, the acceptance is based on a set F of states. Here, F is not a set of states, F is a set of sets of states. So the automata has not multiple accepting states but multiple accepting sets of states.

As mentioned earlier, the motivation to introduce this (minor) variation of what it means for an automaton to accept infinite words comes from the fact that it is just easier to translate LTL into this format.

Büchi automata (generalized or not) is just one example of automata for infinite words. Those are generally known as ω-automata. There are other acceptance conditions (Rabin, Streett, Muller, parity . . .), which we will not cover in the lecture. When allowing non-determinism, they are all equally expressive. It's well-known that for finite-word automata, non-determinism does not add power, resp. that determinism is not a restriction for FSAs. The issue of non-determinism vs. determinism gets more tricky for ω-words. Especially for Büchi-automata: deterministic BAs are strictly less expressive than their non-deterministic variant! For other kinds of automata (Muller, Rabin, Street, parity), their deterministic and non-deterministic versions are equally expressive. In some way, Büchi-automata are thereby not really well-behaved, the other automata are nicer in that way. The class of languages accepted by those automomata is also known as ω-regular languages.

3.4.3 Stuttering

Next we address an issue not so much from automata theory, but motivated by the use we make of the concepts modelling systems and model checking them, In the big-picture dicussion earlier we mentioned that the approach works with two rather similar representations with nodes and edges and labels, namely automata and transition systems. There are pretty close, but they also serve different purposes. The transition systems' role is to model the execution of programs or system, the automata on the other hand represent an LTL formula, and the latter has accepting states, whereas "acceptance" is a concept alien to programs. Instead program may *terminate*.

So that's another mismatch, LTL works on *infinite runs*, and Büchi acceptance, genealized or not, accepts by definition *only* infinite runs.

But that's an easy nut to crack. We cannot allow the system to just terminate, because we can only logically (resp. by Büchi-automata) handle infinite run. So if our transition system (massaged into another Büchi-automaton) terminates, we simply let it artificially continue infinitely by doing nothing. That is known as **stuttering**. This allows to treat finite and infinite acceptance uniformely by Büchi-acceptance condition. It avoids coming up with a more complex alternative acceptance condition that allows to accept finite and infinite words.

Let ε be a predefined nil symbol and the alphabet/label set extended to $\Sigma + \{\varepsilon\}$. So a finite run terminating by reaching some state without successor becomes, by stuttering, an inifite ones, where, at some point, infinitely often ε is done. Stuttering (here) is allowed only at the end, i.e. a run must end in an end-state

Definition 3.4.12 (Stutter extension)**.** The *stutter extension* of a finite run σ with last state s_n, is the ω-run

$$\sigma\ (s_n, \varepsilon, s_n)^{\omega}\ . \tag{3.18}$$

Let's revisit the schedular example again

Example 3.4.13 (Stuttering). The "process scheduler" example from Figure 3.3 uses the accepting state end now as natural end state with an ε-loop which is also accepting (see Figure 3.6). Examples of accepting state sequences resp. accepting words corresponding to an accepting ω-run include the following:

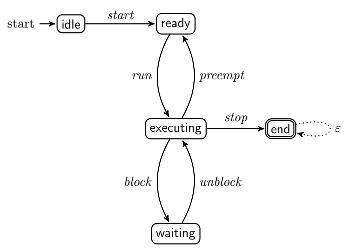


Figure 3.6: Scheduler with stuttering

$$\mathsf{idle\ ready\ executing\ waiting\ executing\ end}^{\omega}$$

and

$$\textit{start run block unblock stop}^{\omega}$$

□

So far, we have introduced the stutter extension of a (fintite) run. But runs will be ultimately runs “through a system” or through an automaton. Of course there could be a state in the automaton when it’s “stuck”. Note that we use automata or transition systems to represent the behavior of the system we model as well as properties we like to check. The stutter-extension on runs is concerned with the “model automaton” representing the system. To me able to judge whether a run generated by the system satisfies an LTL property, it needs to be an *infinite run*, because that’s how $\models$ for LTL properties is defined. The fact that in the construction of the algorithm, also the LTL formula (resp. its negation) will be translated to an automaton is not so relevant for the stutter discussion here.

3.4.4 Something on logic and automata

In this section we bridge a mismatch between transition systems and Büchi automata. The mismatch, mentioned earlier, is that automata are edge labelled and transition systems, at least the ones we introduced, carry information in the worlds or states. The mismatch is not large, so bridging it will be easy

Another issue we look at examples how different Büchi automata can represent LTL properties. Also that is done informally, we don’t show the actual construction. That is for later.

From Kripke structures to Büchi automata

We have encountered different “transition-system formalisms”. One under the name transition systems (or Kripke models or Kripke structures), the other one automata.

The Kripke structures or transition systems are there to model “the system” whereas the automata serve to specify temporal properties of the system.

On the one hand, those formalisms are basically the same. On the other hand, there is a slight mismatch: the automaton is seen as "transition- or edge-labelled", the transition system is "state- or world-labelled". The mentioned fact that the transition systems used in [2] are additionally "transition-labelled" is irrelevant for the discussion here, the labelling there serves mostly to be able to capture synchronization (as mechanism for programming or describing concurrent systems) in the parallel composition of transition systems.

As also mentioned earlier, there is additionally slight ambiguity wrt. terminology. For instance, we speak of states of an automaton or a state (= world) in a transition system or Kripke structure. On the other hand, we also encountered the state-terminology as a mapping from (for example propositional) variables to (boolean) values. Similar ambiguity is there for the notion of *paths*. It should be clear from the context what is what. Also the notions are not contradictory. We will see that for the notion of "state" later, as well.

Now, there may be different ways to deal with the slight mismatch of state-labelled transition system and edge-labelled automata on the other. The way we are following here is as follows. The starting point, even before we come to the question of Büchi-automata, describes the behavior of Kripke structures in terms of statifaction per *state* or "world", not in terms of *edges*. For instance $\Box\Diamond p$ is true for a path which contains infinitely many occurrence of p being true, resp. for a Kripke structure whose every run corresponds to that condition. So, for all infinite behavior of the structure, p has to hold in infinitely many *states* (not transitions); propositions in Kripke-structures hold in states, after all (or Kripke structure are state labelled).

Remember also that we want to to check that the "language" of the system M is a subset of the language described by a LTL-specification φ, like $M \models \varphi$ corresponds to $[\![M]\!] \subseteq [\![\varphi]\!]$. To do that, we'd like to translate LTL-formulas (more specifically $\neg\varphi$) into automata, but those are transition-labelled (as is standard for automata in general). So, $[\![M]\!] $ is a language of infinite words corresponding to sequences of "states" and the state-attached information. On the other hand, $[\![\varphi]\!]$ is a language containing words referring to edge-labels of and automaton.

So there is a slight mismatch. It's not a real problem, one could easily make a tailor-made construction that connects the state-labelled transition systems with the edge-labelled automaton and then define what it means that the combination is does an accepting run. And actually, in effect, that's what we are doing in principle. Nonetheless, it's maybe more pleasing to connect two "equal" formalisms. To do that, we don't go the direct way as sketched. We simply say how to interpret the state-labelled transition system as edge-labelled automaton, resp. we show how in a first step, the transition system can be transformed into an equivalent automaton (which is straighforward). Thus we have two automata, and then we can define the intersection (or product) of two entities of the same kind.

One might also do the "opposite", like translating the automaton into a Kripke-structure, if one wants both logical description and system desciption on equal footing. However, the route we follow is the standard one. It's a minor point anyway and on some level, the details don't matter. On some other level, they do. In particular, if one concretely translates or represents the formula and the system in a model checking tool, one has to be clear about what is what, and which representation is actually done.

Translating transition systems to Büchi automata

We call the "states" here now W for worlds, to distinguish it from the states of the automaton. We write $\rightarrow_M$ the accessibility relation in M, to distinguish it from the labelled transitions in the automaton.

Definition 3.4.14 (Translating a Kripke structure into an Büchi-automaton)**.** Given $M = (W, R, W_0, V)$. An automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ can be obtained from M as follows. The alphabet for the transition labels is given as $\Sigma = 2^P$. For the states, set

$$Q = W + \{i\} \qquad q_0 = i \qquad F = W + \{i\}$$

For the *transitions*, we set $s \xrightarrow{a} s'$ iff $s \rightarrow_M s'$ and $a = V(s')$ $s, s' \in W$ and $i \xrightarrow{a} s \in T$ iff $s \in W_0$ and $a = V(s)$.

Note: the translation turns *all* worlds of the transition system into accepting states of the Büchi automaton. Basically, the accepting conditions are not so "interesting" and making all states accepting means: I am interested in *all* behavior as long as it's inifinite. The KS (and thus the corresponding BA) is not there to "accept" or reject words. It's there to produce infinite runs without stopping (and in case of an end-state it means, continue infinitely anyway by *stuttering*).

Here, the Kripke structure has initial states or initial worlds (W_0), something that we did not have when introducing the concept in the modal-logic section. At that point back then we were more interested in questions of "validity" and "what kind of Kripke-frames is captured by what kind of axioms", things there are important in dealing with validity etc. In that context, one has no big need in particular "initial states" (since being valid means for all states/worlds anyway). But in the context of model checking and describing systems, it's, of course, important.

Note also, that the valuations $V : W \rightarrow (P \rightarrow \mathbb{B})$ attach to each world or "state" a mapping, that assigns to each atomic proposition from P a truth value from $\mathbb{B}$. That can be equivalently seen as attaching to each world a *set* of atomic propositions, i.e., it can be seen as of type $W \rightarrow 2^P$. Perhaps confusingling the assignment of (here Boolean values) to atomic propositions, i.e., functions of type $P \rightarrow \mathbb{B}$, are sometimes also called *state* (more generally: a state is an association of variables to their (current) value, i.e., a state is a current content or snapshot of the memory). The views are not incompatible (think of the program counter as a variable . . .)

Example 3.4.15 (From Kripke structure to Büchi automata)*.* The Kripke structure from Figure 3.7a is translated to the Büchi-automaton from Figure 3.7b. BTW: the (only) infinite run of the Kripke structure satisfies (for instance) $\Box q$ and $\Box\Diamond p$).

□

3.4.5 Describing temporal properties by LTL and Büchi automata

We have two formalisms to describe sets of infinite paths: LTL and Büchi-automata. Even three, if we count transition systems in. As mentioned in the big-picture discussion,

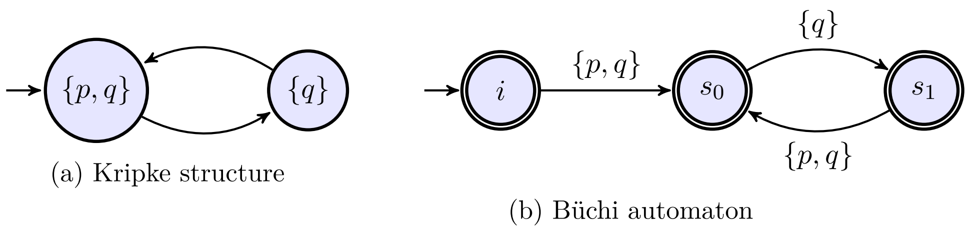


Figure 3.7: Translation

the situation may be compared to standard word languages, with words of finite length, which can be described by regular expressions and finite state automata. Finite state automata and regular expressions are equivalent in that they can describe the same word languages.

In contrast, LTL and Büchi-automata are *not* two versions of the same thing, only half so: for every LTL formula φ, there exists a Büchi automaton that accepts precisely those runs that satisfy φ. The reverse direction does not hold, i.e., Büchi-automata are more expressive.

Here we illustrate how one can use Büchi automata to capture some LTL properties, without showing the contruction; that comes later. Here, we also mention some other formalisms for infinite languages, some more expressive than LTL, some less, some incomparable.

Example 3.4.16 (Stabilization: "eventually always p")*.* Figure 3.8 shows an Büchi-automaton checking the LTL property $\Diamond\Box p$. □

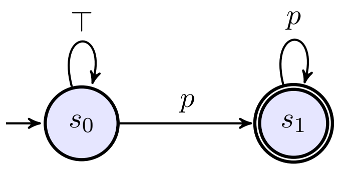


Figure 3.8: Büchi-automaton for stabilization

(Lack of?) expressiveness of LTL

As mentioned, the analogy of Büuchi-automata and LTL on the one and and FSa and regular expressions on the others is not 100%. In the case of finite-word languages, the two mechanisms of automata and of regular expressions are equivalent. Here, LTL is strictly **weaker** than BA.

The Büchi-automata of the following example captures a property that cannot be expressed LTL.

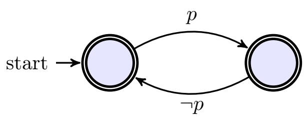
Example 3.4.17*.* The example is about capturing the following temporal property:

> p is always false after an *odd* number of steps

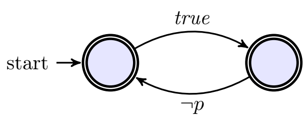
Let's start by trying to capture that with LTL:

$$p \land \Box(p \to \bigcirc\neg p) \land \Box(\neg p \to \bigcirc p) \tag{3.19}$$

The formula from equation (3.19) indeed assures that p is false after odd steps, as required, but it's not exactly what was informally given! The LTL specifies, with p holding initially, a strict oscillation between places where p holds and those where p does not hold. But that's more than what the informal sentence required, which did not insist that on the even places, p holds. At any rate, this oscillating behavion can be represented by the Büchi-automaton from Figure 3.9a.



(a) Strict oscillation



(b) $\neg p$ after odd steps

Figure 3.9: Büchi automata

The second Büchi-automaton from Figure 3.9b does not impose restriction on the even-positions and thus corresponds to the original informal formulation. LTL cannot capture that automaton, though we don't prove it.

The property can be captured by the following temporal formula:

$$\exists t.\ t \land \Box(t \to \bigcirc\neg t) \land \Box(\neg t \to \bigcirc t) \land \Box(\neg t \to p) \tag{3.20}$$

The formula, however, is not LTL. It's a formula from what is called ∃LTL. □

What do do about the mismatch? One can live with the mismatch, of course. That's what we do in the lecture: we translate LTL Büchi-automata for the purpose of model checking, and that's all we really want.

But both formalisms seems kind of natural, so one can try to repair the mismatch (or at least analyze the reason of the mismatch).

Concerning Büchi-automata, there exist also so called **ω-regular** expressions and ω-regular languages, which are a generalization of regular languages and whose expressiveness matches that of non-deterministic Büchi-automata. They look like regular languages, except one can write r^ω (not just r^*). There exist a "crippled" form of "infinite regular expressions" that is an exact match for LTL.

To "repair" the mismatch between ω-regular languages on the one hand and LTL on the other, one could two things. One can ask, what needs to be taken away from BAs resp. ω-regular expressions to make them fit to LTL, resp. ask whether there is an automaton model that exactly fits LTL? Alternatively one can ask: what needs to be added to LTL to make it as expressive as BAs? Example 3.4.17, in particular the formula from equation (3.20) hints at a solution for the second approach: It's a result from [14] which states that allowing prefix *existential quantification* over one propositional variable (the t in the example) is enough to repair the mismatch. That version if LTL is sometimes called “existential LTL” or ∃LTL.[1]

The Spin model checker resp. its input language Promela offers another mechanism that gives the same expressivity to LTL as ω-regular languages. This mechanism is known as **never claims**.

Spin has a translator `ltl2ba` and one can find translators from LTL to BA on the net as well, for instance under `http://www.lsv.fr/~gastin/ltl2ba/`.

Figure 3.10 compares different well-known temporal logics wrt. their expressivity. Concerning LTL, let’s remark that when removing the next-operator $\neg$, the logic becomes strictly less expressive. That fragment is of interest when specifying and verifying properties of *asynchronous* systems. By that we mean systems concsisting of a processes or threads running concurrently, where the next step of the system seen globally is in general done by one of the processes, chosen randomly, and the other processes don’t do anything. Practically, the choices is done perhaps by a scheduler, that picks among the enabled processes on to execute. Typically the scheduler chooses not really randomly, not does it roll the dice freshly after every indidual step of a process. Normally some pricorities or stategy is involved (round-robin, maybe), and every process gets time-slots to do quite a number of steps before pre-empted again. But for modelling, one may chose to abstracting away from the scheduler, and treating it as doing random scheduling. That has not just the advantage of simplifying the model and this the model-checking challenge. Additionally, if one has model-checked a concurrent program as ok under random-scheduling, it works for any specific strategy a real scheduler may apply.

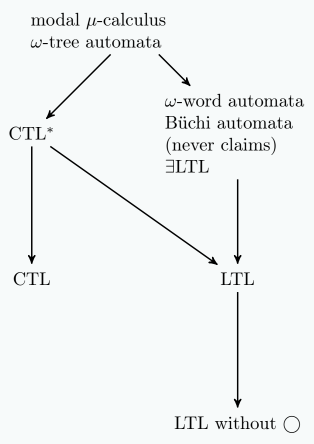


Figure 3.10: Expressivity of a few temporal logics

Now, in such a asynchronpus picture, and if $\bigcirc$ speaks about transitions of the global system and if on specifies local properties of one process, then $\bigcirc\varphi$ make little sense. For instance, if one process is in a state doing `x:=1`, it still makes no sense to say locally

[1]We will not discuss that variant. Unfortunately there is a *different* formalism with which called existential LTL (or ∃LTL. That’s a version where the LTL formula is meant not to hold for all paths, like we do, but is meant to specify that there *exists* a path that satisfies the temporal property. This form of ∃LTL we will encounter later in the context of bounded model checking.

“$\bigcirc(x = 1)$”. In the face of non-deterministic, asynchronous scheduling, it’s not clear what the next global transition will be, and it makes not much to specify what should hold. One can still do it, but it would require global knowledge, and the argumnt here is specifying locally, which is less messy. Locally one could say $\Diamond(x = 1)$ instead of $\Diamond(x = 1)$, assuming that the scheduler is *fair* (but otherwise free to choose randomly).

3.4.6 Automata products

In the big-picture discussion about model-checking as refutation, one of the step is to calculate the intersection

$$[\![M]\!] \cap [\![\mathcal{A}_{\neg\varphi}]\!] \tag{3.21}$$

The intersection or conjunction construction is done on the corresponding automata. So, given two automata, we need to construct an automaton that represents the intersection of the corresponding languages (or the conjuction of the corresponding properties).

In principle, such constructions are known from standard automata. It also called the (or a) **product** construction as the states of the joint automaton consists of pairs of states from the contibuting automata.

The construction for Büchi-automata is more complex than for standard FSAs, and the complication concerns, not suprisingly, the accepting states. For standard automata, and accepting state of the product automaton is simply tuples of the original automata: a finite word in the intersection is accepted if both component automata reach one of their respective accepting state, it’s very straightforward. For Büchi-automata, it’s no longer that easy.

Indeed, in this section we **won’t** present the general product construction for Büchi-automaton, because we don’t need it in its full generality. We are after intersection in the situation of equation (3.21). In particular, the M is the Büchi-automaton that corresponds to the transition system under investigation. What makes it special is that, as Büchi automaton, all its states are accepting (and never gets really stuck since it’s stuttering). In this situation, the only contributing factor to the acceptance of the product is the Büchi automaton from the formula. That’s as it should be anyway, the specification determines whether property φ holds or not, the system M does what it does, and the accepting states of the product are solely determined by the formula.

Two kinds of products

In fact, the section discusses two kinds of automata constructions, i.e., composition operators on automata, called *synchronous* and *asynchronous* product. Both serve two different purposes. The synchronous product is the one we just sketched and will capture language *intersection*. It corresponds to the standard product construction known from standard FSAs. As also said, we only cover the product or intersection between two Büchi-automata, where one is special insofar that it comes from turning the program into such an automaton. In that sense, the way the synchronous product used here is *asymmetric* (normally products are symmtric, i.e., commutative).

The *asynchnonous* product here comes from the underlying compitation model. The presentation here is based on [20] and thus influenced by the choices as made in the Spin model checker. In Spin, systems consists typically of a number of processes running in parallel or concurrently. The input language of the Spin model checker, called Promela, resembles C. Processes run concurrently, communicate via shared variables and via channels. The semantics is a typical **interleaving** semantics. If we ignore synchroniation by channels, processes do their steps independently, i.e., it's an **asynchronous** execution model.

That kind of behavior is called **asynchronous** product here in the lecture (where we consider the system processes as automata). As said, when thinking in terms of processes or threads, one would not use the word "asynchronous product" but rather say, the processes run in parallel or concurrently and that in an indepedent or **asynchonous** manner (though synchronization when needed can be achieve by channels or locks. . .).

There are other forms of concurrency, for instance for systems with a global clock, where there processes run in lock-step. In that case, the processes run 'synchronlously' in parallel (with the clock as global synchronizing mechanism). And in such a setting, parallel composition reps. the product would be *synchronous*

In our setting here, processes run under an interleaving model ("asynchronously") and the composition of the system with the "formula" is done synchronously. The automaton representing the formula is there to observe or monitor the system, and this needs to be done fully in-sync with the model.

While talking about processes in Spin/Promela: as said, they don't run under a global clock as synchronizing mechanism. Instead, they can synchronize and communciate via *channels*. In our presentation, channels won't play a role. Nonetheless, automata equipped with buffered channel communication between them are a well-established model for *protocol specification*. Spin's programming or modelling language is basically processes + channels. That's a popular foundation for protocol verification. Formal models in that direction are known under various names and different notations. One name is *extended finite state automata* where extended means "extended by communication buffers" (mostly FIFO buffers).

Asychronous product

Let's start with the *asynchnonous* product, here of two automata. In the product, a step of the combined automaton consist of a step of one automaton where there other one does nothing

Definition 3.4.18 (Asynchronous product)**.** The *asynchronous product* of two automata A_1 and A_1, (written $A_1 \times A_2$, or $A_1 \parallel A_2$) is given as $(Q, q_0, \Sigma, F, \rightarrow)$ where

- $Q = Q_1 \times Q_2$,
- $q_0 = q_0^1 \times q_0^2$,
- $\Sigma = \Sigma_1 \cup \Sigma_2$, and
- $F = \{(q_1, q_2) \mid q_1 \in F_1 \quad \text{or } q_2 \in F_2\}$.

$$\frac{q_1 \rightarrow_1 q_1'}{(q_1, q_2) \rightarrow (q_1', q_2)} \text{ PAR}_1 \qquad \frac{q_2 \rightarrow_2 q_2'}{(q_1, q_2) \rightarrow (q_1, q_2')} \text{ PAR}_1$$

The product is defined for the *binary* case, i.e., the product of two automata. The definition is symmetric and associative, i.e. $A_1 \times A_2 = A_2 \times A_1$ and $A_1 \times (A_2 \times A_3) = (A_1 \times A_2) \times A_3$. The automata left and right of the equations are equal in the sense if being *isomorphic*. With associativity and commutativity, we can make use of n-ary product, i.e., the product of n automata, for which one can write $\prod A_i$. We could establish that there is also a neutral element wrt. the product, and then the $\prod_{i=1}^{n} A_i$ is also defined for $n = 0$ —the empty product should correspond to the neutral element— but let's not bother).

The core of the above definition is the way the *steps* are defined. The composed automaton can make a step, of either the left or else the right automaton can make a step.

Other ingredients of the definition are more a matter of taste. For instance, we have based the definition on automata with one initial state. One can easily do the "same" product construction in case one has automata with multiple initial states.

Another point is the alphabet: here we take the union of the alphabets. Some presentations would say one can compose only automata over the same alphabet (but also that is a non-central point as one can always "extend" the Σ_1 and Σ_2 to a common alphabet, before doing the composition).

More subtle is the definition of the final states. Remember also that the format of the automaton does not distinguish between standard automata and Büchi automata. The distinction is not based on the "format" or syntax of the automation, it's based in the interpretation of the automaton, in particular the interpretation of the acceptance set.

But for us, we don't care here much: remember that the automata are actually transformed from the system programs or processes. Those don't really have accepting state, resp. after the transformation from transition system or Kripke structure into an automaton, *all* states are accepting (which is a way of saying, that acceptance does not really matter). The good news is: if A_1 and A_2 are of that form that all their states are accepting, then that also holds for $A_1 \times A_2$ in the above definition.

Let's have a look as some toy problem represented implemented as two processes running asynchronously.

Example 3.4.19 ($3n+1$ problem). Assume 2 non-terminating asynchronous processes or automata A_1 and A_2 and a shared variable x. A_1 tests whether the value of x is odd, in which case updates it to $3*x+1$. A_2 tests whether the value of a variable x is even, in which case updates it to $x/2$.

The question is: Does the corresponding function *"terminate"* for all inputs x? Let's formulate "termination" as the following LTL property

$$\Box\Diamond(x \geq 4) \qquad \text{or negated: } \Diamond\Box(x < 4) \tag{3.22}$$

□

In the problem, "termination" is meant reaching the endless cycle $4 \to 2 \to 1 \to 4 \ldots$.

The "$3n+1$" problem is a long-standing open problem. It's known under different names (Collatz's problem, Hasse-Collatz problem, Ulam's conjecture, Kakutani's problem, Thwaites conjecture, Hasse's algorithm, or the Syracuse problem). The problem is not intended of what model checking can do or is good at. It's not even intended to illustrate a *typical* way asynchronous products are used in practice. Remember that the asynchronous product is intended to model concurrency. The problem, however, is *not* concurrent.

What makes it also atypical for standard model checking is that it's an *infinite* problem. The formulation makes a statement for arbitrary n, and that's intended for infinitely many natural numbers (the problem is not meant as saying the function terminates for all numbers up-to `MAXINT` ...). The dependence on some input or some other parameter makes it a *parametrized* problem and *parametrized model checking* is a sub-genre that tries to come up with techniques dealing with parametrization. The parameter can be an input (like for the $3n+1$-problem), but more conventional would be to check some property for a system $P_1 \parallel \ldots P_n$ consisting of an arbitrary number n of copies of the same system, where all the P_1 are identical (perhaps up-to their process identity). Think of using model checking mutual exclusion not for an implementation of the the 5-philophers problem but for the n-philosophers problem.

Remarks about the notation in the example and their interpretation The asynchronous product is given actually slightly (but not conceptually) deviating from the mathematical definition given above. The definition given before was for automata, but now, we are considering "processes" or Kripke structures or transition systems. But also that is not 100% correct according to the earlier definition or at least not at first sight. Transition systems were introduced as "graphs" where the states give values to atomic propositions. That's a very low level way of seeing things.

The example make use of transition system which allow more convenience in notation (one could call it "applied transition systems" or "symbolic"). It's still ultimately the same, but we allow to have programming variables (x in this case) which can be changed and checked. We don't have just a set of propositional variables any more, but *predicates* over the programming variables (like $even(x)$ and $odd(x)$). As far as the logic is concerned, that seems to go into the direction of "first-order LTL" (having sorts and predicates and variables), but we are not actually doing that, for instance we don't introduces $\forall$ and $\exists$, which gives first-order logic it's actual power). The extension is more to the transition-system side of things and $odd(x)$ is not so much seen as a predicate of the logic, but a boolean expression or guard as used in the underlying programming languages. On the LTL side of things, $odd(x)$ can be seen as a proposition which is either true or not depending on the current value of x. So the interpretation of the transition systems and their behavior and how it connects to LTL should be fairly transparent.

The transition systems here also "deviate" from the core definition in that the transitions are *labelled*. The labels are **not** primarily meant as being symbols from an alphabet that the LTL speaks about. LTL speaks about the states. The transition labels are here represent actions that help to specify what the ("symbolic") transition system does when taking the edge. The interpretation of the label $x := 3x + 1$ is fairly obvious, the intension is that the value of x is accoringly modified when going from the source-state to the target state. The other kind of transition is marked by a boolean condition or guard ($odd(x)$ or $even(x)$). The intention is that it's a *conditional transition* that can be taken if the guard is *true* in the source state.

Spin, resp. Promela allows that kind of statements. The language called Promela quite resembles C at the surface (one can also refer to native C), but there is one point where the semantics of Spin **deviates significantly** from C. Writing in C the sequence

```
(x%2); x = 3x+1; ...
```

simply calculates the remainder of x modulo 2, then forgets the outcome and updates the value of x according to the expression on the right-hand side of the assignemt. In other words, the above snippet can be simplified to `x = 3x+1`.

In Spin, in contrast, the first expression is interpeted as a so-called **guard**: the expression `x%2` is calculated and the outcome determines whether to continue or not. In the tradition of C, in case the outcome is `0`, it's interpeted as `false`, if different from `0`, it's seen as `true`. That means `x%2` corresponds (in a C-typical formulation) to the predicate or guard $odd(x)$. In case the guard evaluates to true, it does not do anything (as in C), in case it evaluates to false, it **blocks** and prevent the rest of the process from proceeding. So, it acts as a **synchronizing** construct:, the transition is enabled or not depending on the "circumstances". Of course, the circumcstances, i.e., the value of x, can change by interference from a second process, at which point the transition becomes enabled and the process can thereby proceed.

This explanation should be enough to understand the example. A few words on guards in concurrent programs might still be of general interest. Syntactically, the solution in Spin is a bit obscure one may say. It basically says, if one uses an expression in place of a statement, then the expression has *synchronizing powers* (like stopping the execution of the process). That's a truly *radical* change of interpretation of expressions! Synchronization is at the core of concurrent programming; hence it's probably a bad idea to hide some key ingredient, conditional guards, in reinterpreting expressions ("BTW, expressions now mean sometimes something really novel and powerfull compared to C, even if they look the same"). in defense one could say, a decent C programmer would probably not use expressions as assignments anyway, bit still.

Other languages would make the special nature of guards more obvious, namely by introducing *special* syntax for it. If b is a boolean expression, then if one wants to use it with synchronizing power, the programmer would have to write special syntax, writing perhaps `await(b)` or similar, making that transparent.

For people experienced in concurrency programming, there is another concern wrt. guards like *odd* or similar (not really present in the quite simple example). The previous discussion concentrated on "syntactical" issues or language pragmatic (like questioning the wisdom of avoiding special syntax). The point now may be even more serious. In the transition systems below (and in the code snippet above), there are two steps, namely first the guard is check, for instance $odd(x)$ and, in case the guard had evaluated to true and after taking the guard-labelled transition, then the assignment is done. The problem is: the guard itself has no effect (guards and expression are supposed to be side-effect free); it is intended to enable (or not) the subsequent effect. The problem is: whether or not the guard is true is checked but that may *change* after the *odd* transition and before taking the subsequent $x := 3x + 1$ transition. In other words, the two transitions are *not atomic* which may in general lead to problems. On a related note: it's also questionably whether the assignment $x := 3x + 1$ is guaranteed to be atomic.

To be useful as concurrent programming languare or language, Spin offers the possibility enforce **atomicity**. Sping supports slightly different levels of guaranteeing atomicity, but the details don't concern us here.

Basically one can *group* the two steps togther, $[odd(x); x := 3x+1]$ where we use brackets [and] to denote that the execution should be atomic resp., without interleaving. Spin does not use brackents but would use keywords like `atomic{...}` or `d_step{...}`.

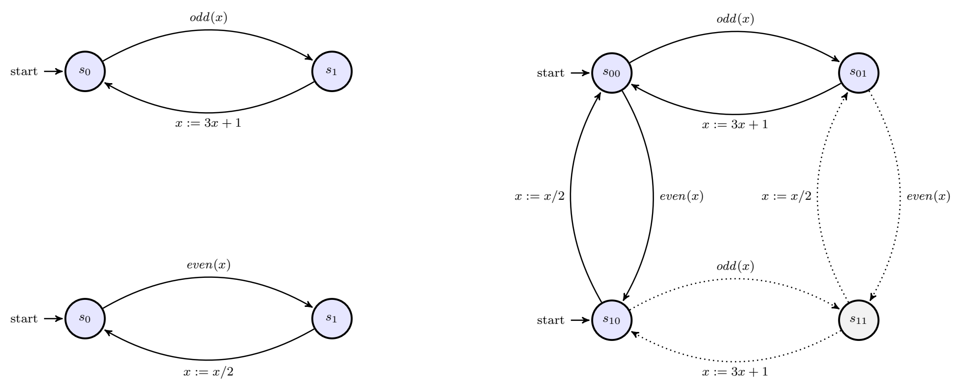
In any case: the shown pattern is rather common: in an atomic section, the first one is the guard, and the rest is the effect. It is not so comming (and a bit tricky) to use guards in the middle of an atomic section. Since that pattern is so common, other languages would offer specially syntax for it `await(b){statements})` and there are different names for it (conditional critical region etc). Also related in the notion of *guarded commands*. In transition systems, instead of having two separate transation, one would cound then have labels of the general formal $\overset{g \triangleright a}{\to}$, where g is the guard and a the action or effect. In our case, one label could be $odd(x) \triangleright x := 3x + 1$.

Anyway, atomicity is not really a problem in our particular (not very typical) example, as the $3n + 1$-problem is not really concurrent anyway.

After all the background information, back to the example and the transition systems for them

Example 3.4.20 ($3n + 1$-problem: asynchronous product). The program informally described in Example 3.4.19 can be represented by the transition systems from Figure 3.11. In the product automaton, the state s_{11} is /unreachable/. When starting at the initial state, the dotted arrows can never be taken. □

Pure automata or transition systems We have made use of the more high-level version of the transition systems, referring to variables like x and transitions with labels that have a "semantics". Besides that, the transition system description as "symbolic", not concrete as it referred to x and not a concrete value of x. So, the description was still a "parametrized problem". We are here *not* looking into parametrized problems. We can only model check



(a) A_1 and A_2 separately (b) $A_1 \times A_2$

Figure 3.11: $3n + 1$-problem transition systems: asynchronous product

concrete (non-abstract, non-symbolic) models. So we need to pick a concrete initial value for x. Example 3.4.21 below chooses $x = 4$. This leads to what Holzmann calls a *pure* transition system. Now, the value of x becomes part of the "transition-system state". So the "state" or world now consists of the control-flow state (for instance initially (s_0, s_0) written also s_{00}) together with the state of the memory, i.e., the value of x.

Example 3.4.21 ($3n$ + 1-problem: Pure transitition system)*.* If we start the transition system of the $3n$ + 1-problem from Example 3.4.20 with a value of 4, we obtain the following concrete or pure transition system from Figure 3.12.

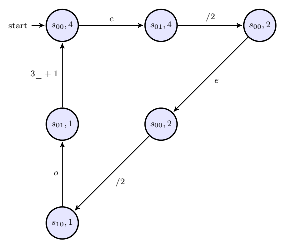


Figure 3.12: Pure automaton: terminal loop

□

In the states, $(s_{00}, 4)$ is supposed to represent a state where both of the original automata A_1 and A_2 are in their respective initial state s_0 and where x has the value 4.

The edge labels can be ignored; they are needed only for the non-pure representation, in the pure transition system they are just kept for readability. In particular: the LTL formula specifying termination does *not* speak about those labels.

We make "short labels" where e and o stands for $even(x)$ and $odd(x)$ in case of the transitions corresponding to guards, and the action labels are similarly shortened. As said, for us, the labels are not really part of the pure transition system anyway. For example, $even(x)$ is true in state (for instance) $(s_{00}, 4)$ anyway (with x understood as carrying the even value 4). So that fact is just captured by a transition $(s_{00}, 4) \rightarrow (s_{01}, 4)$

(and the label is more a reminder for the ready why there is that transition). Since, in that state, the guard $odd(x)$ is false, there are no outgoing transitions marked with $odd(x)$: true guards are represented by transitions in the pure representation, false guards represented by the absence of a transition.

Synchronous product

As said, we define the *synchronous product* for 2 Büchi-automata *in a special case*, where for one of the automata, all its states are accepting. That is the case we are interested in here, where one of the BAs, the one describing the system, is translated from the transition system or Kripke structure. In that translation, all states are marked as accepting, so we can focus on that special case.

Indeed, the general case for two arbitrary BAs is slightly more complex. The slightly tricky part would be how to define the accepting states of the product. Apart from that, it's straightforward.

The general intention of such a "product" construction is that he composed machine accepts the *intersection* of the languages of its two constituents. That's conceptually achieved that both automata run in lock-step. For standard (non-Büchi) FSA, a commong word is accepted, if both A_1 and A_2 reach a respective accepting state. That is easy.

For BAs, we have repeated reachability of accepting states, it becomes more tricky: each A_1 and A_2 has to reach at least one of its accepting states infinitely often. But it's not that they re-visit their respecting accepting state always at the same time! That makes the general construction a bit more tricky (but not really challenging; one can figure it out oneself). If one of the automata, say A_1, has *only* accepting states, things get more easy: one can basically ignore A_1, and base acceptance of the product construction in the accepting states of A_2 alone. That's also intuitively what we want to do in model checking. The acceptance condition of A_2 represents a LTL requirement we want to check. The other automaton just "executes", there are no good runs or bad run in A_1 as such, the goodness/badness of the runs is judged by the acceptance conditions in A_2.

It should be noted that in Holzmann [20] the product automaton for NBAs is defined incorrectly (and in previous years, that error showed up also in our slides). As another side remark: for *generalized* BAs, the construction of the accepting conditions for the sychronous product is slightly more elegant compared to standard BAs.

Definition 3.4.22 (Synchonous product (special case))**.** The *synchronous* product of two finite automata $\mathcal{A}_1$ and $\mathcal{A}_2$ (written $\mathcal{A}_1 \otimes \mathcal{A}_2$), for the special case where $F_1 = Q_1$, is defined as finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ where:

- $Q = Q_1 \times Q_1$
- $q_0 = (q_{01}, q_{02})$
- $\Sigma = \Sigma_1 \times \Sigma_2$.
- $\rightarrow = \rightarrow_1 \times \rightarrow_2$
- $(q_1, q_2) \in F$ if $q_2 \in F_2$

There is a small final piece we need to take care of, that is stuttering. We discussed the issue already. LTL works on infinite paths, so it must be prevented that that transition system just stops progressing. That's easy to achieve, just let a terminated system continue doing extra do-nothing steps. That is called *stuttering*. The following Definition 3.4.23 adds stuttering to the resulting Büchi-automaton (not the transition system).

Definition 3.4.23 (Stutter closure)**.** Given a finite-state automaton $\mathcal{A}$ (Büchi or otherwise), the stutter-closure of $\mathcal{A}$ corresponds to $\mathcal{A}$ (same states, labels, same initial and final states) except that some extra ε-labelled self-loops added to the transitions for each state without outgoing transition.

Example: synch. product for $3n + 1$ system and property

The picture is from Holzmann's lecture material.

Example 3.4.24 ($3n + 1$-problem: product)*.* For the $3n + 1$-example of this section, the product of the automaton from Figure 3.12 with the BA for the LTL property of equation (3.22) from Example 3.4.19 is shown in Figure 3.13.

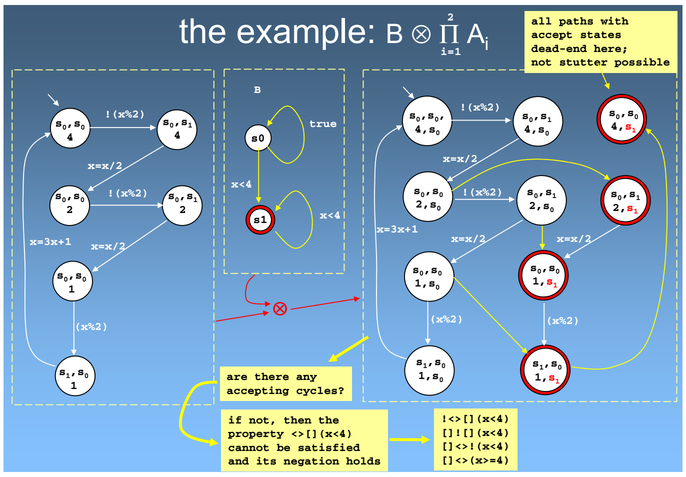


Figure 3.13: Product of system automaton and automaton representing the specification

□

3.5 Model checking algorithm

We have now a number of things in place: the general idea of LTL model checking and the definition of products. But two key technical ingredients are missing. That's how to translate and LTL-formula to an Büchi automata and how to check for emptyness of a Büchi automaton. Those are the more challenging parts of the whole construction, and we cover them to some extent in this section.

3.5.1 Checking for emptyness

One problem to solve is: given an Büchi-automaton, check if its language is empty. For standard automata over finite words, the problem is straightforward: start at the initial state and see if one of the final states is *reachable*. A simple search will do, like depth-first search or some other strategy.

For Büchi-acceptance, the problem gets more complex, but not dramatically so. It involves checking whether some acceptance state can be reached infinitely often. Details depend on whether one deals with plain Büchi-automata or the generalized ones (or some other kinds infinite word automata, which we have not covered).

So, one-time reachability is not enough, one needs to check for some form of *repeated* reachability: from the initial state, reach an accepting state, and then again, and again ... The corresponds to reach an accepting state, and then see if there exist a cycle in the graph starting from that. Because of that shape, it's sometimes illustrated by saying one is looking for a *lasso*.

One could thus go for a lasso-hunting expedition, hunting for lassos, of course, not hunting with lassos.... When done properly it would work. But as so often, an initial plausible idea, even if working, leaves room for algorithmic improvement. Here, looking for reachable states that are the starting and end point of a cycle one reachable state after the other will be hugely wasteful. One will re-discover and explore paths through the graph over and over again.

A better idea is based on the notion of *strongly connected components*. It a concept from graphs and may be known from lectures like "Algorithms and Data Structures".

Definition 3.5.1 (SCC)**.** A subset $S' \subseteq S$ in a directed graph is *strongly connected* if there is a path between any pair of nodes in S', passing only through nodes in S'. A *strongly-connected component* (SCC) is a *maximal* set of such nodes, i.e. it is not possible to add any node to that set and still maintain strong connectivity.

There are efficient algorithms calculating those, for instance one based on doing depth-first search two times, the second time backwards and in a particular order, but we won't repeat them here.

Example 3.5.2 (SCC). Consider the graph of Figure 3.14. Subsets which are strongly-

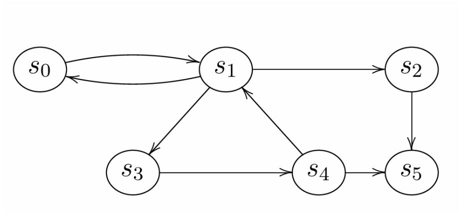

$$\begin{tikzcd}
s_0 \arrow[r, bend left] & s_1 \arrow[l, bend left] \arrow[r] \arrow[dl] & s_2 \arrow[d] \\
s_3 \arrow[r] & s_4 \arrow[ul] \arrow[r] & s_5
\end{tikzcd}$$

Figure 3.14: Strongly connected component

connected are $S = \{s_0, s_1\}$, $S' = \{s_1, s_3, s_4\}$ and $S'' = \{s_0, s_1, s_3, s_4\}$. From those, only S'' is a strongly connected component. □

A strongly connected components is some form of “cluster of cycles”. As long as one stays within the component, one keeps on cycling in the cluster, if one leaves it, one enters another clusters and there is no way back.

Strongly connected components can be use address the emptyness problem of Büchi-automata. It’s based on the following observation. If one considers an infinite run through a system with finitely many states, it’s unavoidable that some states are visited infinitely often. That is clear.

One can formulate something stronger. Assume an infinite run σ through such a system, there comes point in time in the run, from where on *all* subsequent states are visited infinitely often.

That being so, in that run, the mentioned states that visited infinitely often in a suffix of σ are states which are obviously strongly connected.

So, that’s the core observation:

> given a Büchi automaton $\mathcal{A} = (Q, s_0, \Sigma, \rightarrow, F)$ with accepting run σ, then, there is some suffix σ' of σ s.t. every state on σ' is reachable from any other state on σ'. In other words, that set of states is strongly connected and is is reachable from an initial state and contains an accepting state.

Checking non-emptiness of $[\![\mathcal{A}]\!]$ is equivalent to finding an SCC in the graph of $\mathcal{A}$ that is reachable from an initial state and contains an accepting state

There are different algorithms for find SCCs, for instance Tarjan’s version based on depth-first search, or the *nested depth-first search* version implemented in Spin.

If the language $[\![A]\!]$ is non-empty, then there is a *counter-example*. Moreover, the counter-example can be represented in a finite way. It is *ultimately periodic*, i.e., it is of the form $\sigma_1\sigma_2^\omega$, where σ_1 and σ_2 are finite sequences

3.5.2 Translation LTL to Büchi-automata

The translation we show or at least sketch here is from LTL to Generalized Büchi automata (GBA), not to the plain version of Büchi automata. It follows the presentation and terminology of Baier and Katoen [2]. That can be compared to Thompson’s construction for regular expressions (as discussed earlier). As Thompon’s construction, it’s a *structural* translation, i.e., given by induction on the structure of LTL.

The crucial idea, abstractly, is to connect the syntax and semantics. One can compare that Hintikka-sets or similar constructions for FOL.

Before we do the construction, let’s have a look at a few LTL examples and how they can be represented by Büchi automata.

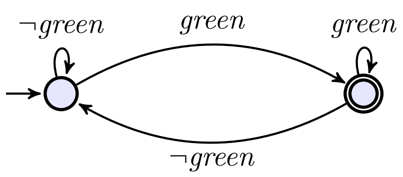


Figure 3.15: Infinitely often green

Example 3.5.3. We have seen the example already, under the name *recurrence*. The infinite occurrence of a property, here *green*, is captured by the formula $\Box\Diamond green$. The corresponding and straightforward Büchi automaton is given in Figure 3.15. The example is a good illustration on the Büchi-acceptance: the infinite occurrences of *green* is directly connected to the infinitely many visits to the accepting state of the automaton. prototypical □

Example 3.5.4 (Response). Also this kind of property we have seen already, in Example 3.2.8. Response can be captured in LTL as:

$$\Box(request \to \Diamond response)$$

Let's abbreviate it to $\Box(a \to \Diamond b)$. The Büchi-automaton accepting that response property is given in Figure 3.16. □

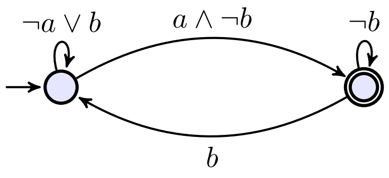


Figure 3.16: Response

Example 3.5.5 (Stabilization). Stabilization (or persistence) can be captured in LTL as:

$$\Diamond\Box a$$

The Büchi-automaton accepting that response property is given in Figure 3.17. □

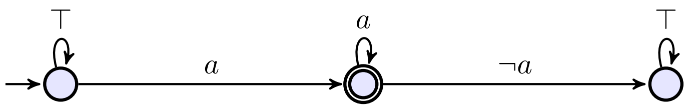


Figure 3.17: Stabilization

The last example captures the dual property from Example 3.5.3. The Büchi automaton (perhaps) feels also slightly different from the others . You can pause before reading on to reflect what's different.

How does the stabilization automaton works anyway? It start's at the initial state, it tries to watch out when the a stabilizes, i.e., to detect the point when that happens. Initially, there's a don't-care self-loop, marked by $\top$, which means it can be always taken. If an a is detected, the automaton can move on to the middle state, and if from that point on, there are only a's, the word is accepted. If it so happens that, after proceeding to the middle accepting state, $\neg a$ occures, it is forced to proceed to the third and non-accepting state and stay there forever, thereby rejecting the word. Note, the previous examples never got stuck in this form in an non-accepting states, there was always the hope for an edge back to an accepting state.

The rejecting behavior from above, moving in into the middle state and then moving out at the next occurence of a $\neg a$, does not mean that word being scanned violates the property.

It's just that in its initial state, the automaton has misjudged the situation: seeing an a, instead of staying in the initial state, the automaton took the a as a sign of the beginning of the stabilized sequence of a. Later it may turn out that this was premature, and staying longer in the initual state will lead to acceptance (in case the sequence satifies the stabilization property). Only it's not knowable how long to wait until exiting the initial state. The definition of acceptance does not care about that: a word is accepted if there **exists** an accepting run, but it does not promise that, given an infinite word, there is exactly one run.

So, one *crucial* difference distinguishing this Büchi-automaton from the two other ones is that it is **non-deterministic**. The situation is worse. For that particular LTL-property and for this particular non-deterministic Büchi-automaton, there is no equivalent deterministic one. We will not prove that, but the discussion around the example my give a feeling that there's not hope for deterministic acceptance for stabilization.

Basic idea of the translation

Here, we don't give construction yet, but some general remarks and some "insightful" property.

The translation translates a minimal version of LTL, i.e., containing only $\bigcirc$ and U as only temporal operator. The most tricky part in the construction, not surprisingly, is how to deal with the until-operator. Until-operators can encode properties like recurrence and stabilizations just discussed. Such properties need properly arranged accepting states, either to be infinitely visited or avoided forever. For each until occurring in the formula to translated, one needs such an arrangement. The translation is easier, if one does not work with plain Büchi automata with one set of accepting states. It's easier to use generalized Büchi-automata, so one can do one set of accepting states *per until.* And then in a later step, one can translate that automaton to a standard Büchi automaton. GBAs were defined in Definition 3.4.11.

Let's refer to the generalized Büchi-automaton for a formula φ by $\mathcal{G}_\varphi$. To get a grip in the construction, we need also get a *mental picture*: what are the states of the automaton, what do they represent, and how ow are they connected by transitions; we deal with acceptance later.

One key idea in this construction is to connect the syntax with semantics, so to build the model "using" formulas. For LTL, it means the states consist of *sets of formulas*. That is a very general idea, and actually, to use syntactic material like sets of formulas to build a semantic model can also be found elsewhere. For instance, completeness of proof systems like for first-order logic can be established that way.

But, as said, the idea is pretty general and thus vague, so what does that mean for us? What we want is a construction such that the states consists of all formulas that hold in that state and the transitions or edges between the states lead from one state to the successors that reflects the changes in the set of formulas that hold. That will basically reflect the interpretation of the $\bigcirc$-operator.

We said, the states represent *all* formulas that hold in that state. We should qualify that a bit, it's about all *relevant* formulas. If we construct the automaton for $\mathcal{G}_\varphi$, then only formulas that have something to do with φ need to be considered, which will be φ, all its sub-formulas and negations thereof.

Let P be the (finite) set of propositional variables of interest, i.e., those mentioned in φ. The *alphabet* Σ of the Büchi-automaton then is *sets* of such propositinal atoms, i.e,

$$\Sigma = 2^P \ .$$

Let's write A, $A', A_i \in \Sigma$ etc. for such sets. Let's look at in infinite sequence $\sigma = A_0A_1A_2\ldots$ such that $\sigma \in [\![\varphi]\!]$ resp.

$$\sigma \models \varphi \ . \tag{3.23}$$

We commented earlier that the terminology concerning states, paths etc. shifts a bit (and we commented also that this should be unproblematic.) We called the models of LTL formulas *paths* (and used the symbol π for it, where, to add perhaps to the confusion, the pieces of the paths where called states (see Definition 3.2.1). Here, the sequence from equation (3.23) is of the appropriate form.

The infinite sequence σ corresponds to a word accepted by the automaton $\mathcal{G}_\varphi$ we want to construct. As such it corresponds to a word following the edges of the automaton, labelled by Σ. However, it also corresponds to a sequence of states of the automaton: the states are built up from sets of (relevant) formulas that are supposed to hold in that state, and that inclused propositional atoms.

Now, how about the states of $\mathcal{G}_\varphi$ then? We said that states are sets of formulas, and the automaton is constructed in such a way that the formulas in the states are those which hold in $baof\varphi$. And that's not just propositonal atoms, but also compount formulas, in particular, temporal formulas. That means, the sequence $\sigma = A_0A_1A_2\ldots$ of (3.23) cannot be not directly the sequence of states. It corresponds to that state sequence only as far as the propositional atoms are concerned.

Given the word A_i from above, let's consider an infinite sequence of B_i so that each B_i extends the corresponding A_i (by subformulas of φ). I.e., $B_i \supseteq A_i$. Let's refer to that extension of σ by $\hat{\sigma} = B_0B_1B_2\ldots$.

The intention for that sets B_i, ultimately the states of the automaton is that they contain exactly the *language* of that state. That is strong requirement. It also implies that for each formula ψ, either ψ is contained in the state or not. That's a key ingredient of the mention principle to build the model, here the states, out of the syntax, sets of formulas.

The language of a state is meant in the standard way: all the infinite words that are accepted when starting at that state as initial state. With that in mind, we can state the following about the sequence from equation 3.23, connecting the automaton-states and their formulas with the rest of the sequence

$$\psi \in B_i \qquad \text{iff} \qquad \underbrace{B_i, B_{i+1}B_{i+2}\ldots}_{\sigma^i} \models' \psi \tag{3.24}$$

$$\psi \in B_i \qquad \text{iff} \qquad \underbrace{A_i, A_{i+1}A_{i+2}\ldots}_{\sigma^i} \models \psi \tag{3.25}$$

The suffix-path σ^i, which satisfies ψ exactly when it is mentioned in B_i (and additionally the intention is that it is a word in the language of the state B_i. The $\hat{\sigma}$, a state sequence, corresponds to an accepting run in $\mathcal{G}_\varphi$

The pool of relevant formulas: the closure of φ As said, the states of $\mathcal{G}_\varphi$ are built for sets of formulas. However, it's not all possible combinations of all possible formulas. There are restrictions on the sets of formulas. One of the restrictions is that one focuses on formulas that can conceivably play a role in an automaton checking φ. Since there are only finitely many subsets of that formula, thus results in a *finite-state* automaton.

Given a formula φ, the pool of formulas taken into account is φ and all its subformulas and negations thereof. We write $closure(\varphi)$ for that set.

Definition 3.5.6 (Closure)**.** The *closure* of an LTL formula φ, written $closure(\varphi)$, is the set of all subformulas of φ and their negation (where ψ and $\neg\neg\psi$ is identified).

Meaningful sets of formulas, only: maximal consistency. Considering elements from the closure of φ is only one restriction. Not all elements from $2^{closure(\varphi)}$ make sense considering the purpose of the states formed by sets of formulas. As said, the sets or states $B \subseteq 2^{closure(\varphi)}$ of the automata are intended to contain exactly all the formulas from the closure that are in the "state-language" of that state, as formulated in equation (3.25), which involves words with elements B_i from $2^{closure(\varphi)}$. Of course, the official semantics of automata speaks about words with letters from $\Sigma = 2^P$ following the edges, not words over $2^{closure(\varphi)}$ but the construction couples them tightly together, for instance in that $B_i \supseteq A_i$ (among other things).

Coming back to the restriction: there is no use of meaningless sets of formulas as states. Literally meaning-less in that they have no interpretation or semantics. For propositional atoms, for instance, it makes no sense to have a state B which contains both p and $\neg p$, which would be a state with the intended interpretation that both p and its negation holds there. Such contradictions neeed to be avoided not just for propositional variables, but for all formulas. So it's required for states B that they don't contain both ψ and $\neg\psi$ at the same time. That assures that for each state, there exists a variable assignment for the propositional variables that make the set B true, at least the propositional, non-temporal part. Such a compositionally non-contradictory set is called **consistent**.

Is that all required for meaningful and useful states? Not yet. We have restricted ourselves to sets of formulas from $\mathit{closure}(\varphi)$, but we also want that for all such formulas ψ, either ψ or $\neg\psi$, so that by looking at the state, the logical status of the formule is determined by whether the formula it's mentioned in the, or its negation.

So the sets of formulas constituting the states of $\mathcal{G}_\varphi$ or not just consistent subsets of $\mathit{closure}(\varphi)$, they are *maximal*: Since in the construction, each subformula of φ is either contained in the state or else $\neg\varphi$, one cannot add any further formula from $\mathit{closure}(\varphi)$ without violating *consistency*. So, states are **maximally consistent sets of formulas from** $\mathit{closure}(\varphi)$. The book [2], on which the material here is based, calls those sets *elementary* sets, but I think calling them maximally consistent is preferrable.

We have discussed at some length the shape of the states of the automaton to construct. That's enough to give meaning to the propositional, non-temporal part of the formula. Of course, the interpretation of formulas of the form $\bigcirc\psi$ semantically is not decided locally at one state, it speaks about the next states. So, if a formula $\bigcirc\psi$ contained in a state (and thereby intended to hold there) is actually true, depends on whether that state is added by edges only to states which mention ψ. So, for temporal properties, one has to focus on the edges properly (which will be done later), but as far as the constuction of the states are concerned, we are done, as states are for the interpretation of local and propositional aspects, whereas the edges are (additionally) take care of the temporal aspects. For $\bigcirc$, that is true, the interpretation of $\bigcirc\psi$ is purely non-local, so there is no restrinction concerning ψ formulas, expect the propositional ones discussed, namely that already propositinally, that exacly ψ or else $\neg\psi$ must be included in the state,

But what about the other temporal operator, the until? As temporal operator, it has no local aspects to take care of in the construction of the states, or has it? The until-operator is the trickiest to translated and actually, there is a local aspect of until, and it is the following: A formula $\varphi\ U\ \psi$ holds in the current state if ψ holds. That also means, the set of formula can contain local contradictions involving until, for instance if it contains both ψ and $\neg(\varphi\ U\ \psi)$.

Definition 3.5.7 (Propositional consistency)**.** A set B is consistent wrt. propositional logic (and relative the closure of φ) if

1. $\psi \in B$ implies $\neg\psi \notin B$.
2. $\bot \notin B$.

Definition 3.5.8 (Closed wrt. propositional entailment)**.** A set B is closed under propositional entailment if

1. $\varphi_1 \wedge \varphi_2 \in B$ iff $\varphi_1 \in B$ and $\varphi_2 \in B$.
2. if $\top \in \mathit{closure}(\varphi)$ then $\top \in B$

Definition 3.5.9 (Local consequences of until)**.** A set B is closed under local entailments wrt. the until operator (and relative the closure of φ) if

1. $\varphi_2 \in B$ implies $\varphi_1 \; U \; \varphi_2 \in B$
2. $\varphi_1 \; U \; \varphi_2 \in B$ and $\varphi_2 \notin B$ implies $\varphi_1 \in B$.

Definition 3.5.10 (Maximality)**.** A set is maximal (relative to the closure of φ), if for all $\psi \in \mathit{closure}(\varphi)$

$$\psi \notin B \qquad \text{implies} \qquad \neg\psi \in B \; .$$

The following definitions just combines the previous one into one:

Definition 3.5.11. Given a LTL-formula φ. A set B is *maximally consistent* (or elementary) wrt. φ if it is propositionally consistent, closed under propositonal entailment and locally entailed formulas wrt. until, and if it is maximal.

After all this explanations and definitions, let's illustrate it by an example. The example illustrates the concept of closure, also of maximal sets of formulas, in particular maximally consistent extensions as the B_i's extending the A_i's in equation (3.25).

Example 3.5.12.

$$\varphi = a \; U \; (\neg a \wedge b) \tag{3.26}$$

The closure of that formula is

$$\{a, b, \neg a, \neg b, \neg(\neg a \wedge b), \neg a \wedge b, \varphi, \neg\varphi\} \; . \tag{3.27}$$

The following combinations are maximally consistent

$$\begin{array}{lcllllll}
B_0 & = & \{ & a, & b, & \neg(\neg a \wedge b), & \varphi & \} \\
B_1 & = & \{ & a, & b, & \neg(\neg a \wedge b), & \neg\varphi & \} \\
B_2 & = & \{ & a, & \neg b, & \neg(\neg a \wedge b), & \varphi & \} \\
B_3 & = & \{ & a, & \neg b, & \neg(\neg a \wedge b), & \neg\varphi & \} \\
B_4 & = & \{ & \neg a, & \neg b, & \neg(\neg a \wedge b), & \neg\varphi & \} \\
B_5 & = & \{ & \neg a, & b, & \neg a \wedge b, & \varphi & \}
\end{array}$$

The first two columns cover all 4 consistent combinations of involving a and b, respectively their negation. Per combination, the third column is fixed: having chosen the truth values for a and b fixes the truth value of $\neg a \wedge b$, i.e., whether $\neg a \wedge b$ is added or its negation $\neg(\neg a \wedge b)$. The last column is for φ, a temporal formula mentioning until. There, based on the conditions from Definition 3.5.9, a choice for a and b may determine wether φ is added or $\neg\varphi$. This is the case for B_4 and B_5.

For B_4, since $\neg(\neg a \wedge b)$ holds and additionally $\neg a$

Example 3.5.13. Let us the formula $\varphi = a\ U\ (\neg a \wedge b)$ again and its closure from equation (3.26).

Let's look at a small satisfying path or run for that until.formula. In the transition system, we assume that a holds in the initial state, b holds in the second. Let also a hold in the second state, though the semantics of until does not inist on that. At that point the execution already satisfies the formula, but for the sake of illustrating the concepts here, let's assume a third state where b holds.

The is shown in the following sequence:

$$\sigma = \{a\}\{a,b\}\{b\}\ldots \;=\; A_0A_1A_2\ldots \tag{3.28}$$

The sets A_i are originally, in the transition systems, the propositions that are supposed to hold in the states of the transition system. In the Büchi-automaton, it refers to a word accepted by the automaton via following the edge labels (see 3.18).

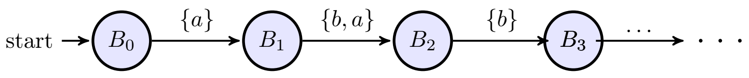


Figure 3.18: Run of the Büchi automaton

The states B_0, B_1 ... mentioned in the run are states of the Büchi-automaton and thus maximally consistent sets, as explained. Additionally, as indicated in equation (3.25), the $B_i \supseteq A_i$, i.e., each B_i is a maximally consistent extension of the corresponding A_i. For example for B_0, one could have:

$$B_0 = \{a, \neg b, \neg(\neg a \wedge b), \varphi\} \tag{3.29}$$

B_0 must contain a, it cannot contain $\neg a$, and for the other subformulas, $\neg a \wedge b$ is added in negated form, and φ as is. □

Construction of GNBA: general

- given P and φ
- given φ, construct an GNBA such that

$$\mathcal{L}(B) = \mathit{words}(\varphi)$$

- 3 core ingredients
  1. states = sets of formulas which (are suppsed to) "hold" in that state
  2. transition relation: connect the states appropriately,
  3. transitions labelled by sets of P.
- labeled transition connected states to match the semantics: for $\bigcirc\varphi$:
  go from a state containing $\bigcirc\varphi$ to a state containing φ. Label the transition with the APs from the start state.

Transition relation

$$\delta : Q \times 2^P \to 2^Q \tag{3.30}$$

- if $A \neq B \cap P$: $\delta(B, A) = \emptyset$
- if $A = B \cap P$, then $\delta(B, A)$ is the set B' such that
  - for every $\bigcirc\psi \in \mathit{closure}(\varphi)$:

    $$\bigcirc\psi \in B \quad \text{iff} \quad \psi \in B'$$

  - for every $\varphi_1 \; U \; \varphi_2 \in \mathit{closure}(\varphi)$:

    $$\begin{array}{lll} \varphi_1 \; U \; \varphi_2 \in B \quad \text{iff} & \varphi_2 \in B & \text{or} \\ & (\varphi_1 \in B \quad \text{and} \quad \varphi_1 \; U \; \varphi_2 \in B') & \end{array}$$

Accepting states

$$F_{\varphi_1 U \varphi_2} = \{B \in Q \mid \varphi_2 \in B \text{ or } \varphi_1 \; U \; \varphi_2 \notin B\} \ .$$

Chapter 4
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

4.1 Introduction

This chapter covers another prominent temporal logic, *computation tree logic* or CTL. Material is taken from the books [2] or [12]. 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.

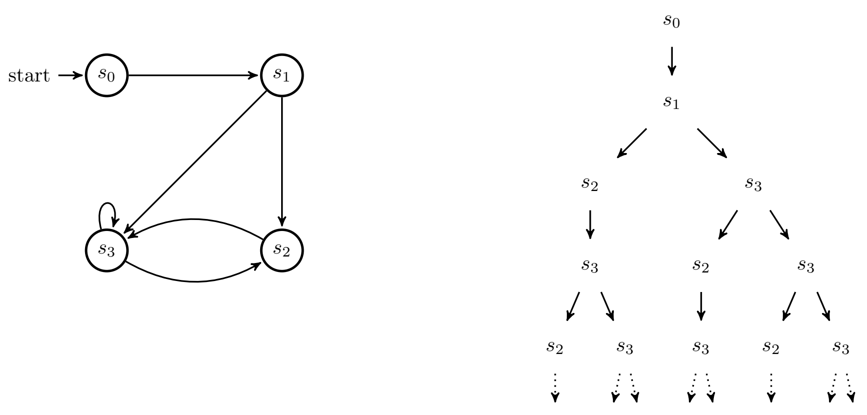
But CTL is also quite different from LTL. LTL treats time as *linear* and at the core the satifaction relation of LTL formula is defined for infinite liniear sequences (traditionally called paths in LTL).

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 4.1b. 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 4.1a with LTL or another linear-time logic, the system would satisfy the property, if all runs or paths starting from the initial state would satisfy it. A tree as the one from Figure 4.1b 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.

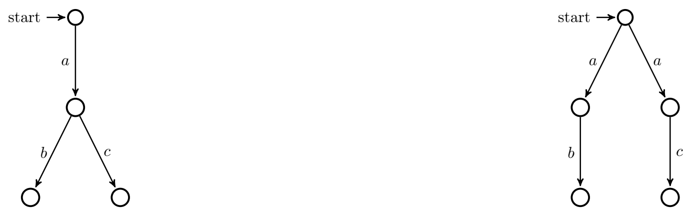


(a) Simple transition system

(b) Unfolding

Figure 4.1: Transition system and prefix of its infinite computation tree

But what difference does that make, if any? Figure 4.2 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 4.2: 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 4.2a is the "same" (equivalent, isomorphic. . .) as the one from Figure 4.2b.

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 4.3. 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.

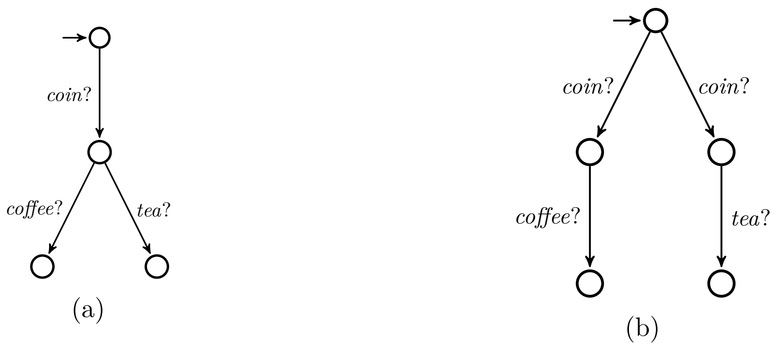


Figure 4.3: Coffee and tea vending machine

The two versions of the vending machine feel drastically different. The first from Figure 4.3a 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 4.3b, 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 4.3b, making choices on its own satisfies a specification perfectly. So even if the version of Figure 4.3a 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.

$$s \models \forall\varphi \quad \text{iff} \quad \pi \models \varphi \quad \text{for all paths } \pi \text{ starting in } s \tag{4.1}$$

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.

4.2 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 4.2.1 (CTL syntax)**.** The syntax of CTL is given by the grammar from Table 4.1, 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 4.1: 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 4.2.2 (∞). 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 4.2.3 (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

$$\forall\Box(\neg crit_1 \vee \neg crit_2) \ . \tag{4.2}$$

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 4.2.3 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 4.2.4 (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

$$(\forall\Box\forall\Diamond crit_1) \wedge (\forall\Box\forall\Diamond crit_2). \tag{4.3}$$

□

All the examples could easily have been expressed in LTL as well. In particular the example with infinitely many occurrences from Example 4.2.2 and the variation thereof, the progress Example 4.2.4. 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

$$\exists\Box\exists\Diamond p \ , \tag{4.4}$$

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

$$\exists\Box\Diamond p \tag{4.5}$$

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 (4.4) expresses *something different.*

As it turns out, infinite occurrence on some path, captured by equation (4.5), is an example of a property not expressible by CTL. The formula from equation (4.4) seem almost to express that, but it's not the same, in a subtle manner. One can show that the two formulations from equations (4.4) and (4.5) are not equivalent. That's done by a concrete example which satisfies equation (4.4) 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 4.2.5 (Response). For all behaviors, each request sooner or later will entail a response:

$$\forall\Box(\mathit{request} \rightarrow \forall\Diamond \mathit{response}). \tag{4.6}$$

□

Example 4.2.6 (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):

$$\forall\Box\exists\Diamond \mathit{start}. \tag{4.7}$$

□

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

4.2.1 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 4.2.7 (Satifaction relation)**.**

$$\begin{array}{lll} s & \models p & \textit{iff} \quad p \in V(s) \\ s & \models \neg\Phi & \textit{iff} \quad \text{not } s \models \Phi \\ s & \models \Phi_1 \land \Phi_2 & \textit{iff} \quad s \models \Phi_1 \text{ and } s \models \Phi_2 \\ s & \models \exists\varphi & \textit{iff} \quad \pi \models \varphi \text{ for some } \pi \in \mathit{paths}(s) \\ s & \models \forall\varphi & \textit{iff} \quad \pi \models \varphi \text{ for all } \pi \in \mathit{paths}(s) \end{array} \tag{4.8}$$

$$\begin{array}{lll} \pi & \models \bigcirc\Phi & \textit{iff} \quad \pi^1 \models \Phi \\ \pi & \models \Phi_1 \; U \; \Phi_2 & \textit{iff} \quad \exists j \geq 0.(\pi^j \models \Phi_2 \text{ and } \forall 0 \leq k < j.\pi^k \models \Phi_1) \end{array} \tag{4.9}$$

4.3 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.

4.3.1 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 4.3.1 (Existential normal form)**.** CTL state formilas in *existential normal form* are given by the following grammar:

$$\begin{array}{rcl} \Phi & ::= & \top \mid p \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \\ & \mid & \exists \bigcirc \Phi \mid \exists \Phi_1 \; U \; \Phi_2 \mid \exists \Box \Phi \mid \end{array} \tag{4.10}$$

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 . . .

4.3.2 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 4.3.2. Consider the transition system of Figure 4.4 and the CTL property from equation (4.11). 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.

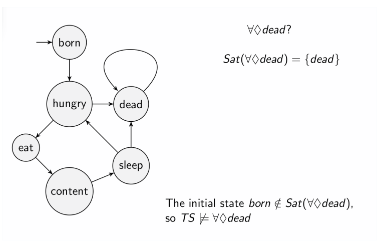

born
hungry
dead
eat
sleep
content
$\forall\Diamond dead$?
$Sat(\forall\Diamond dead) = \{dead\}$
The initial state $born \notin Sat(\forall\Diamond dead)$,
so $TS \not\models \forall\Diamond dead$

Figure 4.4: Transition system

$$\forall\Diamond dead. \tag{4.11}$$

□

Example 4.3.3. Consider again transition system of Figure 4.5. This time, let's look at the CTL property from equation (4.11):

$$\exists \bigcirc hungry \land \exists(eat \ U \ \neg dead). \tag{4.12}$$

Figure 4.5 shows the syntax tree of the formula. Additionally, the *sat*-sets are indicated for most nodes.

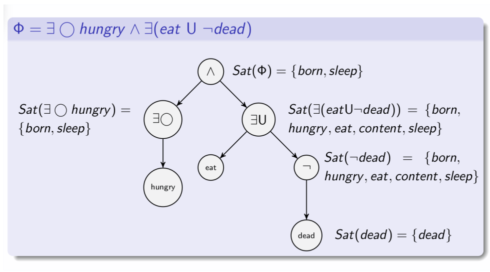

$\Phi = \exists\bigcirc hungry \land \exists(eat \ U \ \neg dead)$
$\land$
$Sat(\Phi) = \{born, sleep\}$
$Sat(\exists\bigcirc hungry) = \{born, sleep\}$
$\exists\bigcirc$
$\exists U$
$Sat(\exists(eat U \neg dead)) = \{born, hungry, eat, content, sleep\}$
eat
$\neg$
$Sat(\neg dead) = \{born, hungry, eat, content, sleep\}$
hungry
dead
$Sat(dead) = \{dead\}$

Figure 4.5: 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 4.4. 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.* □

```
input:  finite transition system $T$ and $\Phi$
output: $T \models \Phi$
--------------------------------------------------
for all $i \leq |\Phi|$ do
  for all $\Psi \in sub(\Phi)$ with $|\Psi| = i$ do
    compute $sat(\Psi)$ from $sat(\Psi')$ (* max. genuine $\Psi' \subseteq sat(\Psi)$ *)
  od
od
return $I \subseteq sat(\Phi)$
```

Listing 4.1: Basic algorithm

```
input:  finite transition system $T$ and $\Phi$
output: $T \models \Phi$
--------------------------------------------------
switch $\Phi$
    $\top$                  => return $S$
    $p$                     => return $\{s \in S \mid p \in V(s)\}$
    $\neg\Phi$              => $S \setminus sat(\Phi)$
    $\Phi_1 \wedge \Phi_2$  => return $sat(\Phi_1) \cap sat(\Phi_2)$
    $\exists \bigcirc \Phi$ => return $pre(sat(\Phi))$
    $\exists(\Phi_1\ U\ \Phi_2)$ => ``$lfp$ for $\exists\ U$''
    $\exists\Box\Phi$       => ``$gfp$ for $\exists\Box$''
```

Listing 4.2: Recursive algorithm

4.3.3 Characterization of *sat* & fixpoint calculations

Next we have to fill in the blanks in the algorithmic skeleton of Listing 4.2. 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 4.2.

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 4.2, 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 4.3.

```
$T$ := $sat(B)$;
while $pre(T) \setminus T \neq \emptyset$ do
   let $s \in pre(T) \setminus T$ ;
  $T$ := $T \cup \{s\}$;
od;
return $T$;
```

Listing 4.3: 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 4.3 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 4.3 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

$$T := T \cup \mathit{pick}_{\emptyset} T \ .$$

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

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

```
$T$ := $sat(A)$;
while    $T \neq T \cup \{pick(pre(T))\}$
  $T$ := $T \cup pick(pre(T))$;
od;
return $T$;
```

Listing 4.4: Backward exploration for $\exists\Diamond A$

At the exit of the loop, we have therefore

$$T = T \cup \{\mathit{pick}(\mathit{pre}(T))\} \ . \tag{4.13}$$

i.e., each iteration step i increases the current set T_i properly to $T_{i+1} = \{\mathit{pick}(\mathit{pre}(T))\}$, 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 (4.13).

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

$$\begin{aligned} T_0 &= A \\ T_{j+1} &= F(T_j) \qquad \text{where} \quad F(X) = \mathit{pick}_{\emptyset}(\mathit{pre}(X)) \cup X \end{aligned} \tag{4.14}$$

The T_i's from Unlike the algorithm code, the definition from equations (4.14) correspond to the value of T in the different iteration rounds, though the formulation in equation (4.14), 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 (4.13) 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.

$$A = T_0 \subset T_1 \subset T_2 \subset \ldots \subset T_k = T_{k+1} = T_{k+2} \ldots \tag{4.15}$$

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

$$A \subseteq T_i, \quad \text{and} \quad T_i \subseteq T_{j+1} \ . \tag{4.16}$$

Equation (4.15) 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 (4.16) 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 4.16, 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 $\supseteq$

$$\begin{array}{rcll} T & \supseteq & A & \\ T & \supseteq & F(T) & \quad \text{where} \quad F(X) = \mathit{pick}_{\emptyset}(\mathit{pre}(X)) \cup X \end{array} \tag{4.17}$$

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 (4.18) into a single line.

$$T \ \supseteq \ F'(T) \qquad \text{where} \quad F'(X) = \mathit{pick}_{\emptyset}(\mathit{pre}(X)) \cup X \cup A \tag{4.18}$$

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 (4.18) 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):

$$X \ = \ F'(X) \tag{4.19}$$

[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 (4.18). 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 (4.19) and (4.18). 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 (4.19) and (4.18) 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 (4.15), 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 4.2 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 (4.14), using $pick_\emptyset(pre(X))$, we simply add *all* states of $pre(X)$ in one swoop:

$$\begin{aligned} T_0 &= A \\ T_{j+1} &= F(T_j) \qquad \text{where} \quad F(X) = pre(X) \cup X \end{aligned} \tag{4.20}$$

Likeise, the code from Listing 4.4 is readily adapted to the one of Listing 4.5.

```
$T$ := $sat(B)$;
while $T \neq T \cup pre(T)$ do
  $T$ := $T \cup pre(T)$
od;
return $T$;
```

Listing 4.5: "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 4.5, if the step

$$T := T \cup pre(T) \tag{4.21}$$

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 (4.21) 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 4.1, 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 4.6: Characterization of CTL operators

```
$T$ := $sat(\Phi_2)$;
while $\{s \in sat(\Phi_1) \setminus T \mid post(s) \cap T \neq \emptyset\} \neq \emptyset$ do
   let $s \in \{s \in sat(\Phi_1) \setminus T \mid post(s) \cap T \neq \emptyset\}$;
   $T$ := $T \cup \{s\}$;
od;
```

```
return $T$;
```

Listing 4.6: Fixpoint calculation for $\exists \Phi_1 \; U \; \Phi_2$

```
$T$ := $sat(\Phi)$;
while $\{s \in T \mid post(s) \cap T = \emptyset\} \neq \emptyset$ do
    let $s \in \{s \in T \mid post(s) \cap T = \emptyset\}$;
    $T$ := $T \setminus \{s\}$;
od;
return $T$;
```

Listing 4.7: Fixpoint calculation for $\exists \Box \Phi$

4.4 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.

4.4.1 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.

$$Eval(Var) = Var \to \{0,1\} \; . \tag{4.22}$$

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

$$f : \mathit{Eval}(\mathit{Var}) \to \{0,1\} = (\mathit{Var} \to \{0,1\}) \to \{0,1\} \ . \tag{4.23}$$

The special case $\mathit{Var} = \emptyset$ 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

$$\sigma \models \varphi \ , \tag{4.24}$$

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 (4.23) 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 (4.24) 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 4.2. 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 4.4.2), 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 4.4.1. 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 (4.24), 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 corresponds 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

$$\begin{aligned}(f_1 \vee f_2)([z_1 = b_1, \ldots, z_k = b_k]) = \\ \max(f_1([z_1 = b_1, \ldots, z_m = b_m]), f_2([z_n = b_n, \ldots, z_k = b_k]))\end{aligned} \tag{4.25}$$

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 4.4.3.

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 4.4.2 (Cofactors)**.** Assume a variable set $\mathit{Var} = \{z, y_1, \ldots, y_m\}$ and let $f : (\mathit{Var} \to \{0,1\}) \to \{0,1\}$ 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

$$f\mid_{z=1}(b_1, \ldots, b_m) = f(1, b_1, \ldots, b_m) \tag{4.26}$$

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

$$f\mid_{z_1=b_1,\ldots,z_k=b_k} = (\ldots(f\mid_{z=b_1})\ldots)\mid_{z_k=b_k} \tag{4.27}$$

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

$$f\mid_{z=1} \neq f\mid_{z=0} \quad . \tag{4.28}$$

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 [31] also uses the term "symbolic analysis" in its title "A Symbolic Analysis of Relay and Switching Circuits".

Lemma 4.4.4 (Shannon expansion)**.** *Let f be a switching function for Var. Then*

$$f = (\neg z \wedge f\mid_{z=1}) \vee (z \wedge f\mid_{z=0}) \tag{4.29}$$

for all variables z from Var.

Figure 4.7

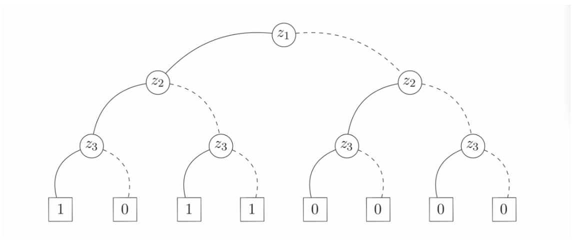


Figure 4.7: Binary decision tree for $z_1 \wedge (\neg z_2 \vee z_3)$

Definition 4.4.5 (Existential quantification)**.**

$$\exists z.f = f\mid_{z=1} \vee f\mid_{z=0} \tag{4.30}$$

4.4.2 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

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

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).

$$\Delta : ((\vec{x}, \vec{x}') \to \{0,1\}) \to \{0,1\}, \qquad \Delta(s_1, s_2[\vec{x}' \leftarrow \vec{x}]) = \begin{cases} 1 & \text{if } s_1 \to s_2 \\ 0 & \text{else} \end{cases} \ . \qquad (4.31)$$

In equation (4.31), 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 4.4.1 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.

4.4.3 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 4.2 can remain unchanged.

Let's then discuss the individual cases of the algo from Listing 4.2 one by one. For the temporal cases, see in particular also the corresponding cases from the characterization from Figure 4.6.

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

$$\exists \vec{x}'.\, \underbrace{\Delta(\vec{x}, \vec{x}')}_{s \in pre(s')} \wedge \underbrace{\chi_A[\vec{x}' \leftarrow \vec{x}]}_{s' \in A} \tag{4.32}$$

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 4.5, 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 (4.32):

$$\exists\vec{x}'.\,\underbrace{\Delta(\vec{x},\vec{x}')}_{s\in pre(s')} \wedge \underbrace{f_j[\vec{x}' \leftarrow \vec{x}]}_{s'\in T_j} \tag{4.33}$$

The treatment of th until-operator is not much harder to encode (see its characterization from Figure 4.6). The resulting code is shown in Listing 4.8.

```
$f_0(\vec{x})$ := $\chi_{A_1}(\vec{x})$;
$j$ := 0;
repeat
   $f_{j+1}(\vec{x})$ := $f_{j+1}(\vec{x}) \vee (\chi_{A_2}(\vec{x}) \wedge \exists\vec{x}'.\Delta(\vec{x},\vec{x}') \wedge f_j(\vec{x}'))$;
until $f_j(\vec{x}) = f_{j-1}(\vec{x})$;
return  $f_j(\vec{x})$.
```

Listing 4.8: Symbolic exploration $sat(\exists(A_1 \; U \; A_2))$

Finally, Listing 4.9 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 $\vee$.

```
$f_0(\vec{x})$ := $\chi_{A}(\vec{x})$;
$j$ := 0;
repeat
   $f_{j+1}(\vec{x})$ := $f_{j+1}(\vec{x}) \wedge \exists\vec{x}'.\Delta(\vec{x},\vec{x}') \wedge f_j(\vec{x}'))$;
until $f_j(\vec{x}) = f_{j-1}(\vec{x})$;
return  $f_j(\vec{x})$.
```

Listing 4.9: Symbolic exploration $sat(\exists\Box A_)$

Chapter 5 SAT-based & bounded model checking

What is it about?

Learning Targets of this Chapter

Bounded-model checking

Contents

5.1 Intro

The slides here take inspiration from the presentation [11] and the article [10]. Another good (handbook-)article is [4].

We have talked about model checking, i.e., solving the problem whether or not a model satisfies a given formula, of $M \models^? \varphi$. The focus was on the model being a transition system T, typically finite state, and the formula expressing some temporal property, written in some some temporal logic or other. In particular we have covered so far LTL and automata-based model checking and symbolic CTL model checking, using BDDs.

Definition 5.1.1 (Kripke structure)**.** A *Kripke structure* or transition system is a tuple $(S, I, \rightarrow, V)$ where S is the set of states,$I \subseteq S$ the set of initial states, $\rightarrow \subseteq \ S \times S$ the transition relation, and $V : S \rightarrow 2^P$ the *valuation* function (aka. (state) labelling function).

The conceptual *idea* of bounded model checking is quite trivial. When facing the task of verifying a system, we simply give up on exploring all of the system, but content ourselves in exploring it only up-to some point, i.e., imposing an upper bound on the exploration. In the context of LTL-model checking, the upper bound is on the *depth* of exploration, i.e., one puts a upper bound on the length on the *paths* the model checker explores.

Of course, the idea of putting an bound on the amount of effort spent in analysing a system or model can take different forms. One could for instance limit the time the model checker runs (or stop it from running if one looses patience waiting for an outcome). That would also in a way be bounded model checking thought it would not fall into a what is conventionally be called a bounded model checking technique; it would be too ad-hoc. Characteristic for classical bounded model checking is also, that model checking is presented as a SAT-solving problem, and just shutting down a process when it runs too long does not qualify for that.

But besides bounding the length of the paths, other criteria have been studied, for instance the number of context switching in parallel processes etc. We will not cover alternatives, but focus on the classical situation: exploring a system only up-to a paths of a fixed length.

Above it was said that bounded model checking gives up the hope of exploring the whole system, basically giving up the hope of system verification. One should probably formulate that less categorical.

First, in principle bounded model checking may not give that up, one could increase the bound iteratively until it spots an error or if one knows the bound is big enough. For finite-state systems such may be possible. But on the other hand, even if possible, it is somehow not too natural and a bit against the spirit of bounded model checking, which focuses more on spotting errors within exactly specified bounds. Bounded model checking shares this spirit-ual focus of finding errors with testing (focusing on a finite set of test cases) and with run-time verification (looking just at one run).

Secondly, and probably more importantly: that unbounded model checking allows full and absolute verification of correct systems resp. finding bugs in faulty ones is also more a fiction. In the introductory part we discussed why that is is a naive view in practice. One is that model checking typically works with *models* of the real system and with certain abstractions. That alone may result in that some errors are overlooked (and/or lead to the detection of spurious errors, which are errors caused by the abstraction but absent in the more detailed system). And even in when abstracting away from certain details, the state space may still be infinite or too big. So, when doing “unbounded” model checking in such a situation, the analysis may find its natural bounds by the maximum memory usage, for instance, or the practically acceptable running time.

Back from the more philosophical remarks, back to the technical realization of the general idea of exploring all paths up to a fixed length of k. An important aspect of the approach we have encountered already in connection with CTL-model checking. In a quite different form, thought, but there is the general commonality of a

> symbolic treatment via a propositional encoding for the purpose of the exploration.

In the context of CTL model checking we have looked at how different propositional constructs, ultimatly how propositional constraints can be represented and worked with. The propositional encoding back then was to capture sets of states, and the transition system etc. At some level, those entities where described by (quantified) boolean formulas. The formulic constraints were interpreted as boolean (swiching) functions, ultimatelty represented as ROBDDs.

Here, we are concerned representing finite paths through a given transition system (not (the evolution of) sets of states as in the CTL case). That means, the boolean formulas look different, but the idea is similar.

In the CTL case, which does a backwards, breadth-first exploration, a crucial ingredient was to capture the set of predecessor states for a given set A of states. That set $spre(A)$ was captured by a formula that relied on (propositional) existential quantification and relied on the propositional representation of the states and the transition relation $\rightarrow$ of the transition system.

Following similar ideas, it's plausible that one can represent a *finite* path of the form

$$s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_m \ . \tag{5.1}$$

As mentioned earlier, in its basic form, bounded model checking focuses in finding errors, namely errors that can be spotted with a finite distance, say k, from the initial states. One could, if one prefers that, spot errors with a finite distance of a different point in the program, or errors doing the same thing backwards, i.e., errors at most k-step *before* a given point, maybe before the end of the program. Those variations are inessential for the idea, and let's discuss it by assuming we do it forward, and start at an initial state, and let's assume there's only one of those, which is a common set-up anyway.

Encoding the path from equation (5.1) as SAT problem descibes all paths of finite length m, i.e., the solutions to that SAT-problem are those paths. Indirectly that descibes also all states s_m that are at a distance m away from the initial state s_0.

Now, if we see bounded model checking as a form of bug-hunting with a bound how deep to look for a bug, then of course it's not good enough to encode equation (5.1). Assume that we are after a safety property, a simple invariance $\Box A'$, with A' propositional. Seen as a state formula, it means we want to establish resp. refute $\forall \Box A'$. To refute it means to establish an existential state formula of the form

$$\exists \Diamond A \tag{5.2}$$

(with $A = \neg A'$). Taking the path specification from equation (5.1), i.e., its SAT encoding and with proposition A at hand, one can formulate that

$$s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_m \ \wedge s_m \text{ satifies } A \tag{5.3}$$

The states are best understood as variables. As far as s_0 is concerned, there may only be one state that qualifies, under the assumption that the model has only one initial state. Checking equation (5.3) for a satifying solution mean, looking for states $s_0, \ldots s_m$ such that s_m satisfies the proposition A. To say it differently but equivalently, the s_i are best seen as exitentially quantified. At any rate, in this way, equation (5.3) does not represent a particular path with concrete states s_0 to s_m, but *all paths of length* m that start at the initial state and with some end-state s_m that satisfies A. That *symbolic* treatment is similar to what we have seen in CTL model checking where one crucial trick was to represent the *sets* of states and the set of predecessor states of a given set in a propositional manner.

So we know that we can capture paths of length m and a witness to $\exists \Diamond A$ from equation (5.2). Let's abbreviate the corresponding formula from equation (5.3) by

$$\exists \Diamond_m A \tag{5.4}$$

with m fixed. Now, we are looking for witnesses at an arbitrary distances, i.e., what we are actually after is something like $\exists m : \mathbb{N}.\exists \Diamond_m$. That's not a propositional formula. The inner existential quantifier is ok, we made use of existential quantifications also in the encoding of $pre(A)$ for CTL model checking. It quantifies over finite entities, resp. their propositional representation. So, that can be done propistionally, in so-called quantified boolean or quantified propositional logics, which is ultimately not much different than propositional logic without that operation. In particular it's about *finite* structures.

But the outer quantification $\exists m : \mathbb{N}$, that's a different story, that's no longer propositional. Another way to see it, to look for a witness means to solve the sat-problem $\exists\Diamond A$ is to solve

$$\exists\Diamond_0 A \lor \exists\Diamond_1 A \lor \exists\Diamond_2 A \lor \ldots \tag{5.5}$$

That's just a different way of expression the existential quantification over $\mathbb{N}$. And of course neither that is propositionally allowed. Disjunctions, of course, are allowed, but equation (5.5) makes use of an *inifinite* disjunction, which one cannot do.

But that's where the boundedness of bounded model checking comes in. The sat-constraint $\exists\Diamond A$ captures standard, unbounded model checking, unfortunately in a non-propositinal way. But if we restrict ourselves to find a witness only up-to a fixed depth k, then the setting becomes finite. Instead of the existential quantification over all natural numbers, one has to deal only with a quantification $\exists m \in \{0, \ldots, k\}$, resp. the infinite disjunction from equation (5.5) is now defused to a propositional finite disjunction

$$\bigvee_{m=0}^{k} \exists\Diamond_m A \tag{5.6}$$

5.1.1 What else need to be taken care of?

We sketched the core idea behind sat-based bounded model checking. In the short remarks, we simplified the situation slightly, basically by focusing on finding a encoding for a the problem $\exists\Diamond_{m\leq k} A$ and looking for a satisfying witness for that. That corresponds to refuting the simple invariance property $\forall\Box\neg A$, at least in a bounded manner.

However, we want apply the approach not just to formulas $\forall\Box A'$, which is the simplest form of a safety property, but to all of LTL.

Generally, of course, that's not possible: one cannot hope to verify or refute a property by looking only at path of finite length. At least not the the way we told the story. Additional considerations would come it, which are somehow orthogonal, and thus not in the spirit of the story of bounded model checking as sat problem over finite paths. We leave those extra considerations out in this discussion (for now, or completely).

That actually also applies to the situation from above, when reasoning about $\forall\Box A'$ in a bounded manner. We can *refute* that property by finding a finite counter example. But one cannot *verify* it in the described manner (without additional tricks), i.e., we cannot *decide* the logical status of that formula.

The above formula is the simplest form of safety property. But the same can be said about liveness properties, neither their status can be decided. The situation is somehow dual, however. Let's take as property to verify $\forall\Diamond\varphi$, which means $\exists\Box\neg\varphi$ for refutation resp. as sat problem. In a bounded form, it's about sat-solving

$$\exists\Box_{\leq k}\neg\varphi\ . \tag{5.7}$$

If there exists a solution to the constraint, i.e., a path as witness for that formula, resp. a counter-example for the original formula in its bounded form $\forall\Diamond_{\leq k}\varphi$, then we have a finite witness-path where always $\neg\varphi$. But having established the constraint (5.7) does not imply that also $\exists\Box\neg\varphi$, in other words, we cannot *dis*-prove $\forall\Diamond\varphi$. One the other hand, if the constraint (5.7) is not satisfiable, it means $\forall\Diamond_{\leq k}\varphi$ is the case, and that means also unboundedly $\forall\Diamond\varphi$ where it holds.

So the safety property could be disproved in a bounded manner (if it actually does not hold), and one gets counter-example for it. The sample liveness propert could be proved (if it actually holds).

In some way, that's not posible. There are properties that cannot be refuted in that manner (nor can they be verified that way). At least not the the way we told the story. Additional considerations would come it, which are somehow orthogonal, and thus not in the spirit of the story of bounded model checking as sat problem over finite paths. We leave those extra considerations out in this discussion (for now, or completely).

Without any extra tricks, it's clear that one cannot verify a problem $\forall\Diamond\varphi$, and one cannot refute it, i.e. find a counter-example as a witness for the negation $\exists\Box\neg\varphi$. Looking only at paths up-to length k (and without extra tricks) gives **no** information about the status of $\forall\Diamond\varphi$. Looking at a single path (for the sake of argument) and assuming that φ does not hold up to k, it's clear that without looking beyond the bound we cannot know

But we are of course not doing model checking on one path, but all paths up to a given bound. If, by some lucky chance, φ would become true within the explored k-horizon *on all bounded paths*, then one could conclude that $\forall\Diamond\varphi$ actually holds. But we are not getting hold of all such paths. We are doing sat-solving and are working with $\exists\Box\neg\varphi$. The information we get is either

$$\exists\Box_{\leq k}\neg\varphi \quad \text{or} \quad \forall\Diamond_{\leq k}\varphi$$

and we only get to know that there exists bounded path where $\Box\neg\varphi$ or else not.

If there does not exist such bounded path where $\neg\varphi$ is always true, it means $\forall\Box_{\leq k}\varphi$forit has not information about the situation beyond the bound, and it does not help. If, on the other hand, there exists a bounded path where $\Box\varphi$ holds, it carries no information eather

5.2 Bounded semantics

We have covered the standard semantics of LTL earlier, interpreting LTL formulas over *infinite* paths. The interpretation of formulas over a transition system being a derived notion.

In the CTL-part, in particular in the model checking part, we restricted our interest to formulas in a particular normal form, it was CTL-formulas in ENF, existential normal form. We also mentioned that other normal forms exists, for CTL or other logics. We use a quite common such normal form here for LTL and bounded model checking, namely **negation normal form, NFF**.

In such a form, one restricts the use of negation in that only atomic propositions or propositional variables are allowed to occur negated and negation cannot be used on compound formulas. To compensate for that, we need both $\wedge$ *and* $\vee$, both $\Diamond$ and $\Box$ etc.

In the presentation (following [4]), we leave out, however, the until-operator and the other binary temporal operators. Not that it would be complex, but the presentation is slightly simpler.

An initialzed path satisfying φ is called a *witness* for $\exists\varphi$.

or counter example, remember *refutation*

π of a transition system T

witness of φ in T is a path starting in an initial states and that satisfies φ.

We assume the formulas in *negation normal form.*

Looking for witnesses for formulas of the form $\exists\Diamond\varphi$. In other words the approach can deal straightforwardly with finding *counter-examples* for formulas of the form $\forall\Box\varphi'$, generally counter-examples to *safety properties.*

In the dual case of looking for counter-examples for liveness properties, i.e., looking for witnesses of for instance formulas of the form $\exists\Box\varphi$, things get more involved. It's intuitively clear that a finite prefix of an infinite path cannot be used as witness for

$$\exists\Box\varphi\ . \tag{5.8}$$

Unless one adds one additional clever idea, a central one for bounded model checking in the form discussed here.

The idea is to take into account **looping behavior.** Consider Figure 5.1, depicting a loop or a lasso. Both words are used for that, in the original publication it was called loop, though in the handbook article, the word "lasso" is used. I prefer lasso, and a lasso consists of two parts, the *stem*, in the picture going from s_0 to s_l and the loop part, the part depicted as cycle at the end.

The picture is a visualization of lasso or loop, but it's (also) understood as a *path*, i.e., an infinite sequence of states (see Definition 5.2.1). The infinite path it respesents has a finite prefix, the stem, and then an infinite repetition of the looping part (see also equation (5.9)).

Even if, technically, a lasso is an infinite path, it can be *finitely represented,* as in Figure 5.1 and it also can detected looking only at a finite prefix. Having reached state s_k and detecting that one can continue to a state one has seen before, state s_l in the picture,

$$\begin{tikzcd}
\bullet_{s_0} \arrow[r] & \bullet \arrow[r] & \bullet_{s_l} \arrow[r] & \bullet \arrow[r] & \bullet_{s_i} \arrow[r] & \bullet_{s_k} \arrow[llll, bend left]
\end{tikzcd}$$

Figure 5.1: A lasso

shows that there is a cycle in the transition system. It shows also that the prefix *can* be extended to a infinite path by repeating the cycle forever. That means that in a situation with a lasso, a finite prefix can actually serve as a witness for the from equation (5.8). Note that it's important that in the approach, looking for witnesses, we are dealing with existentially quantified formulas $\exists\varphi$, so the detection of a loop shows there exists an infinite path as witness. So a finite sequence of $k+1$-states represents (the existance of) a lasso, if continues, and we use the term lasso also for that finite representation consisting of $s_0 \ldots s_k$, with the last state s_k as the one with the back-transition to some earlier s_l.

Definition 5.2.1 (Lasso)**.** Assume $l \leq k$. A path π is a (k,l)-*lasso* if $\pi_k \to \pi_l$ and

$$\pi = u \cdot v^{\omega} \tag{5.9}$$

with

$$u = \pi_0 \ldots \pi_{l-1} \quad \text{and} \quad v = \pi_l \ldots \pi_k$$

A path π is a k-*lasso* if there exists an l with $0 \leq l \leq k$ s.t. π is a (k,l)-lasso

Of course, not all finite paths are k-lassos, for instance if all states are different. Technically, that's not the only reason, why a path of length $k+1$ is no lasso. The definition as given insists that, to be a lasso, the last state s_k loops back, if an earlier state s_m with $m < k$ is repeated, then technically that's not a loop. But its prefix $s_0 s_1 \ldots s_m$ would be. In the larger picture, it's not so important, as the approach works with prefixes of paths of all length up-to a pre-defined upper bound say k, i.e., the lasso at m will be detected and taken into account.

At any rate, some finite path of length $k+1$ are lassos (or contain lassos in their prefix) and some not. One could increase k, increasing also the chance to find some lassos. Having a path with a lasso is a good thing, because, as explained, in this favorable situation, we can hope to use it as witness for an $\Box$-style formula (or a counter-example for an original liveness property). Of course not all lassos are factually a witness for such a formula, the φ must also hold at all positions.

But for non-loop finite paths, the situation is worse. They can serve as witness for $\exists\Diamond\varphi$ formulas; for that kind of properties the lassos play no role. But without loop, they don't contain information for $\forall\Box\varphi$-formulas.

When we will define the so-called *bounded* semantics for LTL, i.e., when a finite path satisfies an LTL formula, we distinguish the two situations: paths which are lassos and paths which are not. In both cases, sometimes the finite path can serve at a witness. In this case the semantic will report back that this is the case.

But there will be cases where the finite prefix contains not enough information to serve as a witness (and to serve as counter-example for the original $\forall$-formula). We discussed the case that obviously non-lasso prefixes can never serve as witness for $\Box$-formulas. But the same problem, the fact that a finite prefix does not contain enough information, also applies to $\Diamond$-formulas. If we want a witness for $\Diamond\varphi$, but the finite prefix contains no occurrence of φ, we likewise don't know.

The question is

> What should the semantics give as an answer is such "don't know'-situations?

One possibility could be to work not with a binary logic with true and false, but perhaps with true, false, and "don't know". Some approaches for run-time verification use a 3-valued version of LTL for that.[1] But here we do something else. We stick to 2 boolean values, but in case of not knowing, the bounded semantics reacts pessimistically and gives *false* as judgement. The emphasis is on finding a witness for a formula (resp. a counter-example for the non-negated property), and if the length of exploration cannot be used as such a witness, then, with the current bound, there is no witness (yet) and the finite path does not satify the formula in the bounded semantics.

Let's now define the bounded semantics, and let's do that in the form of a satisfaction relation $\models$, as before. The bounded version is written $\models_k$, where k is the bound up-to which we consider the path. As mentioned, the definition is given in two "variants", one for paths which are in the form of a lasso, and one for finite paths where that is not the case. In the latter case we will also work with a refined version of $\models_k$, written $\models_k^i$, see below. Let's start with the *lasso-case*.

Definition 5.2.2 (Bounded semantics: with lasso)**.** Let π be a k-loop. A formula φ is *valid* along π *with bound* k, written

$$\pi \models_k \varphi \quad \text{iff} \quad \pi \models \varphi \ .$$

The definition is quite trivial, it's the original, unchanged semantics $\models$. In the presence of a k-loop and with the bound at k, we have all the information we need. The loop case is exact, non approximative.

On the other hand, that's only the case if we consider the bounded semantics on one path (as the definition does). In the larger conceptual picture, we are exploring a transition system, and we are exploring paths up-to a bound of k. Thinking in particular about looking for a witness for $\exists\Box\varphi$, in the positive case, namely that the formula holds, the semantic is exact also in the larger picture of exploring the transition system. In the negative outcome, however, when the lasso-path does *not* satisfy the formula, of course it does not mean ...

[1] Actually also 4-valued version. It's 4 values in that they make use of 2 version of "unknown", somehow like "unknown, but it looks good", and "unknown, but it looks bad", a "tendency" towards true resp false. This is an informal description and not 100% accurate, and there are technical reasons which speak for 4 values instead of 3.

Definition 5.2.3 (Bounded semantics: without lasso)**.**

$$\begin{array}{lll}
\pi \models_k^i p & \text{iff} & p \in L(\pi_i) \\
\pi \models_k^i \neg p & \text{iff} & p \notin L(\pi_i) \\
\pi \models_k^i \varphi_1 \land \varphi_2 & \text{iff} & \pi \models_k^i \varphi_1 \quad \text{and} \quad \pi \models_k^i \varphi_2 \\
\pi \models_k^i \varphi_1 \lor \varphi_2 & \text{iff} & \pi \models_k^i \varphi_1 \quad \text{or} \quad \pi \models_k^i \varphi_2 \\
 & & \\
\pi \models_k^i \Box\varphi & & \text{is always false} \\
\pi \models_k^i \Diamond\varphi & \text{iff} & \exists j. i \leq j \leq k. \quad \pi \models_k^j \varphi \\
\pi \models_k^i \bigcirc\varphi & \text{iff} & i < k \quad \text{and} \quad \pi \models_k^{i+1} \varphi \\
\pi \models_k^i \varphi_1 \; U \; \varphi_2 & \text{iff} & \exists j, i \leq j \leq k. \pi \models_k^j \varphi_2 \text{ and } \forall n, i \leq n < j. \pi \models_k^n \varphi_1 \\
\pi \models_k^i \varphi_1 \; R \; \varphi_2 & \text{iff} & \exists j, i \leq j \leq k. \pi \models_k^j \varphi_1 \text{ and } \forall n, i \leq n < j. \pi \models_k^n \varphi_2
\end{array}$$

Now that we have established the bounded semantics, we look into it connection to the original one. Basically, in the limit, both semantics coincide. The bounded semantics is an approximation of the real semantics, in the sense that in case of a positive answer in the bounded semantics, that carries over to the real, unbounded semantics. In other words, the bounded semantics is a *sound* approximation of the unbounded semantics, and that for every bound k. See Lemma 5.2.4.

Lemma 5.2.4 (Soundness (per path))**.**

$$\pi \models_k \varphi \quad \textit{implies} \quad \pi \models \varphi$$

That's not surprising, and was easy to achieve: the bounded semantics in case of not having enough information, stipulated that the formula does *not* hold. The reverse direction, completeness is harder, and it means that ultimately, if a formula for existential LTL holds in a transition system, it also holds for the bounded semantics, if only the k is large enough. See Lemma 5.2.5.

Lemma 5.2.5 (Completeness (for TSs/KSs))**.**

$$S \models \exists\varphi \quad \textit{implies} \quad S \models_k \exists\varphi \quad \textit{for some } k \geq 0$$

Both results together give the following

Theorem 5.2.6.

$$S \models \exists\varphi \quad \textit{iff} \quad S \models_k \exists\varphi \quad \textit{for some } k \geq 0$$

So, to use a bounded model checker, one has pick a large enough k and the tool gives the correct answer. The k may not be known, so one could guess one. Or better, model checker could increase the k automaticall in case an attempt is unsuccessful, hoping that one finds an positive answer at some point and before running out of resources.

Besides that one may well run out of resouces before one gets a positive answer, banking in on soundness, the case where the $\exists\varphi$ does *not* hold is principally more problematic.

Completeness from Lemma 5.2.5 states that in this case, one need to check $\models_k$ **for all** $k \geq 0$**!** That's of course not doable.

The existential LTL formula $\exists\varphi$ is typically the negation of a standard $\forall$-formula, one tries to refute. That's why we said, bounded model checking focuses on finding bugs (i.e., refuting the original formula), but *not* on verification: one cannot check the bounded model checking problem for all k.

There are things one can do about that. For instance, in a finite-state system, if one knows the size of the transition system (more precisely its diameter), one can use that to know when there's no use in looking for larger k's, and if until then one has not found a witness (a counter-example to the original $\forall$-formula), one is sure there is none. But we don't into that further.

5.3 Propositional encoding

Next we will present how a bounded model checking problem can be addressed by sat-solving by showing an encoding of such problems. We show the encoding as originally proposed, which is a pretty straightforward translation of the bounded LTL semantics and of course the transition system into propositional logics. Indeed, afterwards, other encodings and representation have been proposed, for instance some working with Büchi-automata or other techniques that lead to more efficient implementations. See for instance [23] which is also covered in [4]. Also explicit-state, non-symbolic method have been explored for bounded model checking. See for instance for some discussion of different techniques [9].

Here, as said, we stick to the original proposal of the encoding. Some of that we basically know already from symbolic model checking of CTL, namely how to encode the transition system. Here we don't need to encode the set of predecessors *pre*, but finite paths through the system, but the priciples are the same. Talking about CTL model checking: in that part we continued to explain how the encoding can be understood as boolean switching functions, and how those can be canonically represented as BDDs. That one popular way in CTL model checking.

Here we don't do that, we just show the formulaic representation as boolean formula. How to handle those we leave to a SAT-solver, which may not be based on BDDs; most SAT solvers are using variations of David-Putnam's procedure (DP).

Goal: $[\![T, \varphi]\!]_k$ is *satisfiable* by some π iff π is a witness for φ

The encoding will be done in three separate parts, one dealing with the (finite paths) of the given transition system. Secondly, capturing the loop or lasso condition, and finally, representing the formula φ, resp. its bounded satisfaction relation. The latter part is split into the semantics dealing with paths with a lasso, and those without (as it was defined earlier)

Encoding the transition system

Let's start with the transition system, resp. bounded paths through it. Let T be a transition system, then

$$[\![T]\!]_k \triangleq I(s_0) \wedge \bigwedge_{i=0}^{k-1} s_i \to s_{i+1} \tag{5.10}$$

Loop condition

Remember the concept of (k,l)-loop or lasso and of k-lasso from Definition 5.2.1 (remember also Figure 5.1). Both are readily encodable. ${}_lL_k$, representing a (k,l)-loop is a simple abbreviation for the fact that there is an edge in the transition system between the to involved states.

$$ {}_lL_k \triangleq s_k \to s_l $$

With that the k-lasso is given as a finite disjunction (or bounded existential quantification) as follow:

Definition 5.3.1 (Loop condition)**.**

$$L_k \triangleq \bigvee_{l=0}^{k} {}_lL_k$$

Encoding (the semantics of) φ

Earlier, the bounded semantics was defined using the relation $\models$. Here we capture the semantics as the set of states where the formula holds, i.e., for encode $[\![\_]\!]$.

Lasso-case: bounded semantics for a (k,l)-loop Encoding the propositional part as propositnal formula is, of course, trivial

$$\begin{array}{rcl}
{}_l[\![p]\!]_k^i & \triangleq & p(s_i) \\
{}_l[\![\neg p]\!]_k^i & \triangleq & \neg p(s_i) \\
{}_l[\![\varphi_1 \wedge \varphi_2]\!]_k^i & \triangleq & {}_l[\![\varphi_1]\!]_k^i \wedge {}_l[\![\varphi_2]\!]_k^i \\
{}_l[\![\varphi_1 \vee \varphi_2]\!]_k^i & \triangleq & {}_l[\![\varphi_1]\!]_k^i \vee {}_l[\![\varphi_2]\!]_k^i
\end{array}$$

The temporal part is a but more interesting. To do that, we also need to "define" the successor of a state, resp. we need to convince ourselves that one can propositionally refer to the successor of a state in a path with a loop. We use $succ(i)$ to refer to the successor of i in a (k,l)-loop as

$$succ(i) = \begin{cases} i+1 & \text{for } i < k \\ l & \text{for } k \end{cases} \tag{5.11}$$

$$\begin{array}{rcl}
{}_l[\![\Box\varphi]\!]^i_k & \triangleq & {}_l[\![\varphi]\!]^i_k \wedge {}_l[\![\Box\varphi]\!]^{succ(i)}_k \\
{}_l[\![\Diamond\varphi]\!]^i_k & \triangleq & {}_l[\![\varphi]\!]^i_k \vee {}_l[\![\Diamond\varphi]\!]^{succ(i)}_k \\
{}_l[\![\bigcirc\varphi]\!]^i_k & \triangleq & {}_l[\![\varphi]\!]^{succ(i)}_k \\
{}_l[\![\varphi_1 \ U \ \varphi_2]\!]^i_k & \triangleq & {}_l[\![\varphi_1]\!]^i_k \vee {}_l[\![\varphi_1 \ U \ \varphi_2]\!]^{succ(i)}_k \\
{}_l[\![\varphi_1 \ R \ \varphi_2]\!]^i_k & \triangleq & {}_l[\![\varphi_2]\!]^i_k \wedge {}_l[\![\varphi_1 \ R \ \varphi_2]\!]^{succ(i)}_k
\end{array}$$

Translation for paths without a loop The translation follows the same principles (the index l is not needed obviously here). And instead of the more "complex" $succ(i)$ from before, we can simply use $i+1$, otherwise: the definition stays "the same"

The propositional part is boring again (and we don'd show it):

For the temporal case, we can make, we have $\forall i \leq k$:

$$\begin{array}{rcl}
[\![\Box\varphi]\!]^i_k & \triangleq & [\![\varphi]\!]^i_k \wedge [\![\Box\varphi]\!]^{i+1}_k \\
[\![\Diamond\varphi]\!]^i_k & \triangleq & [\![\varphi]\!]^i_k \vee [\![\Diamond\varphi]\!]^{i+1}_k \\
[\![\bigcirc\varphi]\!]^i_k & \triangleq & [\![\varphi]\!]^{i+1}_k \\
[\![\varphi_1 \ U \ \varphi_2]\!]^i_k & \triangleq & [\![\varphi_1]\!]^i_k \vee [\![\varphi_1 \ U \ \varphi_2]\!]^{i+1}_k \\
[\![\varphi_1 \ R \ \varphi_2]\!]^i_k & \triangleq & [\![\varphi_2]\!]^i_k \wedge [\![\varphi_1 \ R \ \varphi_2]\!]^{i+1}_k
\end{array}$$

We can see it as induction case. The "induction" goes backwards insofar the situation of i is defined in terms of $i+1$. So the base case is the one at the end, namely for $k+1$.

$$[\![\varphi]\!]^{k+1}_k \triangleq \mathit{false} \tag{5.12}$$

Putting it together Finally, we can put the three ingredients together, the encoding of the paths, the loop condition, and the encoding of the formula.

$$\begin{array}{rl}
[\![S,\varphi]\!]_k \triangleq & [\![S]\!]_k \wedge \\
& (\ (\neg L_k \wedge [\![\varphi]\!]^0_k) \\
& \quad \vee \quad (\bigvee_{l=0}^{k} ({}_l L_k \wedge \ {}_l[\![\varphi]\!]^0_k) \quad)
\end{array} \tag{5.13}$$

Theorem 5.3.2.

$$[\![S,\varphi]\!]_k \ \textit{satisfiable} \quad \textit{iff} \quad S \models_k \exists\varphi \ .$$

Chapter 6 Partial-order reduction

What is it about?

Learning Targets of this Chapter

The chapter gives an introduction to *partial order reduction*, an important optimization technique to avoid or at least mitigate the state-space explosion problem.

Contents

6.1 Introduction

The material here is based on Chapter 10 from the book [12] or the handbook article [28]. [2] does not cover partial-order reduction.

A fundamental limitation in model checking is the *state-space explosion problem*. Model checking is basically intractable, i.e. it suffers from a combinatorial explosion which renders the state space exponential in the problem or system size.

All analyses based on "exploration" or "searching" suffer from the fact that problems become unmanagable when confronted with realistic problems. That's also true for other approaches like SAT/SMT-solving, and there is no lack of intractable problems in all kinds of fields. If we look at (explicit-state) model checking very naively, and perhaps even focus on only very simple problems like checking $\Box\varphi$ ("always φ"), the model checking problem phrased like this and as such seems not really untractable (complexity-wise). It's nothing else than graph search, checking "reachability" of a state that violates the property φ. Searching through a graph is *tractable* (it has *linear* complexity, measured in the size of the graph, i.e., linear in the number of nodes and edges). So that's far from "untractable".

In model checking, it's the size of the graph, that causes problems. That's typically exponential in the description of the program. In model checking one is often interested in temporal properties of reactive, concurrent programs, consisting of more than one process or thread running in parallel. Typically, the size of the global transition system "explodes" when increasing the number of processes, due to the many interleavings of the different local process behaviours one needs to explore.

Of course, there may be other sources that make a raw state exploration unmanagable. If the problem depends on data input (like inputting numbers), the the size of the problem increases exponentially with the "size" of the input data. If one uses integers with only one byte length, one already has to take 2^8 inputs into account. Normally, of course, one has (immensely) larger data to deal with, and perhaps not just with one input, but repeated input in a reactive system. Those kind of data dependence quickly goes out of hand. That is also a form of "state space explosion problem", but mostly, when talking about the state-space explosion problem for model checking, one means the state space explosion due to different *interleavings* of concurrent processes. Dealing with data is not the strong suit of traditional model checkers, so it sometime better to deal with data with different techniques, and/or to ignore the data This means to abstract away from data (that's also known as *data abstraction*) and let the model checker focus on the part of the problem it is better suited, the reactive behavior and temporal properties.

Partial-order reduction is a technique to reduce the explored state space by avoiding irrelevant interleavings

One last word about "complexity": Before we said that model checking is linear in the size of the transition system. That's of course an oversimplification, insofar that the "size" of the formula plays a role as well. For instance, in the section about the μ-calculus it was hinted at the the alternation-depth is connected to the complexity of the model checking problem in that contaxt. For model checking LTL, the time complexity is actually exponential in the size of the formula. Normally, that's not referred to as state space explosion and also in practice, the size of the formula is not the limiting factor. Also, if one has many properties to check, which can be seen as a big conjuction, one can check the individual properties one by one.

Battling the state space explosion

As it's such major road block, it's clear that many different techniques have be proposed, investigated, and implemented to address it. An incomplete and somewhat unordered list symbolic techniques, BDDs, abstraction, compositional approaches, symmetry reduction, special data representations, parallelization of model checking, the use of "compiler optimizations" on the model (like slicing, live variable analysis . . .). And here we are doing *partial order reduction.*

"Asynchronous" systems and interleaving Partial-order reduction is most effective in asynchonous systems. The distinction is for systems with different parts working in parallel or concurrently, and one can make that distinction for hard- or software. In HW, synchronous behavior can be achieved by a global hardware clock, that forces different components to work in lock step. The global clock is used to *synchronize* the different parts. Also in software, synchronous behavior has its place (one could have protocols simulating or realizing a global clock) there are also so-called *synchronous languages*, programming languages based on a synchronous execution model, they are often used to model and describe HW, resp. software running on top of synchronous HW.

Concurrent software and programs, though, more typically behave *asynchronously*, i.e., without assuming a global clock. A good illustration are different independent processes inside an operating system, say on a single processor. The operation system juggles the different processes via a *scheduler*. The scheduler allocates "time slices" to processes, letting a process run for a while, until it's the turn of another process (preemptive scheduling).

In a mono-processor, it's one process at a time, and the scheduler **interleaves** the steps of different processes. That's a prototypical asynchronous picture.

Of course, often processes or threads etc. don't run in a completely independent or "a-synchronous" manner. To allow coordination and communication (and perhaps to help the scheduler), there are different ways of *synchronization* and constructs for synchronization purposes (locks, fences, semaphores, barriers, channels ...). Very abstractly, synchronization just means to *restrict* the completely free and independent execution. Even if processes coordinate their actions using various means of synchronization, one still speaks of *asynchronous* parallelism. If one would go so far in tie the processes together by using a sequence of global barriers, where each processes takes part in, then that very restrictive mode of synchronization would effectively correspond to having a global clock and synchronous behavior.

The two ways of compose two automata "in parallel" reflected those two ends of the spectrum: completely asynchronous and completely synchronous. (The definition was done for Büchi-automata, but the specifics of (Büchi-)acceptence are an orthogonal issue that have to do with the specific "logical" needs we had for those automata (representing LTL). The synchronous-vs.-asynchronous composition is independent from those details.

Where does the name come from? The name of the technique seems to promise reductions based on "partial order". We'll see about the reductions of the state space later, but why "partial order"?

A partial order (or partial order relation) is a binary relation which is *reflexive*, *transitive*, and *anti-symmetric*

What's the connection? The short story is maybe the following: exploring the state space involves exploring different interleavings of steps of different processes. Often that means one can do steps either in *one order* in one exploration, and in *reversed* order in a *alternative* exploration (and the whole trick will be to figure out situations when the exploration of the alternative order is not needed). One will not figure out precisely *all* situations where one can leave out alternatives, that would be too costly. So, one conservatively overapproximate it: when in doubt with the available information, better explore it.

It's of course not *always* the case that one can reorder steps into an alternative order. Steps within the same process might well be executed in the order written down; likewise, synchronization and communication may enforce that steps are done in one particular order or at least that they cannot be freely shuffled around (that's, in a way, the whole point of synchronization). Anyway, one may therefore see the actions or steps as *partially ordered*, at least when considering the behavior of the system as a whole. Focusing on one run or path, of course, presents one particular schedule and the steps in that run appear in a *linear* or *total order*. In one particular run, it's not represented, if two events are ordered by necessity (one is the cause of the other for instance) or whether the order is accidental.

That's the short story. Based on ideas as discussed, people proposed ways to describe concurrent behavior different from the *interleaving* picture, but based on *partial orders*. Those kind of styles of semantics are connected to **true concurrency** semantics, to distinguish them from "interleaving semantics" (which thereby could be called a "fake concurrency" semantics...). There is a point to it, though. Remember the informal discussion of asynchronous processes and interleaving, referring to scheduling processes on a single-core processor. There, clearly concurrency is an illusion maintained by the operating system's scheduler, that juggles the different processes so fast that, for the human, they appear to be concurrent, whereas "in reality", there is at most one process actually executed at a time. Two things being concurrent, in that picture, is just a different way of saying, they can occur in either order. True concurrency semantics takes a diffent point of view, seeing concurrency as something different from just unordered.

As simple litmus test: A semantics that considers $a \parallel b$ as equivalent to $ab + ba$ is an interleaving semantics, if the two "systems" are different, it's a true concurrency interpretation (details may apply).

For instance, Petri-nets is a quite old "true concurrency" model (they exist also in many flavors, and there are other true concurrency models as well). True concurrency models make use of partial orders (and perhaps other relations as well), but we don't go into true concurrency models.

Independent from the true-vs-"fake" concurrency question: there is a connection between partial order semantics and semantics based on arbitrary interleavings. It's a known mathematical fact that every partial order can be linearized (i.e., turned into a total order), and more generally, that a partial order is equivalent to the set of all its linearizations. The first statement, that partial orders are linearizable, may be known from the 2000-level course "algorithms and data structures". In that course, a straightforward solution to the problem is presented known as "Dijkstra's algorithm".

POR here takes as starting point sets of executions or runs, which are linearizations. It does not take as a starting point a partial-order or a true concurrency semantics. While connections between partial orders and linearizations are easy, well-known, and hold generally, i.e., for all partial orders, they are more an inspiration than a technical basis for partial order reduction here. Nailing down a concrete partial order semantics for *concrete* situations in an asynchronous setting with specific synchronization constructs is not so easy. It's much easier to specify what a program can do for the next step; that leads to an operational semantics which also specifies all possible runs of a program. *Implicitly,* that also contains all alternative runs, so one could say (based on the mentioned "math fact") that somehow indirectly one may view it as that it describes a "partial order" between the steps of the behavior of the program. But it's, as said "implicit" and for the behaviors per program. But it's far from easy to start upfront with a partial-order based semantics for all programs.

POR therefore is not based directly on an explicit partial order semantics. It does not even strive to reconstruct fully the underlying partial order that is hidden in the set of all interleavings of one given program. It does something more *modest* (but also more ambitious at the same time, as it has to be done during the model-checking run and has to be done efficiently). Perhaps POR is inspired by the connection between partial orders and possible linearizations and partial order semantics, but one can understand POR even simpler:

> Under some circumstances, it does not matter in which way steps are done and in other circumstances it does. POR tries to figure out when alternative orders don't matter and avoids them. That needs to be done *while* running the "program".

Since this is done while running the model checker, compromises need to be done how much effort is done to estimate or predict if a step is necessary or not, as it needs to be efficient. It also (and connected to that) needs to be "local". One cannot first generate all runs, then filter out duplicates, and then model check the rest. Instead, when exploring the state space during the model checker run, a "local" decision needs to be made, like "shall I explore the next candidate edge, or can I let it be."

Of course, the criterion should not be trival like: I leave out an edge if I know that I have seen the resulting state already. That's ridiculous, as one might as well follow the edge and then backtrack after discovering that I have seen the state already. Exploring *one more* additional edge and then checking is probably easier compared to to make some fancy overhead to avoid that very last step. One has to do better, namely: one can leave out an edge, if all what *follows* is covered already or actually what will be covered later. At any rate, one cannot expect those estimations to be *precise* in recovering all of the theoretically possible reduction (if one had a full partial-order picture, which one does not have anyway).

The contrete details when to explore and edge and when not depend also on the programming *language* and its constructs. For instance, if one has shared variables, and the model checker is in a state where, in a next step, process 1 can write atomically to a variable or process 2 can write atomically to the same variable, it's clear that one has to explore both alternatives. Or does one? What if they write the same value? Well, perhaps checking that particular situation is not worth in checking, and one may conservatively explore both orderings anyway.

To postpone the details of more concrete language constructs for later, one abstracts away from concrete types of actions first and introduces the concepts of *dependence* and *independence* (for instance, two reads to the same variable may be independent, whereas two writes may not). The theory justifying the POR is then based on notions of independence and when the order of execution is irrelevant and can be commuted.

That is perhaps inspired by partial-order thinking, but can be an approximation at best (for practical reasons), therefore a better name of partial-order reduction may be **commutativity-based reduction** (see Peled [28] who makes that argument).

POR can be also understood as an example of analysis techniques that exploit *equivalences*

> Exploiting "equivalences" means: Instead of checking all "situations", figure which are **equivalent**. That also implies, equivalent wrt. the property being checked) and then check only one **representatives** (or at least not all) per equivalence class.

There are other such techniques along those line. One is known under the name *symmetry reduction*; we don't cover that, but the underlying idea is simple. Often systems have some symmetries, one can exploit. Not that it's a typical model-checking problem, but think of the 8 queens problem: is it possible to place 8 queens on chess-board without that they can attack each other in one move? If one wants to solve that combinatorially, one can exploit the symmetry inherent in the fact that a solution can be rotated 90 degrees and it's remains a solution.

Of course, in this well-known example the symmetry is obvious, and someone who want to solve the puzzle may arrange a search algorithm (and/or the respresentation of the chess board) so make use of that knowledge. In general, symmetries may not be so obvious and one would like to have the model checker detect and exploit them, but that's the general idea. And that is quite similar in POR, where the model check tries to detect equivalent *behaviors*.

(Labelled) transition systems In the lecture we have variously encountered representations based on states and transitions. Kripke structures or Kripke models, transition systems, and also automata belong into that category. There may be minor differences in terminology (states vs. worlds) and perpaps notation, but these representations are not radically different. Maybe the biggest difference is that we made use of the automata as an mechanism for *accepting* runs of a system, in particular the Büchi-automata used for LTL model checking accept infinite-word languages. Transition-systems, on the other hand, represent the system, program, or model under investigation, and do not have accepting states. Another difference was that automata were edge-labelled with letters from some alphabet, whereas for the transition systems we focused on information carried by the state. For instance, one could see propositional information to be a form of labelling: each state is "labelled" with the sets of propositional atoms that are assumed to hold in that state. Another terminology we also used was "valuation".

At any rate, the edges of transition systems so far did not play an important role, but now we will be dealing with edge-labelled transition systems.[1] Typically, if one talks about *labelled* transition systems (LTS), one means transition systems with the *edges*-labelled (independent of wether or not the states are (also) labelled with propsotional information).

Now, the labels in the edges become important, because we need to look into when steps (i.e., transitions) in a system can be re-ordered resp. when some steps can be ignored in an exploration. To do so, the steps should carry information about the action they are representing, like: when a read from x by process P_1 is followed by a read to the same variable by process P_2, then the two steps or transitions can be swapped. Or generally $\xrightarrow{\alpha}$, $\xrightarrow{\beta}$, $\xrightarrow{\alpha_1}$, etc.

Apart from that now we consider transition-labels, the definition of transition system or Kripke-structure is basically unchanged.

[1]In the context of dynamic logics and multi-modal logics, the transition systems had "multiple" transition relations, which is the same as having labelled transitions.

Definition 6.1.1 (Labelled transition system)**.** A *lablled transition system T* over a set of labels L (also called alphabet) is a tuple $(S, \rightarrow, L, S_0, V)$ where

- S is a set of states.
- $\rightarrow \subseteq S \times L \times S$ is the L-labelled relation between states, the transition relation.
- $S_0 \subseteq S$ is the set of starting states
- $V : S \rightarrow 2^P$ is a map labeling each state with a set of propositional variables.

Determinism and enabledness

In the context of automata (Büchi or otherwise), we have introduced the concept of *determinism.* We use the same definition here for labelled transition systems.

Determinism, generally, applies to situations where the result, the outcome, the nexts state etc. is fixed, as opposed to when more than one result is possible. That would be non-deterministic. Functions are deterministic: the output is determined by the input.

For transition systems (and automata), the conventional definion of determinism is that in a state, the successor state determined, *for a given label.* So it's not as *strict* as the the successor is fixed independent from the action label; that would make the transition system a linear structure.

The action labels, abstractly seen, are elements from some alphabet, but in concrete representations, labels may also be "structured" for instance $W_1(x, 1)$ could represent that process P_1 does some write instruction to variable x, writing value 1, or similar. The label may or may not be seen as input

Definition 6.1.2 (Deterministic)**.** A labelled transition system is *deterministic* if $s_1 \xrightarrow{\alpha} s_1$ and $s \xrightarrow{s_2}$ implies $s_1 = s_2$ (for all states s_0, s_1, and s_2).

See also the depiction in Figure 6.2a later.

In that case (and since we are focusing on deterministic systems): we also write $s' = \alpha(s)$ for $s \xrightarrow{\alpha} s'$ (or $\alpha(s, s')$).

The definition here in the introductory section is not the last word about determinism and related issues in this chapter. Section 6.3.1, there will be variations on the topic.

A subtle point in that context will be the question whether there *is* a next state doing a particular step. If, in a state one can *do* and $\xrightarrow{\alpha}$-transition, one says α is enabled in that state. Otherwise, a is disabled there.

Definition 6.1.3 (Enabled)**.** $\xrightarrow{\alpha}$ *enabled* in s, if $s \xrightarrow{\alpha}$. Otherwise $\xrightarrow{\alpha}$ *disabled* in s.

As a side remark: we will talk about paths π

$$s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots$$

but the paths now are not necessarily infinite. For technical reasons, we focused on infinite paths in LTL (which was not a real restriction, as finite ones could be artificially made infinite by stuttering after reaching the official end).

Concurrency in asynchronous systems

In loosely coupled concurrent systems or asynchronous systems, the actions of different processes running in parallel in an asynchronous fashion are largely independent. In an interleaving model, a concrete run or execution (or path) is an arbitrary linearization. In a single-processor setting it's the scheduler's task to pick at each point one possible next step, i.e., make a choice between the different processes who's next. In a practical schedular, of course it will not make a new pick at each clock tick, but let one picked process run its course until its time slice is up one something else triggers the schedular to give another processs its turn. But in the most extreme case, the scheduler may in principle reconsider the options after each tick. That's often assumed in model checking: one is given the system, but one does not know the scheduler, resp. one wants to verify the system for any possible scheduler (and in isolation, i.e., ignoring other unrelated processes running under the same scheduler.

In a situation where the processes are completelety independent, that leads to a behavior sketched in Figure 6.1 (for actions of three processes, each doing just one step). The actions themselves are assumed atomic.

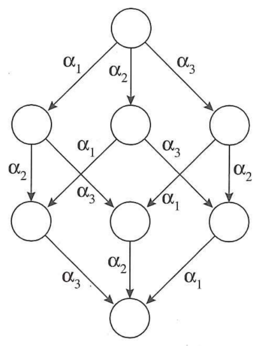


Figure 6.1: Interleaving of three actions / steps

If we have n transition relations (corresponding to n different independent processes, that gives rise to $n!$ different orderings and 2^n states.

Of course that's a very simple case, complete intependence of the relations (and processes). In more realistic situations, some actions or transitions are independent, and some not.

We talked about independence here informally, with an intutive understanding of what it means that processes run indepdently, i.e., have nothing to do with each other. Later we present a definition of *independence* abstractly in terms of labelled transition relations, and not necessarily for actions of parallel processes.

Still, the picture of parallel processes doing steps is useful and gives intuition behind the definitions later.

There will be two aspects of when we call processes (resp. their steps) independent, resp. when they are not independent.

One is that the *order* of steps does not matter, for instance, if two processes write to the same variable at the same time, the outcome will be different depending who comes first. In this situation, both actions or steps can swap their place, i.e., a scheduler can execute them in either order, but the outcome is different. Then the systems would not be considered independent.

But is another aspect: it's still a situation of "who is first" or "whom the scheduler chooses first". However, there is no "second step". So at one point two processes can do a next step, but the one not chosen has lost it's chance. The step that had been *enabled* before the competing situation has been resolved becomes *disabled* by making the decision (that's why we introduced the notion of enabledness for labelled transitions already). At any rate, if the action of one process disables the possible next action of another process, also that breaks independence of course. In terms of concurrent processes, the latter type of non-independence is typical for *synchronization* actions (think lock-taking).

We come back to it in Section 6.3.1

6.2 Pruning the state space: ample sets

Partial order reduction is about reducing the state-space. This section does not provide a concrete solution (for a combination of a concrete programming language and concrete temporal logics). It sketches an skeleton of an idea how achieve the state space reductions.

For the setting, however, we assume a explicit state approach, based on *depths-first search* (DFS).

A super-unrealstic approach would be the following

1. generate explicitly the state space (by DFS for example),
2. then prune it and remove equivalent transitions & states, and finally
3. model-check the property on the smaller space.

Of course it makes no sense to first generate the all of the state space, the trick must be to *avoid* doing that. So a slightly more realistic approach is

1. generate explictly the reduced state space (using a modified DFS) and then
2. model-check the property.

That's still not realistic; a more practical way will be to both stages at the same time, but for presentational reasons, we focus on generating resp. exploring a reduced state-space. That will be done by modifying a standard depth-first exploration procedure.

Actually, this section here will not provide much more than setting the stage. Let's first recall standard DFS, i.e., the problem to explore or traverse a given graph via a tryically recursive procedure.

We should perhaps be careful when talking about traversing a “given” graph or transition sytem. We do *not* want to imply that the transitition system is stored (via some standard graph representation) in the memory ready to be explored. That would imply the super-unrealistic setting where the whole state space is generated up-front.

Instead, the state-space is explored which being generated (hopefully only partially). In model-checking terms and independent from partial-order reduction, that is known as *on-the-fly* model-checking.

Depth-first search does some recursive state exploration: exploring one given means **visiting all its neighbors or successors**, and doing the same for each of them, remembering which states one has seen already, so that one can stop and backtrack, if one sees a state for the second time, otherwise woul would run into an infitely exploration loop.

One realization of depth-first seach in pseudo-code is shown in Listing 6.1. It’s not based on recursive calls of a procedure, but making use of a explicit stack. Furthermore it sketches some way to maintain the states in in some hashed form and organizes the search making use of a work-list. For instance, the basic exploration steps adds successor-steps to the work-list and remove the current action.

Those organasational things are not central for depth-first exploration. The code from Listing 6.1, while still a depth-first strategy however, deviates **crucially** from plain depth-first search. But instead of exploring posible next steps (i.e., all neighbors), the approach realizes the following improvement:

> Don’t explore *all* enabled transitions, follow **enough enabled** transitions.

That’s done in line 6, adding a set of actions to the work-list called **ample.** The word *“ample”* means, especially in this context, something like “enough” or “sufficiently many”. In general, **ample** set of transitions in a state $\subseteq$ set of enabled transitions in a state

```
hash($s_0$);
set on_stack($s_0$);
expand_state($s_0$);

procedure expand_state($s$);
  work_set($s$) := ample($s$);
  while work_set($s$) $\neq \emptyset$
  do
      let $\alpha \in$ work_set($s$);
      work_set($s$) := work_set($s$) $\setminus\{\alpha\}$
      $s'$ := $\alpha(s)$
      if   is_new($s'$)
      then hash($s'$)
           set on_stack($s'$);
           expand_state($s'$);
      end if
  end while
  set completed($s$)
end procedure;
```

Listing 6.1: Modified DFS (ample sets)

Requirements on ample sets Of course, on that level, it's *not* a solution to any problem. We have so far not done more than expressing the wish to achieve our goal of pruning the explored state space by a small modification the standard DFS algorithm (and we use the terminology of ample sets to talk about that modification).

The only thing we know is that for each state, the state's ample set of actions is, by definition, a subset of the actions enabled in that state. So the question is: how to restrict the set of enabled transitions (with this restriction called "ample"). There are some *general* requirements on the restriction:

1. pruning with ample does not change the outcome of the model checking run (**correctness**)
2. pruning should, however, cut out a *significant* amount
3. calculating the ample set: not too much *overhead*

The first is a condition sine-qua-non: correctness must not be compromised. How well the remaining to criteria are fulfilled decides on how much the pruning improves the performance. The last 2, however, are somehow in conflict with each other. Investing too much effort in calculating the smallest possible (and still correct) subset may be counter-productive.

The details of a concrete correct and efficient realization of the ample-set idea also depend on the programming or modelling language.

The general idea of ample sets has been explored in the literature in many variations. The names the papers used for their respective proposal are mostly not very indicative of how a particular solution works; names of concepts in well-known approaches include "sleep sets", "persistent sets", "stubborn sets" . . .

6.3 Equivalent behavior

Before looking an algorithmic approach, we loop a bit deeper into *indepence* of systems. We touched upon that already in the introductory remarks of Section 6.1, namely when we consider (the actions of) two or more processes as independen. Here we define more formally the notion of independence of labelled transitions or relations. That can be somehow related also to the question of systems being deterministic, at least deterministic as far as the result is concerned in that the outcome may not depend on the schedule. Therefore we discuss also that. Especially the definition of independence is important for partial-order reduction.

Later a further aspect needs to be taken into account and that has to do with the fact that we are doing verification of a specification (here an LTL formula). Even if the order of actions don't matter with respect to the outcome, still we have to be careful if the order matters wrt. formula, i.e. whether the answer to the model checking question depends on the order. That will be captured by the notion of visibility resp. invisibility.

6.3.1 Equivalent reordering of behavior: Independence

Determinism, confluence, and commuting diamond property

Partial-order reduction works best for asynchronous, loosely coupled systems, as we said, when different parts of the system run independently and without interfering with each other. Of course, the situation where processes run completely and always indepdent are seldom. Resp. they are uninteresting. If parts of the system are truly independent, there is no need to jointly model them and jointly verify them.

In some way, such a setting with, say two completely independent processes, would be ideal for partial-order reduction. A really optimal reduction could reduce the system to doing the steps of one process first, and then doing the other afterward. If each process is in itself deterministic, the reduced system doing first P_1 and then P_2, would show a linear, deterministic behavor, and no combinatorial state explosion due to unterleaving the steps of the processes when exploring $P_1 \parallel P_2$ ($\parallel$ here for the asynchronous parallel composition).

But of course, it's a stupid idea to try to jointly analyse $P_1 \parallel P_2$ and hope some mystical POR-model-checker will comes up with an ideal reduction (which is unrealistic) if one already knows that P_1 and P_2 are independent.

More interesting are processes that interact from time to time, working on shared variables, exchanging messages etc, but also work independently for large stretches of time. Doing independent actions means that the outcome does not depend on the order of the actions.

Assume that processes are internally deterministic, i.e., there are no internal sources of non-determinism. That means, the "outcome" can only be influenced by the way the actions of different processes are interleaved. In case of independent, actions, the outcome remains the same, as there order can be inverted. In the (unintesting) case of completely independent processes, also the outcome is the same, whether we do $P_1; P_2$ or $P_2; P_1$: every (deterministic) process calculates its own result, it's only a question who comes first.

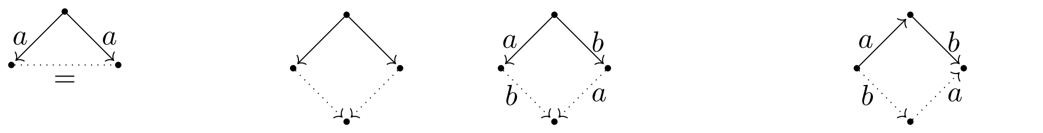
This is to say the notions of determinism and independence are closely connected. They are not the same though (and we have not formally defined what independence actually means). For a transition system to be deterministic, remember see Definition 6.1.2.

The version of POR and the notion of ample sets will be based on a so-called *independence relation* (see Definition 6.3.1). Like ample-sets it will be rather non-concrete. I.e., it will not specify what concrete actions (reading, writing, synchronizations etc.) in some setting *are* independent. It spells out more general conditions as to when a relationship between actions or transitions qualifies to be called an independence relation.

Before doing that, let's at least have a short look at aspects relating "independence" of actions and determinism, because the notions are in spirit similar, but technically not the same.

We will discuss it mostly abstractly, i.e., as a property of relations or pairs of relations. Assume we are dealing with a labelled transition system.

and vice versa. We assume that the transition relations $\xrightarrow{\alpha_i}$ are *deterministic*, and we write $\alpha_i(s)$ for $s \xrightarrow{\alpha_i}$.



(a) Determinism (b) Diamond property (c) comm. d-property (d) swap./commuting

Figure 6.2: Different flavors of determinism and "order does not matter"

Definition 6.3.1 (Independence)**.** An *independence relation* $I \subseteq \rightarrow \times \rightarrow$ is a symmetric, antireflexive relation such that the following holds, for all states $s \in S$ and all $(\xrightarrow{\alpha_1}, \xrightarrow{\alpha_2}) \in I$

Enabledness If $\alpha_1, \alpha_2 \in \mathit{enabled}(s)$, then $\alpha_1 \in \mathit{enabled}(\alpha_2(s))$

Commutativity: if $\alpha_1, \alpha_2 \in \mathit{enabled}(s)$, then

$$\alpha_1(\alpha_2(s)) = \alpha_2(\alpha_1(s))$$

The complement of a independence relation is called, not surprisingly, a *dependence relation*. I.e., given some independence relation I then $D = (\rightarrow \times \rightarrow) \setminus I$ is a dependence relation.

Is that all?

Let's look at the commuting diamond diagram again, from Figure 6.2c, resp. repeated in Figure 6.3. We made arguments that situations of that form have the potential for saving since the order of the two steps does not matter, since each order, they reach the same state, r in Figure 6.3.

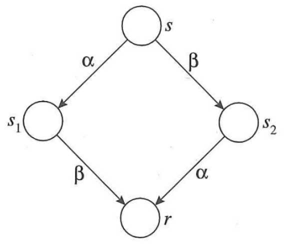


Figure 6.3: Commuting diamond

There are, however, two issues or complications to take into account

1. The checked **property** might be sensitive to the choice between s_1 and s_2 (and not just depend on s and r)
2. s_1 and s_2 may have **other successors** not shown in the diagram.

6.3.2 Equivalence wrt. formula(s): visibility and stuttering

The first point is an aspect we have ignored in the discussion so far, namely the influence of the property or the class of properties to check. However, in a situation like the communting diamond, of one wants to explore the path via s_1 only, ignoring the one over s_2, it must be sure that satisfaction of the property to check does not depend on visiting those states. For instance, if there is a proposition that holds in s_1 but not in s_2 or vice versa, then one has to explore both alternatives. If propositions can make differentiate between two states, one says, the formula can "see" the difference or that the two states resp. their difference is *observable* or *visible* by the formula. The picture behind that terminology is that checking for a formula is a way to observe a system. Behavior that cannot make a difference whether the formula holds or not is not observable or invisible. That terminology is often used for classes of formulas or whole logics. More expressive logics can observe more behavior, resp. differentiate between more systems that weaker ones. Weaker ones, in that sense, have more potential for reductions, here partial-order reduction.

Here, for our purposes, we define when to call a *transition* as being (in-)visible for a set of propositional atoms.

Definition 6.3.2 (Propositional vsibility)**.** Assume a valuation $V : S \to 2^P$. The transition relation $\xrightarrow{\alpha}$ is *invisible* wrt. a set of $P' \subseteq P$ if for all $s_1 \xrightarrow{\alpha} s_2$ and all $p \in P'$, the following holds

$$s_1 \models p \quad \text{iff} \quad s_2 \models p \tag{6.1}$$

The definition means, that no α-transition changes the truth-status of an of the propositional variables in P'.

So, state changes which don't change the truth status of any propositional variable (in P') is invisible (wrt. P'). It's a step that does not matter concerning the question whether the transition system satisfies the specification or not. Steps that don't matter are also called *stutter steps* and the phenomenon *stuttering*.

We have encountered stuttering already in connection with LTL and the definition of π. Paths, in that context, were defined as being necessarily infinite. To accomodate terminating behavior of a system, we extended an terminating behavior by an infinite sequence of stuttering steps at the end, to obtain an infinite path.

Of course, it is possible that the system does some "actual" stuttering while still running, doing stretches of invible steps. Those stretches of stuttering steps, resp. maximal such stretches, are sometimes called *blocks*. Note in passing: terminating behavior, which stuttering at the end, consists of a finite number of blocks, since the stuttering extension after termination is represented by one infinitely long block.

Two paths that consist of the "same" sequence of blocks are called stuttering-equivalent. When saying the "same" sequence of block, we mean that each block of π is represented by a corresponding block in π' with the same propositional valuation, but the size of the block, i.e., the number of stuttering steps inside the block may different. This is shown in Figure 6.4. We write $\sim_{st}$ for stutter-equivalence for paths.

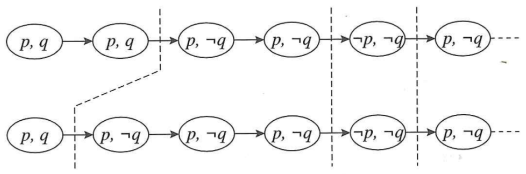


Figure 6.4: Stuttering-equivalent path

The different states in a block have the same status as far as the satisfaction of propositional variables is concerned. That carries over to general propositional formulas, of course. But what about temporal properties of LTL? Does an LTL formula that holds for some path also hold for a stutter-equivalent one, does $\pi \models \varphi$ and $\pi \sim_{st} \pi'$ imply $\pi' \models \varphi$. To say it differently: can the addition or removal of stutter steps in a π change the truth status of the path with respect to a temporal property.

Before we closer at that, let's first give a name to LTL formulas whose truth-status is untouched by adding or removing stuttering steps.

Definition 6.3.3 (Stutter invariance)**.** An LTL formula φ is *invariant under stuttering* iff for all pairs of paths π_1 and π_2 with $\pi_1 \sim_{st} \pi_2$,

$$\pi_1 \models \varphi \quad \text{iff} \quad \pi_2 \models \varphi$$

Using the observability-terminology, one could say that for those formulas stuttering-steps are invisible. Technically, we have introduced the notion of invisibility for propositional proporties only (and we defined it for transitions $\xrightarrow{\alpha}$ in a transition system) in Definition 6.3.2, but conceptually it's analogous.

Back to the question: can additional stutter steps in a π change the truth status of the path with respect to a temporal property, i.e., are LTL formulas stutter invariant (or at least some). We clarified already that for propositions are stutter invariant, but that's not very interesting and helpful.

However, adding (or removing) a stutter steps may change satisfaction of $\bigcirc$-*formulas!*. That's because blocks are *finite*. Assume that some $\bigcirc p$ us be true at *last* position of a block and whether it's true or not depends on the situation in the subsequent block. Therefore, extending the block longer by adding a stutter step may change the truth status in the considered position. Similar for removing stutter steps.

It turns out, if we banish $\bigcirc$ from LTL, formulas become stutter invariant. The truth status of $\Diamond$ and $\Box$ (and for the more complex binary operators) does not depend on the length of the stutter blocks. For example $\Diamond$ refers to some finite point in the future, and adding or removing stutter steps does not change that. Note that stuttering means adding or removing only *finitely* many steps. That means, stuttering cannot turn a finite block into an infinite one.

The restricted form of LTL is known as next-free LTL, and abbreviated as LTL$_{-\bigcirc}$(or also LTL$_{\text{-X}}$).

$\mathrm{LTL}_{-\bigcirc}$ is stuttering invariant. That's good for reducing the state space, because it gives hope to ignore states, ideally to take only one representative in each block. And the more stuttering and the larger the blocks, the more potential for space reduction.

We defined the concepts like visibility and stutteing on transition systems and paths. Those stem of course from the description of a concurrent system, consisting of processes or threads running in parallel. As a general observation: the more loosely coupled or more asychronous a system is, the more one can expect to see stuttering. That's probably not self-evident, it's also not a mathematical fact. It's also not just a consequence on the loose coupling of the system, but also based on how one typically specifies properties for such loosely coupled system.

Of course the logic because less expressive that way. That's in a way the point, making it less expressive makes stutter steps unobservable.

stuttering (in this form): important for *asynchronous* systems ...

6.4 POR for $\mathrm{LTL}_{-\bigcirc}$

$\mathrm{LTL}_{-\bigcirc}$ is is one useful and fuitful general setting for POR (especially for asynchronous systems), so let's look at that. Partial-order reduction and related ideas have also been studied from different angels and settings and logics. One may also try to achieve reductions for, say, safety-properties, only,, or even try with reductions per formula. But we do it for $\mathrm{LTL}_{-\bigcirc}$, and we want to do that in terms of the *ample-set* variant of depth-first search.

Given a transition system T, let's refer to the reduced or pruned version as $T^{\text{✂}}$, reduced in the sense containing only the part of the state space explored via the neighbors in the ample-sets.

$$T, s \models \varphi \qquad \text{iff} \qquad T^{\text{✂}}, s \models \varphi$$

Since we a doing a form of LTL, the correctness is mainly a condition on *paths,* i.e., all the path in T starting from s. So the correctness is assured if each path in the original system is equivalently represented after pruning, at least once.

each path π_1 in T starting in s is represented by an **equivalent** path π_2 in $T^{\text{✂}}$, starting in s

6.4.1 Conditions on selecting ample sets

The correctness condition spell out for path is straightforward and should be obvious. Far less obvious is how to achieve that by doing a proper definition of the ample set. Of course, if correctness were the only goal, that could be trivially achieved by setting the ample set to be the equal to the set of enabled transition in each state. Then there would be no reduction whatsoever, and that is of course correct. The trick must be to make the ample set as small as (reasoably) possible, without compromising correctness.

In the design of the algorithm, it's about making a selection of enabled steps and following those, leaving out others., local decisions, paths ...

reorderings of

- each pruned path can be "reordered" to an which is explored (using *independence*). It also includes a condition covering end-states. See Section 6.3.1.
- make sure that the reordering (pre-poning) does not change the logical status, based on the notions of *stuttering* and *visibility*. See Section 6.3.2.
- "fairness": make use not to prune "relevant" transitions by letting the search **cycle** in irrelevant ones.

We will later show code snippets covering those conditions.

Reordering conditions ($\mathbf{C}_0$, $\mathbf{C}_1$)

This section is making sure that, when cutting out a path or rather a whole set of paths by not exploring some neighbors, the omitted paths are covered equivalently otherwise.

This requirement is split into to conditions. The first one, let's call it $\mathbf{C}_0$ is very trivial. We know that $ample(s) \subseteq enableds$ and thus if $enabled(s) = \emptyset$, also the ample set is of course empty: if there are no neighbors, one can not continue exploring anyway and the depth-first seach backtracks. Note that it's not about that there are no more *unexplored* neighbors; also in this case the exploration backtracks. It's about having reached a dead end.

But if there are enabled steps, i.e. $enabled(s) \neq \emptyset$, then it's clear that we cannot set the ample-set to $\emptyset$ as well. Without at least following one of the possible neighbors, there's no way to know what would have happened if we followed at least one. So we have to require the following:

$\mathbf{C}_0$: stop at dead ends, only.

$$ample(s) = \emptyset \text{ iff } enabled(s) = \emptyset \tag{6.2}$$

The next condition is a bit more tricky. It's also not formulated in an actionable or readily implementable form. It spell out just a condition on the ample sets in a state connection with paths that start in that state.

$\mathbf{C}_1$ Along every path in T starting at s, the following condition holds: a transition **dependent** on a transition in $ample(s)$ cannot be executed without a transition from $ample(s)$ occuring *first*.

Observation 6.4.1. Under condition $\mathbf{C}_1$, we have:

$$\mathit{ample}(s) \bowtie \neg \mathit{ample}(s) \ . \tag{6.3}$$

As a consequence of the definition: in a state, all enabled ones but not ample are independent from the ample ones. If a transition (assuming otherwise) that is dependent on one in ample would both be enabled but not covered by the ample set, then one could simply start a path using that transition, contradicting the condition.

Another way of saying the same is

The set $\mathit{ample}(s)$ is closed under #

As a consequence of $\mathbf{C}_1$, one can distinnguish two forms of paths, with respect of which actions from within and outside the ample-set they contain. These two forms are

$$\beta_0\beta_1 \ldots \beta_m \alpha \quad \text{or} \quad \beta_0\beta_1\beta_2 \ldots \quad \text{with} \quad \alpha \in \mathit{ample}(s) \quad \text{and} \quad \beta_i \bowtie \mathit{ample}(s) \ . \tag{6.4}$$

In the first case, the path has a finite prefix $\beta_0\beta_1 \ldots \beta_m\alpha$, i.e., after a few β_i independent from all actions in the ample set, at some point a member of the ample set (of s) is taken. In the second case, there is an infinite sequence $\beta_0\beta_1\beta_2 \ldots$ where also $\beta_i \bowtie \mathit{ample}(s)$.

Note that the two case are *not* mutually exclusive. Remember that the ample sets are closed under #, thanks to $\mathbf{C}_1$, but the condition is a loose one. It states that if a action is outside in the ample set, it has to be independent from it. However, it does not forbid to put independent actions inside.

However, without loss of generality, we can focus on the case that all $\beta_i \notin \mathit{ample}(s)$, in which case the two conditions from equation (6.4) are mutually exclusive. Either there is eventually an α from the ample set of s preceded by β's outside that ample set. Or there are infinitely many such β's. As a side remark, in that form, it is a form of weak-until property of the paths.

We can now make an argument that we can reorder paths appropriately, using *commutation*. Consider the situation in Figure 6.5, which corresponds to the first case of equation (6.4). The assumptions are, as agreed, that $\alpha \in \mathit{ample}(s)$, that $\beta_i \notin \mathit{ample}(s)$,

Consider further the two paths starting in s,

$$\pi_1 = \vec{\beta}\alpha \quad \text{and} \quad \pi_2 = \alpha\vec{\beta} \ .$$

As far as the unreduced transition system is concerned, we have that $\pi_1 \in T, s$ implies $\pi_2 \in T, s$ (and vice versa). That's a direct consequence from the fact that α and the β's are independent (see Definition 6.3.1).

Concerning $T^{✂}$, we have under the given assumptions,

$$\pi_1 \notin T^{✂} \quad \text{and} \quad \pi_2 \in T^{✂} \ ,$$

at least if we assume $m > 0$, excluding the uninteresting and trivial corner case that there are no β's at all. So it's an example of an actual reduction, the pruned system leaves out π_2 but explores the reordered one

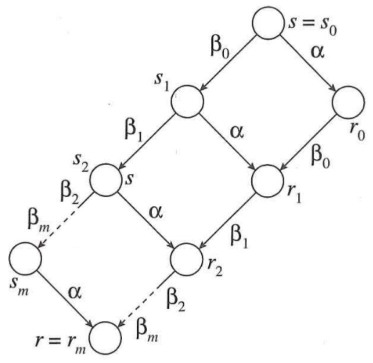


Figure 6.5: $\mathbf{C}_1$ (reorder) and $\mathbf{C}_2$ (invisibility)

Invisibility (C$_2$)

So, as far as their *existance* in T is concerned, π_1 and π_2 are "equivalent" (and all the "intermediate" paths as well, like $\beta'\alpha\beta''$). If one path exists, it's guaranteed that the other exists, and vice versa and they have the same start and end state (s and r in the picture).

But are they interchangable also wrt. the intermediate, visisited states, in particular, are the two paths interchangeble wrt. the *property* we model check? Well, one paths visits $s_0, s_1, \ldots s_m, r$ the other one $s, r_0, \ldots, r_m$ (with start and end states coinciding, i.e., $s_0 = s$ and $r_m = r$). So the question is:

does it matter if one passes though the state r_i or the state s_i?

Of course, it may matter if some property holds for r_i but not for s_i or vice versa. The r_i and s_i states are connected by α, i.e.

$$s_i \xrightarrow{\alpha} r_i$$

Now, whether π_1 or π_2 is taken (or one of the "intermediate mixtures) does not matter provided that same formulas hold, comparing r_i with s_i. That's guaranteed if α is invisible (with respect to the atomic propositions)

The answer is clearly *no, it does not matter* provided that the satifaction or "dissatisfaction" of the property does not depend on whether one is in s_i or r_i. That form of "invariance" has been called "invisibility".

The perspective is that the a formula *observes* the transition system, it can "see" if a truth status changes (from true to false or the other way around). Observing *changes* means being able to observe transitions. And, in this picture, a transition is invisible or not observable, if taking said transition doe not lead to change of any truth values. Actually, visibility has been defined with resp. to atomic propositions only, more complex formulas don't need to be considered, resp. their non-observability follow as a consequence.

$\mathbf{C}_2$ (invisibility):

If s is not fully expanded, then every $\alpha \in \mathit{ample}(s)$ is *invisible*.

A state s is *fully expanded* if $\mathit{ample}(s) = \mathit{enabled}(s)$. That's a situation where all enabled transitions are explored anyway, so in that case, the ample-set at s is certainly ok, without need to require invisibility of any transitions.

If we ignore transitions, the non-ignored transitions must all be *invisible*.

$\mathbf{C}_3$ (cycle condition)

The previous condition $\mathbf{C}_2$ insisted on invisiblity of an action α, in case one omits alternatives. The transition system from Figure 6.5 shown previously illustrated that, that if α is invisble, the uncovered path (in the picture) with α at the end can be reordered with α at the beginning *without* omitting intermediate states with different logical status. That last condition about invisibility took care about *one* form of paths from equation (6.4) that follow as consequence of condition $\mathbf{C}_1$, namely the one with finitely many β_i's (not in the ample set of a state s) followed by one α from the ample set of s.

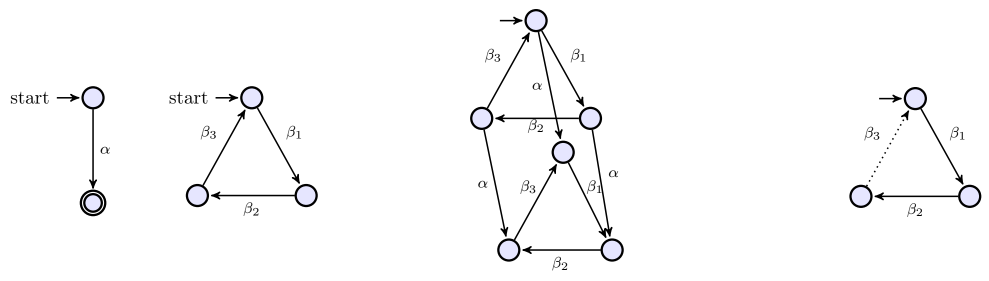
The final condition $\mathbf{C}_3$ is about the second form of paths from equation (6.4), namely the ones with infinitely many β_i, and *never* any α from the ample set.

Based on the intution of the ample sets we can already intuitively see that one has to be careful there. The ample set in a state represent the transitions that should be explored, and the rest from $\neg\mathit{ample}(s)$ are the one that are intended to be ignored (because one can argue that they are equivalently covered otherwise during the exploration). Now, a transition in the ample set of a state marks it as “this transition needs to be explored”. Postponing it forever is not the way to go.

Like the other conditions as well, condition $\mathbf{C}_3$, is not a condition on the behavor or the form of paths (like “don't look at paths where transitions $\alpha \in \mathit{ample}(s)$ are postponed forever”), it's a condition on the forms of the ample set in the state that must be designed in such a way that, when running the system, all paths have the desired properties (in particular guaranteeing correctness, or avoiding infinite postponements of the form sketched).

Let's look at Figure 6.6 The three figures serve to illustrate the previously discussed problem of “infinite postponement”. To complete the example, we need to add one piece of information, namely the “logical part”, i.e., at which states satisfy which propositions.

Figure 6.6a contains two separate processes and Figure 6.6b their (asynchronous) parallel composition. In the example, assume that α is *visible*, the β_is are *invisible*. We also assume that the α is *independent* from all the β's. With all its transitions invisible, the second process is stuttering. That means also, that, as far as the property to be model-checked is concerned, the focus is on the first process, the second one is irrelevant as far as the property is concerned.



(a) Two processes (b) Parallel composition T (c) $T^{\text{✂}}$

Figure 6.6: Illustration of the cycle condition

Since the actions of the two processes are also indepdent, the two processes are completely independent. There is no synchronization between them (independence) and they are not "secretly" couple via the property (invisibility). The example is this pretty trivial, but it's used to illustrated the need for the last condition.

More concretely, the picture may repesent a situation, where there is one boolean variable, initially say "false", and the process to the left sets it to "true" via it's transition $\xrightarrow{\alpha}$, i.e., the label α may represent the assignment `p := true`. The other process does not do anything (except spinning around, cycling through its three states), resp. does nothing interesting as far as the property to be checked is concerned. For instance, the property could be $\Diamond(p = \mathit{true})$, and the second process only operates on variables other than p

Figure 6.6b shows the two processes running in parallel in an asynchronous fashion, i.e., interleaving their steps. The overall combined behavior is given by the transition system T, with 6 states. In that T with its 6 states and if we assume one propositional atom p, then p is false in all 3 states on the top of the picture, and true in the three states on the bottom.

For this system, we can find ample sets that satisfy all the three conditions so far, but still fail to achieve correctness. That's easily doable by *systematically ignoring* α, i.e., not including this transition in any of the ample sets.

I.e., each state has an one element ample set $\mathit{ample}(s) = \{\beta_i\}$, and α is not included anywhere.

It's easy to check that this choice satisfies $\mathbf{C}_0$ (trivally, since no ample set or enabled set is empty), $\mathbf{C}_1$ (since α is assumed to be independent from the β_is; remember that $\mathbf{C}_1$ speaks about paths in T, not in $T^{\text{✂}}$). And finally $\mathbf{C}_2$ is satisfied as well, as the example is constructed in such a way that the α_i are all **invisible**, as required by $\mathbf{C}_2$.

In contrast to the β's, transition α *is* visible, it does not stutter, so taking it matters wrt. the verification problem. However, the ample sets chosen as given, leads to explorations in $T^{\text{✂}}$ *ignoring* α.

The last condition $\mathbf{C}_3$ excludes such infinite avoidance. Seen as condition one the graph itself, it's a condition on a cycle, not a condition on infinite paths resp. only indirectly so, since in finite-state systems, infinite paths must come from running through at least one cycle. What needs to be ensured is that a situation as in the example cannot occur. That $\xrightarrow{a}$ is not included in *some* of the 3 states of the last picture is fine. What is not fine is that it's left out in *all* of them in the cycle. If left out completely would allow, as in the example, to construct a path running through this cycle where the transition is constantly enabled but always in $\neg ample(s_i)$, so no state "takes responsiblity" to at least one time, explore that edge.

In the example, the neglegted edge α is a **visible** one. But the requirement stating "do not systematically neglect an edge" also applies to invisible ones, as well. Even if some edge itself is invisible, one may reach behavior after taking it that *is visible* and needs to be checked. The example is also specific insofar in that $\xrightarrow{\alpha}$ is *continuously* enabled (but not taken). Condition $\mathbf{C}_3$ is more stringent: don't neglect a transition $\xrightarrow{\alpha}$ that is somewhere enabled in a cycle.

This condition is connected with the notion of *fairness*. It's a notion that is relevant in concurrent systems. In practical systems (like operating systems), it also can be understood as a property of a *scheduler*. In our example, with two processes, a behavior that constantly schedules the second process, with systematically ignoring the first one (despite the fact that it *could* do a step, namely $\xrightarrow{\alpha}$), that's a non-fair behair. Of course, after the first process has done $\xrightarrow{\alpha}$, it cannot do any further (no transition is enabled, and that will remain so as well, as the process is terminated). If, in that situation, the scheduler "choses" only $\xrightarrow{\beta_i}$ steps from the second process, but no steps from the first, that does not count as being unfair.

There are, though, two variations of the concept of fairness, namely *strong fairness* and *weak fairness*. The illustrating example corresponds to the *weak* variant (resp. it illustrates behavior which not weakly fear). Since it's not even weakly fair, it also fails to be strongly fair, though. It illustrates a situation, where $\xrightarrow{\alpha}$ is neglected despite being *constantly enable*. The chose infiniten path $\beta_1\beta_2\beta_3\beta_1\ldots$ has an inifinite sequence of points where α is *constantly* enabled. Weak fairness requires that one cannot have an action (like α) enabled infinitely long without also taking it. fairness

Strong fairness say: of an action is enabled *infinitely often* (but can be disabled in between the places when it's enabled again), then, for fairness sake, it must be taken: strong fairness means, if an action is enabled infinitely often in an execution, it needs also to be taken infinitely often.

Condition $\mathbf{C}_3$ coming up next corresponds to the strong variant of fairness.

Side remark 6.4.2 (Zeno)**.** A final side remark (not too relevant perhaps for POR): as part of the illustration example, the chosen β_i transitions are all invisible. The resulting behavior (without imposing $\mathbf{C}_3$) is not just unfair in the described sense, neglecting $\xrightarrow{\alpha}$, the behavior is also doing an infinite amout of do-nothing steps (here formulated by having the $\xrightarrow{\alpha_i}$ as invisible). The have no influence on the satisfaction of formulas. More practically, one can see then as no-operation or skip steps (sometimes executing `NOP` steps, eating up processor cycles without doing anything) or do-nothing "stutter" steps added to the model (like we did in LTL).

Either way: infinitely many do-nothing or skip or stutter steps is seen as a simple and discrete form of so called *Zeno-behavior*. That's in honor of an old Greek philosopher Zeno of Elea, who is remembered for some speculative paradoxes (retold by Aristotle), often concerning infitely many (smaller and smaller time) steps. The most well-known of those is probably the tale of Achilles and the tartoise, racing against each other. □

Now, without furter ado, here's the condition

> **C**$_3$ (cycle condition):
>
> A cycle is not allowed if it contains a state in which some transition α is enabled but never included in $\mathit{ample}(s)$ for any state s on the cycle.

With all the conditions nailed now, let's go back to the two issues we sketched in connection with the commuting diamond Figure 6.3. As a recap: the two issues mentioned where:

1. Does the satisfaction depends on chosing the path via s_1 or via s_2?
2. When following only one path, do we forget to check successors?

Let's focus on the second issue and let's look at Figure 6.7, where s_1 had successor(s) reachable via transition γ. Assume the state s_1 is omitted, i.e., $\beta \in \mathit{ample}(s)$, but not α.

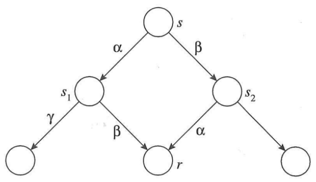


Figure 6.7: Forgotten successors?

So, by omitting s_1, do we forget to check parts of the system?

the conditions imply

$$\begin{array}{ccc} ss_2r & \sim_{st} & ss_1r \\ ss_1s_1' & \sim_{st} & ss_2rr' \end{array}$$

Calculating the ample sets

We don't go much into details here, but looking at the different conditions, it's clear that they quite different in complexity. The conditions need to be checked *on-the-flow* during the depth-first exploration. Therefore checking the condition can prefably be done efficiently.

C$_0$ and **C**$_2$ are easy. More tricky is **C**$_1$. Note that the condition refers to T, to to $T^{\text{✂}}$. It is equivalent to reachability checking. As for **C**$_3$, also that is tricky. One can replace the condition by a modified one, that is easier to establish namely at least one state along each cycle must be fully expanded

Since we do DFS: watch out for "back edges": $\mathbf{C}'_3$: If s is not fully expanded, then no transition in $ample(s)$ may reach a state that is on the search *stack*

Applying the ample-set theory of POR

The whole presentation of partial-order reduction based on ample-sets is rather abstraction. It speaks about multiple relations, i.e., the labelled transitions and conditions on them, like being independent resp. when they are inter-dependent.

In practice actions correspond to atomic steps in a (concurrent) program or a model thereof. It's also important that those steps are also *deterministic*, since one general assumption underlying the whole setting is that the labelled transition system is deterministic. If that's not the case, the framework does not work in the presented form, resp. need to be refined.

That's typically not big restriction, atomic individualy steps are deterministic. A source of non-determinism, however, could be is abstraction, for instance in that the model abstracts away from details of a concrete programs. Also that actually is not really problematic for the POR framework.

Anyway, we

General remarks on heuristics

- dependence and independence $\bowtie$ "theoretical" relation between (deterministic) relations
- "use case": capturing steps of concurrent programs
  - processes with program counter (control points)
  - different ways of
    - synchronization
    - sharing memory
    - communication
- calculating (approx. of) ample sets: dependent on the programming model

Let's fix some notations and definitions. We write now α for $\xrightarrow{\alpha}$. We assume a fixed, finite set of processes P_i or just i for short. We also write T_i: those transitions that "belong to" P_i

In the abstract discussions we referred to states of a transition system by s or simulir. Now, the states are assumed to have more internal structure. Since the program is seen as the parallel consposition of a number of processes. The overall state contains then the tuple of the states of the individual processes. Each process is a particular location, or *control-flow point* in its own execution. If the control-flow is depicted as some transition system (or control-flow graph), that control-flow point corresponds to one node in that transition system. We can also see it as the current value of the *program counter*. For notation, given a global

Definition 6.4.3 (Referring to parts of a global state)**.** Let $pc_i(s)$ be the value of the program counter of process i in state s. We refer by $T_i(s)$ to the set of enabled transitions in state s enable in process P_i

Definition 6.4.4 (Relationships between actions)**.** $dep(\alpha)$: transitions interdependent with α $pre(\alpha)$: transitions whose execution *may* enable α

can be over-approximative

When are transitions (inter)dependent The definition of independence has two aspects, commutativity is one. The other one concerns enabledness. Basically, it's about *preserving* enabledness, resp. to forbid an action to disable the other. Since the definition of independence is symmetric, this condition works both ways. For α_1 and α_2 being independent means that for all states where both are enabled, taking α_1 must not disable α_2, and vice versa.

Being inter-dependent means that communtativity or the enableness condition breaks, at least in one state. For the condition that independence breaks in a least one state, it's however likely that if two enabled actions don't commute on one state, the don't commute in a different states where they happen to be also enabled. Same for breaking the preseving-enabledness condition. After all, actions like reading from or writing to memory or other forms of interactions, work uniformely.

We said that $\alpha_1 \# \alpha_2$ mean violating commutativity or disabling the competitor. Actually, it's unlikely or unrealistic to find examples of actions which violate both. The reason is simple: assume that taking α_1 disables α_2. Then j$\xrightarrow{\alpha_1}\xrightarrow{\alpha_2}$ does not occur, with α_2 disabled after taking α_1. That means, $\alpha_2(\alpha_1(s))$ does not exists or is undefined. Then one could make the argument, that breaks the second requirement for indepedence

$$\alpha_2(\alpha_1(s)) = \alpha_2(\alpha_2(s)) \tag{6.5}$$

becaise the left-hand side is undefined and the right-hand side may still be ok. To break independence, it's enough that α_1 disables α_2, it's not needed that α_2 also does the same with α_1. In practice, when an action disables another, the situation is symmtetric. It describes a *choice point* in a program where one of two (or more) actions is taken and the others disabled. Choice-point not in the sense of a case-construct of a sequential program. Typically would be *lock-taking* actions. The hole purpose of a lock is to do exactly that (to ensure mutex): let at most one action succeed, and disabling competitors, at least for some time, until the lock becomes free again.

But the behavior of the lock-taking actins is symmetric, if process P_1 takes the lock at a place where P_2 could also take it, P_2 is disabled, and vice versa. That means, both sides of equation 6.5 don't exists or are undefined.

Assuming that in practice such disabling actions are symmetric, there could be another reason why in a situaton $\alpha_1 \# \alpha_2$, both swapping and the enabledness condition are violated. That has to do with the fact that one needs to find only one state where the conditions for $\alpha_1 \bowtie \alpha_2$ don't hold. And it could be that in one state, commutativity is violated, and in another one the enabledness part. So that would be another situation when both fail, namely at different states. But again, that unplausible in practice. Actions typically act uniform in states.

Transitions that may enable α ($pre(\alpha)$) Assume α is an action from one specific process P_i

$$pre(\alpha) \supseteq \{\beta \mid \alpha \notin enabled(s), \beta \in enabled(s), \alpha \in enabled(\beta(s))\} \tag{6.6}$$

- $pre(\alpha)$ includes
  - "local predecessor" of i ("program order")
  - **shared variables**: if enabling conditions of α involves shared variables: the set contains *all other transitions* that can change these shared variables
  - **message passing**: if α is a send (reps. receive), the $pre(\alpha)$ contains transitions of other processes that receive (resp. send) on the channel

6.4.2 Some code snippets for the conditions

We have discussed the different conditions in some detail and mentioned a bit how they can be checked. Here, for completeness, some pseudo-code that sketch how to algorithmically determine the conditions. For the third condition, the code shows the variant $\mathbf{C}_3'$ mentioned shortly earlier which is easier to check than the more precise one $\mathbf{C}_3$.

```
function ample ($s$) =
 for all $P_i$ such that $T_i(s) \neq \emptyset$ // try to focus on one $P_i$
   if
        check_C1($s, P_1$) ∧
        check_C2($T_i(s)$) ∧
        check_C3'($s$, $T_i(s)$)
   then
          return $T_i(s)$
   if
 end for all      // too bad, cannot focus on any but
 return $enabled(s)$ // fully expanded can't be wrong
end
```

Listing 6.2: ample

```
function check_C2($X$) =
  for all $\alpha \in X$
  do  if   visible($\alpha$)
      then false
      else true
```

Listing 6.3: Check $\mathbf{C}_2$

```
function check_C3' ($s, X$) =
  for all $\alpha \in X$
   do
      if   on_stack($\alpha(s)$)
      then false
      else true
```

Listing 6.4: Check $\mathbf{C}_3'$

```
function check_C1 ($s, P_i$) =
  for all $P_j \neq P_i$
   do
       if          $dep(T_i(s)) \cap T_j \neq \emptyset$
          $\vee$
                   $pre(current_i(s) \setminus T_i(s)) \cap T_j \neq \emptyset$
       then return false
   end forall;
  return true
```

Listing 6.5: Check $\mathbf{C}_1$

Chapter 7 Symbolic execution

What is it about?

Learning Targets of this Chapter

The chapter gives an not too deep introduction to *symbolic* execution and *concolic* execution.

Contents

7.1 Introduction

The material here is partly based on [16] (in particular the DART part). The slides take inspiration also from a presentation of Marco Probst, University Freiburg, see the link here, in particular, some of the graphs are reused and adapted from that presentation. More material may be found in the survey paper [3].

Symbolic execution is a quite "old" technique, one or the starting point for it is [21] from 1976. It's a technique natural also in the context of *testing* (and in the chapter, we talk also about some aspects of testing). We cover also the ideas behind *concolic* execution, a portmanteau word meaning "concrete and symbolic". Symbolic execution is also used in compilers, for optimization and code generation.

```
f(int x, int y){
  if (x*x*x* > 0) {
    if (x > 0 && y == 10) {
      fail();
    }
  } else {
    if (x > 0 && y == 20) {
      fail ();
    }
  }

  complete();
}
```

Listing 7.1: Sample code

Let's take a look at Listing 7.1. The code has no particular purpose, except that it will be used to discuss testing, symbolic execution, and also concolic execution. The function has two possible outcomes, namely success or failure, represented by calls to corresponding procedures. Note that non-termination is not an issue, there is no loop in the procedure. In general, symbolic execution works best or most straightforward on straight-line programs and loops pose challenges for symbolic execution. The problems with loops are similar the challenges they pose for bounded model checking . Indeed, BMC shares some commonalities with symbolic execution: both are making use of SAT/SMT solving.

How to analyse a (simple) program like that?

One has different options there; and can of course pursue more than one of them. One standard thing to do is **testing**. Testing is probably *the* most used method for ensuring software (and system) "quality".

Testing is a broad field and has very many aspects, and there are very different approaches to testing and techniques and different testing goals. Those techniques are also used in combination and in different phases of software engineering cycle.

For function bodies in isolation like the ones shown, *unit testing* is a way to go (perhaps a part of a larger testing set-up for the whole product).

One could do "verification", whatever that means. The term could include code review, or a formal verification of the code towards a specification, perhaps with the help of a theorem prover. Later we will disucss symbolic and concolic execution. Before we do that, what about model-checking?

Model-checking a program like that is challenging. Model-checking methods and corresponding (temporal) logics are mostly geared towards concurrent and reactive programs anyway. In particular, standard model checking techniques are not very suitable for programs involving data calculations. The given code is a procedure with *input* and its behavior is *determined* by the input. So, *given* the input, it's a deterministic (and sequential) problem and with a concretely fixed input, there is also no "state-space explosion". Generally, though, the problem is *infinite* in size, if one assumes the mathematical integers as input, resp., unmanagably huge, if one assumes a concrete machine-representation of integers, i.e., for practical purposes, the state space is basically infinite, even though the program is tiny.

Of course, common sense would tell that if the program would works for having $x = 2345$ and $y = 6789$, there is no reason to suspect it would fail for $x = 2346$ and y unchanged, for example. In that particular tiny example, that is clear from the fact that those particular numbers are never even mentioned in the code, they are nowhere near any corner case where one would expect trouble.

This way of thinking (what are corner cases) is typical for testing, and is obvious also for unexperienced programmers (or testers). Of course it is based on the assumption that the code is available, as the intuitive notion of "corner case" rests on the assumption one can analyze the code and that one sees in particular which conditionals are used. For instance, there's no way of knowing which corner cases the `complete()` might have, should it have access to those variables `x` and `y`, except perhaps some "usual suspects" like uninitialized value, 0, `MAXINT` and +/- 1 of those perhaps.

There are many forms of testing, in general, with different goals, under different assumptions, and different artifacts being tested. The form of (software) testing where the code is available is sometimes called *white-box testing* or *structural testing* (the terms white-box and black-box testing are considered out-dated by some, but widely used anyway).

Coverage

The intuitive thinking about "corner cases" basically is motivated by making sure that all possible "ways" of executing the code or actually done. In testing that's connected to the notion of *coverage*. In the context of white-box testing, one want to cover "all the code". What that exactly means depends on the chosen coverage criterion or criteria. The crudest one (which therefore is not really used) would be *line coverage* that every line must be executed and covered by a test case. It's not a useful concept: it would allow the tester to claim 100% line coverage if the program would be formatted in a single line… That's of course silly, so typically, criteria are based on covering elements of the programs represented by a *control-flow graph* (CFG, see the pictures later), and then one speaks about node coverage, or edge coverage, or further refinements, depending on the set-up. For instance, if one had a language that supports composed boolean conditions, and if one had a CFG representation that puts such composite conditions into *one node* of the CFG, then covering only that node, or covering both true and false branch of that node will not test all the individual *contributions* of the parts of the formular to that true-or-false condition. If want wants more ambitious coverage criteria, one may that those into account as well, which would be better than simple edge coverage.

So, there are very many coverage criteria. Known ones include

- node coverage
- edges coverage, condition coverage
- combinations thereof, and
- path coverage

They are defined to answer the question

When have I tested "enough"?

Agreeing on some coverage criterion then measuring how much coverage a test gives is one thing. Another important and more complex thing is to figure out what test cases are needed to achieve good coverage, and then arrange for that automatically. In the given example, that may be simple. The example is tiny, one can see a few boolean conditions and easily figure out inputs that cover each decision as being both true (for one test case) and false (for another). Practically, one may choose the exact corner-cases and then one off, since one should not forget that the *real* goal is not "coverage", the real goes is to make sure that a piece of code has no errors, or rather more realistically: testing should have a better than random chance to detect errors, should there be some. As a matter of fact, one common source of errors is getting the corner cases wrong (like writing $<$ in a conditional instead of $\leq$ or the other way around, especially in loops), which is sometimes called off-by-one error. So, if the code contains a simple, non-compound condition $x > 0$, choosing as input $x = 700$ and $x = -700$ may cover both cases (= 100% edge coverage for that conditional), but practically, choosing $x = 1$ and $x = 0$ may be better.

But anyway, to achieve good "coverage" and/or good testing of corner cases, the **real question** is:

How to do that *systematically* and *automatically*? How to generate necessary input for the test-cases to achieve or approximate the chosen coverage criteria?

That in a way a the starting point of *symbolic execution*, which has its origin in testing. As coverage, it's based typically on something more ambitious than edge coverage or some of the refinements of that. It's based on **path coverage**. Path coverage requires that each *path* from the beginning of the procedure till the end is covered. If there are loops, there are infinitely many paths, which explains the mentioned fact, that loops are problematic. The method is called "symbolic" as it's not about *concrete* values to cover all paths (if possible). So, if one has a condition $x > 0$ as before, it's not about choosing $x = 700$ and $x = -700$ (or maybe better $x = 1$ and $x = 0$). Symbolically, one has two situations: simply $x > 0$ and it's negation $\neg(x > 0)$ (which corresponds to $x \leq 0$), i.e., the two possible outcomes of a condition with that conditions corresponds to two *constraints*. It should be noted: even if, in the presence of loops, there are infinitely many paths 100% path coverage does not cover *all reachable states*, as different values can lead to the same path. That means full path coverage is not the same as full verification or model checking.

Programs typically contain more control structure than just one or two conditions. So, symbolic execution just takes *all paths*, each path involves taking a number of decisions along its way, every one either positively or negatively, and collects all constraints in a big conjuction.

There is more to say about symbolic execution as a field, but that's one core idea in a nutshell.

Path coverage is often considered as too ambitious as coverage criterion. Of course, sometimes tests cannot cover 100% of the simpler critera as well. Nodes that belong to dead code cannot be covered. In a unit with dead code, one cannot achieve 100% coverage. But perhaps one should, since indirectly, dead code may be a sign of a problem as well (only one cannot test dead code in a conventional way, and in a way, there may be no point to test it either). In the presence of loops, there are typically *infinitely many paths*. That means, no matter how many test cases one comes up with, the coverage is always 0%, so in this plain form, one cannot use path coverage to measure if one has tested "enough". Note also: the fact that there are infinitely many paths is not the same as saying that the program itself is non-terminating (for some input). The notion of paths (in the context of path coverage) refer to paths through the control flow graph (CFG), which is an abstraction. The paths may or may not correspond to paths through the graph done when *executing* the actual program. That also means, there may be paths in the CFG that are unrealizable, and in particular, all loops in the progam may actually terminate, but that's something one cannot see in the CFG, where one can see just a cycle in the graph.

Let's revisit the small progrom from earlier, from Listing 7.1. Figure 7.1 shows the corresponding control-flow graph. The graphical "design" used in that figure is sometimes called flow-graph, using some conventions. For instance that the condinals are represented by diamond- or rhombus-shaped nodes, the input-nodes by rhomboid etc. For us, those conventions don't matter much (and they may also vary from presentation to presentation). But maybe they help to visualize the notion of control flow graph. Indeed, control-flow graphs are not primarily a visialization, the are often concrete data structures inside a compiler or model checker, and important intermediate representation, serving various analysis, optimization, and code generation purposes.

Here, coverage is defined in terms of paths through the control-flow graph, and thus also (an implementation of) symbolic execution is based on some form of control-flow graph.

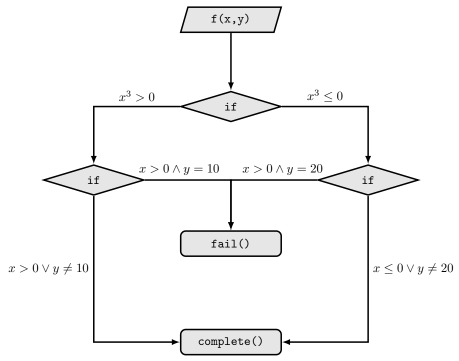
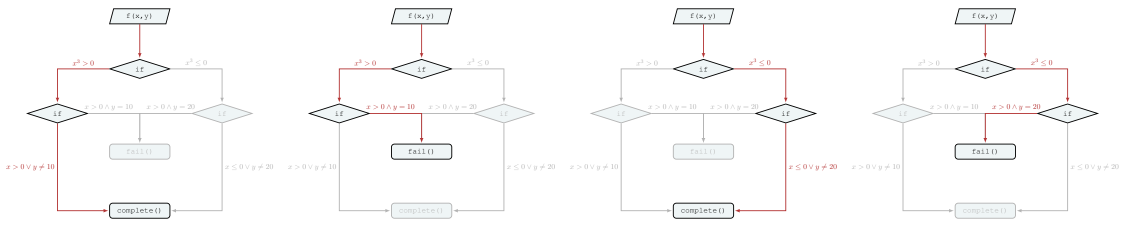


Figure 7.1: Control-flow graph/flow chart

The boolean conditions on the edges correspond to the the condition in case the if- resp. the else-case is taken of the corresponding if-statement. The two branches are mutually explusive (in conventional, deterministic programs), the false-branch is the negation of the true-branch. It should be noted that we assume that there the conditions are are side-effect free. That's generally good programming style, even in case the programming language would support it. Besides, it's of course not real restriction. It's easy to transform "dirty" programs with side-effects in conditions into one that is more disciplined that way. An actually, should the programmer turn a deaf ear on advice like "boolean conditions are better side-effect free", the compiler (or model-checker or analysis tool) will take care of it. Normally, control-flow graphs are not meant for human consumption (unless one uses a graphical programming notation, UML etc), it's an internal representation of the tool, for the purpose of analysis. The flow graph may not even be to represent the control-flow in source code syntax, but perhaps for a lower level intermediate code representation. Keeping that clean helps with whatever one intends to use the CFG for, like code generation, analysis, optimization, etc. Or in our case, symbolic execution, which is a form of analysis anyway. Thus, we it's perfectly fine and realistic to assume the conditions are side-effect free.

The control-flow graph of the program is very simple and there are only 4 different paths from the initial node to one of the terminal nodes. Those four path are shown in Figure 7.2.

Following a path accumulates the conditions as they appear on the positive, resp. the negative edge on the decisions being taken, depending on which decision is assumed the particular path take. If one follows a path from the beginning to the end, one has a boolean constraint which consists of the *conjunction* of all the indidual boolean constraints. One



(a) Path 1 (b) Path 2 (c) Path 3 (d) Path 4

Figure 7.2: Four different paths through the control-flow graph

such constraint is also called the **path constraint** or **path condition** for that particular path.

For instance, the first path, as marked in Figure 7.2a, has the path condition

$$(x^3 > 0) \wedge (x > 0 \wedge y \neq 10) \tag{7.1}$$

and the last path from Figure 7.2d, has the path condition.

$$(x^3 \leq 0) \wedge (x > 0 \wedge y = 20)\ . \tag{7.2}$$

Obviously, the condition for the first path can be simplified, plausibly to $x > 0 \wedge y \neq 10$. The one from equation (7.2) has no solution (in the assumed conventional interpretation on “numbers”). It corrsponds to the constraint “false”. As for terminology: the corresponding path is **unrealizable.**

Figure 7.3 contains all three realizable paths, marked in red. It’s not the same as dead code, but of course one cannot find input that covers that path, as the path condition is contradictory.

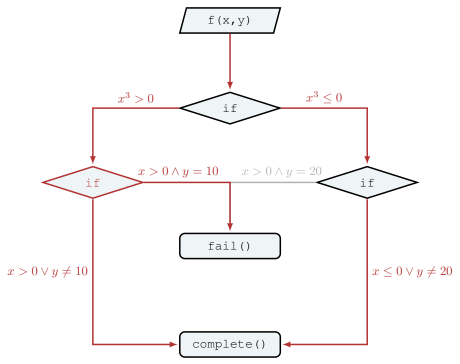


Figure 7.3: All realizable paths

Now the goal is: find a set of inputs to run program so that all *realizable* paths are covered, resp. find a method that automatically does so.

Random testing

Let's start with perhaps the most "naive" way of testing, namely to run the program repeatedly with randomly generated inputs. "Naive" is perhaps an exaggeration, random testing has and is being used, (and studied evaluated and compared to other forms, it has been refined etc.) So it has its place, at least including randomized aspects into testing. We ignore in the discussion here is that in testing one needs among other things, also a way to judge the outcome of the tests. I.e., one needs a specification of the expected outcomes, or at least of the expectation what should or should not happen (like that one does not want to see certain general errors). In the small program from Listing 7.1 that's assumed given by the two possible termination outcomes, either properly completed or failed.

For us important is that testing, randomized or otherwise works with **concrete** input values, and does a **dynamic** execution of programs, observing its *actual* behavior and compare it against *expected behavior*.

Example 7.1.1 (Random testing). Let's use the program from Listing 7.1 resp. its control-flow graph for illustration. Testing means providing different inputs that lead to different outcomes, following potentially different paths, and let's assume the input is generated randomly. For instance, one could use $(x, y) = (700, 500)$, and $(x, y) = (-700, 500)$ . . . The first pair of values results in the path of Figure 7.2a, the second pair in the path of Figure 7.2c, both ending in the non-erroneous end-state, i.e., the two test cases are passed.

The path from Figure 7.2d is unrealizable. But with the two inputs so far, the realizable from Figure 7.2b has so far been missed.

We shouldn't count the unrealizable 4th path one among the ones we missed. But the realizable path shown should be covered. In particular, it's one that would point to an error in the program, the other two so far found no bug.

The problem with this is: to randomly hit that particular path has an astronomically **low probability** (hitting $y = 10$ by chance is very unlikely, indeed). Actually, this way of testing, at least the way of selecting input, may even not even be called *white box*, as it ignores information inside the body of the function, for instance that $y = 10$ seem a profitable corner case. □

In defense of random testing one may say: it may be easy in this particular case, to pick more reasonable or promising input like $y = 10$. That's not just because the program is small. Note in particular, that x and y are also *not updated in* fancy ways (maybe conditionally updated, maybe even using pointers and other complications). One may have to invest heavily in complex theories that may be time-consuming to run before one can get a decent grip on improving on the randomness of the input. And, in a way, *symbolic execution* is an investment in theory (SMT solving) to find an alternative way of testing, thereby also going from a black-box approach for selecting the inputs to a white-box view.

To avoid a mis-conception: random testing is not synonymous with white-box testing. If one does random input testing the way described, and then used path coverage to measure how good the test suites have been, that's *white-box* testing: to *rate* the path coverage, one needs access to the code. It's only that the available white-box information is not taken into account for shaping the test cases in a meaningful way (except for perhaps stop testing, when one feels the random input has achieved sufficient node/edge/path/whatever-coverage).

So, how to get to the missing path from Figure 7.2b? One input that would do the job is, for instance $(x, y) = (145, 10)$, but hitting the concrete value $y = 10$ by chance, as said, as a *very* low probability.

But that's where working with *symbolic representations* can do better, where it's not about individual, concrete values, but sets of values, resp. symbolic or formulaic representations of sets of values. In symbolic executions, one works with path constraints or path conditions. The path condition corresponding to the so far missing path from Figure 7.2b, after simplification, is

$$x > 0 \land y = 10 \ .$$

7.2 Symbolic execution

We have essentially introduced the core idea of symbolic execution in the previous section, focusing on path constraints.

Perhaps it's worth iterating that, like in BMC, it's about *SMT-solving* (not just SAT solving), **sat-solving modulo theories**. That's about

boolean combinations of constraints over specific domains with specific *theories*

The theory or theories allows to express properties of values like integers or arrays, etc., that corresponds to data types used in the programming language used for the programs we are analyzing.

One should be aware, that theories may easily lead to undecidability of constraint solving. Integers with addition only have a decidable theory.[1] Add multiplication, and decidability of the theory goes out the window.

Undecidability is a real issue: how many programs use only integers and addition? One could claim that the programs mostly never use real mathematic integers, but just a finite portion of them (up-to `MAX-INT`) so one is dealing with a finite memory, so that makes properties decidable. That's correct, and when dealing with integers and actual programs, one can make the argument, one should deal with the machine integers anyway to make it more realististic. Indeed, one can work with a theory capturing those "realistic" integers or "IEEE floating points", etc. But all those theories are non-trivial. So even if technically decidable (by being finite), it may be computationally too expensive to wait for an answer when doing SMT solving. And there are more data types than just numbers: there are dynamic data structures (linked lists, trees, etc.), and they are conceptually unbounded, as well. Again, one may posit that, in the real world, there is always some upper bound (`out-of-heap-space`, `stack-overflow`), but it's unrealistic to capture

[1]This specific theory is known as Presburger arithmetic.

those limitations in a decidable theory and hope the constraint solver will handle it thereby. It would even make no sense conceptually, if one is doing “unit testing”: the procedure under test may or may not have out-of-memory problems depending on factors *outside* the unit. For instance on how much heap space is already taken away by other data structure in the program.

Anyway, one has to face the sad fact that one will encounter constraints that are either formally undecidable or untractable; in some way, there’s not much practical difference either way. In some not too far-fetched situations, constraint solving may simply not work.

We come back to that later: **concolic execution** is an extension of symbolic execution that addresses exactly that problem: what can I do if my constraint problem exceeds the capabilities of the used SMT solver. But first we finish up with symbolic execution by looking at a super-simple example, but without adding much new technical content to the material, it’s more like rubbing it in a bit more. One difference to the previous example, though, is that now a variable involved in the program is *assigned* to, i.e., changes its value on the path(s) through the execution

Let’s have a look at the code from Listing 7.2 resp. the CFG from Figure 7.4

```
y = read();
y = 2 * y;

if (y==12) {
   fail();
 }

complete();
```

Listing 7.2: Sample code

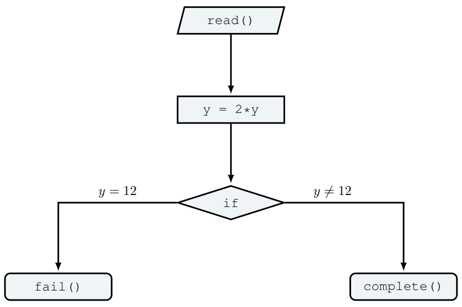


Figure 7.4: CFG

In the C-style code, the “equation” sign of course represents assignments, and the == is a comparison. In the constraints and the annotation on the edges, we use = for comparison, and in the text here, for clarity, we use `:=` for assignments. The difference between assignments and equations should be clear. If not, looking at line 2 of the code snippet: `y := 2 * y` is definitely not the same as the equation $y = 2y$). The latter is unsatisfiable using standard numerical theories.

Let's also introduce the variable s for containing the result of the `read()`-operation. The code contains two asignments, `y := read()` and `y := 2*y`. That leads to two constraints,

$$y = s \quad \text{and} \quad y = 2s$$

at the corresponding points in the program. The branching point in line 4, leads to the two conditions $2s = 12$ and $2s \neq 12$.

As a side remark: we came to the constraint like $y = 2s$ that holds after line 2 by looking at the very simple example. Nore systematic would be to work with different instances or incarnation of the variables. Here with different versions of y, as this is the only variable being asigned to. Actually there are two versions of y, say y_0 after the first assignment and y_1 after the second. I.e., the constraint solver would have to deal with the two constraints

$$y_0 = s \qquad y_1 = 2y_0 \tag{7.3}$$

which is equivalent as far the purpose of the symbolic execution is concerned.

We have seen the same treatment of using different "versions" to represent mutable program variables also in the context of symbolic model checking (where we used unprimed and primed version to capture the pre- and the post-situation in representing the successor states (for the representation of the *pre*- or *post*-sets). And the "versioning" treatment was analogously done for bounded model checking, which also needed to capture paths.

Back to the program. To find, for instance, the erroneous outcome, the onstraint solver needs to solve the path constraint $2s = 12$ (or the slighly longer one from equation (7.3).

That's (in this case) child's play: the solution is $s = 6$.

However, the constraint containts multiplication. We shortly mentioned it before: the theory of natural numbers with addition and multiplication is undecidable.

In this particular example, the constraint is trivially solved by humans, and would pose not problem for constraint solvers. Indeed, the constraint $2s = 12$ is covered by a decidable theory, namely a restriction of the general case of addition and multiplication, where multiplication is restricted to involve only one variable multiplied with constants (so constraints like $xy > 0$ and also $x \times x = 23$ would violate that restriction).

A constraint like $2x + 17y < z$, using an inequation instead of equality, would still be ok: there are 2 variables but they are not multiplied *with each other*. Such restricted forms can be covered by *linear arithmetic*, which has a decidable theory. It's an important class of constraints. For strange historical reason, the field dealing with such (in)equations (and generalizing the question of satisfiability to the question of finding an *optimal solution*) is called *linear programming*. It's also know under the less strange name of *linear optimization*.

Here is a short (intermediate) summary of what's been said, symbolic execution for dummies. It works like this: take the code (resp. the CFG of the code), collect all paths into **path conditions**. A path condition is a big conjunctions of all conditions along each the path. Each single condition b will have one positive mention b in one continuation of the path, one negated mention $\neg b$ in the other continuation.

Then feed the contraints to an appropriate SMT constraint solver, in particular solve the constraints for paths leading to errors. The whole approach works best for loop-free programs (and we will not cover what could be done for loops).

Even if one leaves out loops, which are problematic and focuses on straight-line code, the path constraint themselves may not easily solvable.

Looking again at the code from Listing 7.1 and the paths through the CFG, the path constraints mention x^3 as part of their path constraints. In particular also for the realizable path from Figure 7.2b and the unrealizable one from Figure 7.2d.

With the numerical constraints non-linear, we are definitely leaving the safe ground of decidable theories. Many contstraint solvers would throw the towel when facing those, for instance by only accept linear constraints in numerical domains, or under other restriction. Most solvers would not be ready to deal with random math constraints.

What can one do?

What can one do, beyond throwing the towel and accept that SE won't cover all paths or won't work on many programs? Later we will cover so-called concolic testing, but that is only one possible way to address the limitations of constraint solvers.

First make some remarks on *other* ways, as well, even if we don't cover then. One thing one could do is involve humans in some way, in the spirit of theorem proving. Theorem provers typically can do a lot more than guiding a human through manual proof activities. There is a good deal of automation under the hood, including constraint solving and verification in many domains. And even if undecidable, one could give it a shot, maybe relying on heuristics that in practice can handle many situations. But still, any method that involves human assistance in logical argumentation in formal theories is probably hard to sell in most areas and unappealing for large programs. For most areas a technique is either automatic, or unused . . .

One can also give up on the goal of full path coverage. Most testing approaches don't do try that anyway. Random testing that we touched upon makes not attempt in the direction of any guaranteed coverage, path coverage or otherwise. The problem is, if one is after some form of path coverage, in the face of astrononomically many path (or infinitely many), one in practice covers approximately 0% of all paths, even if one invests is a huge amount of test cases. Zero percent sounds worse than it maybe is; after all, it's not coverage one is after, it's about getting the software right, resp. spotting errors or faults (and then repairing them). A particular defect or its symptom may well not be reached by exactly one path, which one hits or misses, but by very many. Besides, there are heuristics one could use, one could check standard corner cases inherent in the input data or, if one has that, corner cases in the *specification* of the unit one tests. That then goes in the direction of *white box* testing, since the test selection is done on the input-output data, and that can be done without having access to the internals of the code or the CFG.

There is another, standard thing one can do, namely working with abstractions.

Abstraction and “static analysis” Abstracting away from details (in a systematic way) allows to cover *all* possible behaviors. The price of that is that one looses precision.

The presentation here presented SE as a way to systematically represent possible paths via path conditions. The representation of the paths is assumed *precise* but collecting exactly the boolean conditions along the way. It’s only we run into trouble when solving them. Static analysis characteristically works with techniques like data flow analysis (or more generally abstract interpretation or type system in way that systematicall *approximates* the concrete behavior. One (typically) does not attempt to capture *precisely* which choices of values lead to which paths. Instead, one works with approximations (of the values) but does not attempt to tailor-make the abstractions such that they fit exactly the paths.

In a way, the treatment in symbolic execution works on abstractions, as well. The values of the input space are carved up. As far as the values for y are concerned, they are grouped into two classes: all the values where $y = 10$ and all the values $y \neq 10$. One can see that as having two abstract values for y, one consisting of $\{10\}$ and one of the set $\mathbb{N} \setminus \{10\}$. That they are represented “symbolically” with “formulas” or constraints is more a matter of perspective. But SE is based on the idea that the abstraction is sculpted by the need to “steer” the abstract execution along all possible paths (at least those which are realizable), and that works fine as long as there are only finitely many such path.

What an approximating analysis on the other hand does is to assume that it can go *either way*, but without remembering which way it goes, just running the analysis approximately (the technical terms is that the analysis is “path insensitive”). There is more that distinguishes data flow analysis from SE. One is that often the purpose is different. In data flow analysis, the purpose is often not to split up the input of a procedure to get good coverage for testing (though it’s a legitimite goal as well). Instead, one analyses (often in the context of a compiler) other aspects of the code. Therefore, even if one is as radical as representing variables like x and y just by the knowledge that they are integers, one typically adds additional information related to what one is interested in (for live variable analysis, some information about when the variables is assigned to, for analysis of nil-pointer problems, when pointer variables get a proper value etc). And typically that is done also not just for input variables of a procedure, but for all variables or other entities one is interested to analyze. In any case, static analysis like data flow analyses are typically not path-sensitive (as explained). Path sensitivity is not fundamentally forbidden, it’s just too expensive to do in many applications. As a consequence, such analyses are less precise, i.e., more approximative. In doing so, problems with undecidablity may disappear thanks to working with abstractions, and loops no longer pose a problem, at least not as serious as for SE.

One way to see analyses like data flow analysis is not to work with abstractions that exactly cover all combinations of “true” and “false” for all encountered conditions. The abstraction is done *independent* from that. In the simplest case (with the most radical abstraction), one could completely ignore the concrete value (perhaps just abstracting it into its type, like `int`). Obviously, when encountering a condition mentioning the comparison $y = 10$, the analyser would not know if the run goes left or right in that case. One might also split into 3 different abstract values, maybe $\{\textit{negative}, 0, \textit{positive}\}$, hoping that this is a good choice, but the choice is independent from the conditions in the program.

The *borderline* between SE and static analysis is, however, not clear cut. For instance, one could do the following: one can replace constraints beyond the capabilities of the chosen SMT solver (like the one involving x^3) by a constraints in *linear arithmic.* Sometimes one can approximate non-linear constraints by linear one. That way, one can no longer have the exact correspondance between the paths and solutions of the path constraints, therfore it becomes a but like (other) static analyses.

So, isn't SE not a static analysis, as well? It sure is, in that it analyses statically the code. Why it's presented here as being slightly different is its motivation: it's part of a more advanced *testing* approach, which is not a static analysis. Testing is *run-time* or *dynamic* analasys. But it's fair to see SE in the presentation here as a static analysis technique used to improve the run-time technique of testing.

7.3 Concolic testing

"Concolic" is a portmanteau work, mixing together the words "concrete" and "symbolic". Another name for the technique is also DSE, **dynamic symbolic execution**. The presentation here covers the approach as realized by the Dart-tool, which introduced the idea [16]. Since it's introcution, the tool and technique evolved further, as well as the acronyms of the tool(s).

It rests combination of two techniques: a) Random testing, which works with dynamic executions involving concrete values and b) symbolic execution, which works statically and with symbolic constraints and formulas. The slogan of the approach is:

Execute dynamically & explore symbolically

In the following we show in a series of figures, how Dart combines random testing and symbolic execution into a *concolic execution* framework. In the slide versions, the exploration of the different path is shown stepwise, in overlays, which illustrate the interplay between the dynamic execution and the symbolic exploration of alternatives . The overlays are not reproduced here.

The example is taken from Section 2.5 from Godefroid et al. [16] and shows how to handle the program from before (Listing 7.1), which involves non-linear constraints. The non-linear constraint there is meant as an example of a constraint that can't be handled by the chosen SMT solver. Often those focus on decidable theories (like linear constraints).

Also standard *over-approximation* techniques ("predicate abstraction") may not be able to precisely analyze a program like that. They would be unable to figure out that a fail state is unreachable taking the path "via the right-hand side" from Figure 7.2d), i.e., unable to pinpoint unrealizable path. The best they would do is that it "may be reachable", thus reporting an error that is actually not possible. The overapprixmation thus leads to *false alarms.* False alarms are problematic if the user drowns in them. The "tester" may have no patience to inspect thousands of warnings, most of which are just false alarms. So, the tool may become unhelpful if the approximation is too coarse. Complex programming structures, especially wild pointer manipulations and spaghetti code, but also *dynamic aspects* such as higher-order functions, dynamic or late binding etc. confuses not just the programmer but also lead to wildely approximative (= unusable) results. Things get worse when adding concurrency to the mix ...

For the example. Figure 7.5 shows a possible first run of the Dart tool. It starts like random testing, picking an random input, say

$$(x, y) = (700, 500) \; . \tag{7.4}$$

This input leads to the path marked in red in Figure 7.5. Of course, picking exactly those two numbers is highly improbable, but picking an x larger than 0 and $y \neq 10$ has a probability of almost 50%. Of course since it's random, Dart may alternatively start off by choosing the input that leads to the path to completions on the right-hand side, which has a probability of likewise 50%. Only the third possible path, stumbling directly across the error by picking $x > 0$ and $y = 10$ is highly unlikely. Anyway, we start as shown in Figure 7.5.

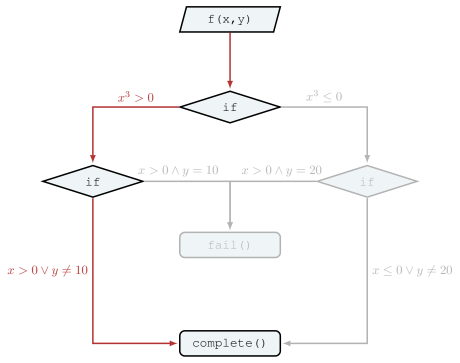


Figure 7.5: Dart (1) (same as Figure 7.2a)

While doing the concrete test run with that input, two boolean conditions have been evaluated to true: $x^3 > 0$ and $y \neq 10$. Those are the path conditions corresponding the path randomly picked. Now, with the goal of path coverage in mind: one should continue with exploring *alternatives*, i.e., explore paths behind the *negation* of those conditions. The negation of the first one is $x^3 \leq 0$. That's a non-linear constraint, i.e., one where a typical SMT solver may chicken out.

So assume Dart does not attempt to do any constraint solving here. Remember the goal: we want to find more or less systematically all paths, but we don't want to overapproximate; we don't want to include unrealizable paths as the might result in false alarms. As we cannot find the *alternative* route at this point in the chosen path by *solving* $x^3 \leq 0$. the only thing we can do at this point is to *use* the path we know that exists as fall-back. That's the path we are currently pursuing, which "solves" the constraint $x^3 > 0$ in having the concrete 700 as one solution.

So we use the *concrete* execution as witness to find one witness solving a constraint we cannot otherwise solve via SMT (more precisely, when we cannot solve its negation, but that amounts generally to the same). In that particular example, we add $x_1 = 700$ as constraint (let's write x_1 when referring to the x in the first run). Now we continue the run with the next conditional. With y picked as 50, the condition $y \neg = 10$ is true. In this

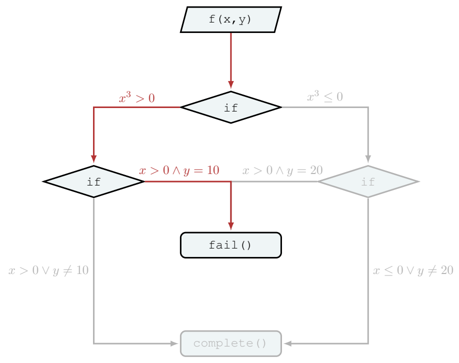


Figure 7.6: Dart (2) (same as Figure 7.2b)

case, the the negation is $y = 10$ which is perfectly solvable (actually: a constraint of that form, equating a variable with a concrete, constant value is a constraint *in solved form*). That's good, so constraint "solving" gave us that $y = 10$ would lead to a *different* path.

So sum up the first run: the randomly generated input from equation (7.4) led to the *concrete execution* from Figure 7.5, and a constraint system of the form

$$(x_1, y_1) = (700, 10)$$

The x_1 is the concrete value in this run, the constraint for y_1 comes from symbolically representing the corresponding alternative in that run (it so happens in the example that the constraint is already in a form ($y_1 = 10$) that has only one solution.

This is the starting point for the *second* run of the method, which is shown in Figure 7.6.

Applying the same method as in the first run, x has the same problem as before, which means we need to use the concrete value 700 as fall-back. That leads to the constraint

$$(x_1 = 700) \wedge (y_2 \neq 10) \ .$$

However, that corresponds to a path already explored. Consequently, after the second run (in this example), *no new inputs are generated.*

If we don't have clear direction (in the form of constraints) what input to take next, we can of course generate a new one randomly. That obviously may result in path already explored. However, in the example, the portion of the graph not yet explored so far is the right-hand side. Sooner or later, the random input generation will pick an input with $x \leq 0$, which explores that part. And actually, it will happen rather sooner than later, let's assume, at iteration n. For concreteness, let's assume the concrete input is

$$(x, y) = (-700, 500) \ .$$

That leads to an execution covering the path from Figure 7.7. The symbolic part chickens out on the first constraint which involves x^3 (besides that the left-hand alternative $x_n^3 > 0$ is already explored), so we have the concrete value $x_n = -700$.

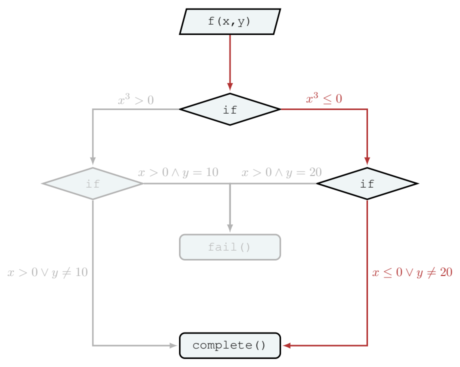


Figure 7.7: Dart (n) (same as Figure 7.2c)

The conditional leads to the additional constraint $x_n > 0 \wedge y = 20$, but that means we have

$$x_n = -700 \quad \wedge \quad x_n > 0 \quad \wedge \quad y = 20 \tag{7.5}$$

which is unsatisfiable. By general reasoning involving the non-linear term x^3, we were aware that this path is unrealizable for any choice of x. The SMT solver may be too weak to draw that conclusion, but at least it will never explore that path, since when the symbolic execution does not work, it relies on *concrete* executions, and those never take that path. So: *no false alarms!*

At that point, we cannot generate new paths any more, all 3 possible paths are covered and the one unrealizable was "covered" insofar that it has been half-symbolically and half-concretely evaluated (see equation (7.5)). So, when figuring out that, the method *stops* generating new tests, having achieved (in this example) the best possible path coverage without generating false alarms.

One can convince oneself, that even with alternative random picks, for instance starting to explore the right-hand side instead of the left hand side as in this illustation, the result would be the same. So with very high probablity (and in short time), the method will achieve that coverage.

Side remark 7.3.1. The example, taken from [16], serves to illustrate in which way the combination of symbolic and concrete execution improved on both plain random testing, symbolic execution, and on approximative methods: it is highly improbably that random testing find the bug, symbolic execution cannot handle the example, and overapproximation give false alarms. Hurrah for concolic execution!

But, on second thought, the example is hand-crafted with the intention to "prove" the superiority of that methods over some competitors. But is it wholly convincing? Well, it worked convincingly enough in the example, in particular stressing the high probablity of covering all realizable paths in a short amount of time.

But that may depend on the (perhaps too cleverly) constructed example. There are two integer input domains: the one for x and the one for y. The one for x is divided 50-50, namely for $x \leq 0$ and $x > 0$. The other domain is *split in an extremely uneven way*: $y = 10$

vs. $y \neq 10$. In both cases the split of the domains correspond to different paths that need to be covered. The SMT solver cannot tackle the *even* split domain for x, as it is written in the form $x^3 \leq 0$ and $x^3 > 0$. The *uneven* split for y, luckily, can be represented by linear constraint and the symbolic treatment can therefore cover the two choices very fast. The even coverage can, with high probability, be covered quite fast by random generation.

If we would have written $y^2 \neq 100 \wedge x > 0$ instead of $y = 10$, the DART method would struggle as well.

So, the example should be read as illustration, in aspects one can hope to improve of the other approaches. Whether it in practice is a step forward can be judged only by applying a corresponding tool to real example programs. Besides that, it also depends on practical issued (which kind of theories should be reasonably covered by the SMT, what data structures does the programming language support, what about external variables and external procedure call etc). The paper [16] reports on experimental evaluation of their approach, providing evidence that the method gives quite added value compared to pure random testing, but they also point out problems of the method in practice

It should also be said, that DART is not the only attempt to improve "stupid random testing" by similar ideas (also before that particular paper). $\square$

Bibliography

[1] Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley.

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

[3] Baldoni, R., Coppa, E., D'Ella, D. C., Demetrescu, C., and Finocchi, I. (2018). A survey of symbolic execution techniques. *ACM Computing Survey*, 51(3).

[4] Biere, A. (2009). Bounded model checking. In Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press.

[5] Bowen, J. P. and Hinchey, M. G. (1995). Seven more myths of formal methods. *IEEE Software*, 12(3):34–41.

[6] Bowen, J. P. and Hinchey, M. G. (2005). Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 8–16, New York, NY, USA. ACM Press.

[7] Büchi, J. R. (1960). Weak second-order arithmentic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92.

[8] Büchi, J. R. (1962). On a decision method in restricted second-order logic. In *Proceedings of the 1960 Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press.

[9] Clarke, E., Kroening, D., Ouaknine, J., and Strichman, O. (2005). Computational challenges- in bounded model checking. *Software Tools for Technology Transfer (STTT)*, 7(2):174–184.

[10] Clarke, E. M. (2008). Model checking – my 27-year quest to overcome the state explosion problem. In Cervesato, I., Veith, H., and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning: 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, Lecture Notes in Artificial Intelligence, pages 182–182. Springer Verlag.

[11] Clarke, E. M. (2017). SAT-based bounded and unbounded model checking. Available electronically on the net. Data of publication unknown.

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

[13] Dijkstra, E. W. (1969). Notes on structured programming. Technical Report EWD-249, Technological University, Eindhoven.

[14] Etessami, K., Wilke, T., and Shuller, R. (2001). Fair simulation relations, parity games, and state space reduction for Büchi automata. In *Proceedings of ICALP 2001*, volume 2076 of *Lecture Notes in Computer Science*, pages 153–167. Springer Verlag.

[15] Garfinkel, S. (2005). History's worst software bugs. Available at `http://archive.wired.com/software/coolapps/news/2005/11/69355?currentPage=all`.

[16] Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: Directed automated runtime testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM.

[17] Goré, R., Heinle, W., and Heuerding, A. (1997). Relations between propositional normal modal logics: an overview. *Journal of Logic and Computation*, 7(5):649–658.

[18] Hall, J. A. (1990). Seven myths of formal methods. *IEEE Software*, 7(5):11–19.

[19] Harel, D., Kozen, D., and Tiuryn, J. (2000). *Dynamic Logic*. Foundations of Computing. MIT Press.

[20] Holzmann, G. J. (2003). *The Spin Model Checker*. Addison-Wesley.

[21] King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.

[22] Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143.

[23] Latvala, T., Biere, A., Heljanko, K., and Junttila, T. (2004). Simple bounded ltl model checking. In Hu, A. J. and Martin, A. K., editors, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04), Austin, Texas, USA, November 2004.*, volume 3312 of *Lecture Notes in Computer Science*. Springer Verlag. An extended version is available as University of Helsinki Technical Report UT-TCS-A92.

[24] MacKenzie, D. A. (2001). *Mechnanizing Proof*. MIT Press.

[25] Manna, Z. and Pnueli, A. (1992). *The temporal logic of reactive and concurrent systems—Specification*. Springer Verlag, New York.

[26] McConnell, S. (2004). *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA.

[27] Peled, D. (2001). *Software Reliability Methods*. Springer Verlag.

[28] Peled, D. (2018). Partial-order reduction. In Clarke, E. C., Henzinger, T. A., Veith, H., and Bloem, R., editors, *Handbook of Model Checking*. Springer Verlag.

[29] Pygott, C. (2019). The VIPER microprocessor. In *2019 6th IEEE History of Electrotechnology Conference (HISTELCON)*, pages 14–19.

[30] Schneider, K. (2004). *Verification of Reactive Systems*. Springer Verlag.

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

Index