



# Course Script

## IN 5110: Specification and Verification of Parallel Systems

IN5110, autumn 2019

Martin Steffen, Volker Stolz

# Contents

<b>1</b>	<b>Formal methods</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	Motivating example . . . . .	1
1.2.1	A simple computational problem . . . . .	1
1.2.2	A straightforward implementation . . . . .	2
1.2.3	The solution (?) . . . . .	2
1.2.4	Should we trust software? . . . . .	3
1.3	How to guarantee correctness? . . . . .	4
1.3.1	Correctness . . . . .	4
1.3.2	How to guarantee correctness? . . . . .	5
1.3.3	Validation & verification . . . . .	6
1.3.4	Approaches for validation . . . . .	6
1.4	Software bugs . . . . .	7
1.4.1	Sources of errors . . . . .	7
1.4.2	Errors in the SE process . . . . .	8
1.4.3	Costs of fixing defects . . . . .	9
1.4.4	Hall of shame . . . . .	9
1.5	On formal methods . . . . .	11
1.5.1	What are formal methods? . . . . .	11
1.5.2	Terminology: Verification . . . . .	11
1.5.3	Limitations . . . . .	12
1.5.4	Any advantage? . . . . .	12
1.5.5	Another netfind: “bitcoin” and formal methods :-)	14
1.5.6	Using formal methods . . . . .	14
1.5.7	Formal specification . . . . .	14
1.5.8	Proving properties about the specification . . . . .	15
1.5.9	Formal synthesis . . . . .	15
1.5.10	Verifying specifications w.r.t. implementations . . . . .	16
1.5.11	A few success stories . . . . .	16
1.5.12	Classification of systems . . . . .	16
1.5.13	Classification of systems: architecture . . . . .	16
1.5.14	Classification of systems: type of interaction . . . . .	17
1.5.15	Taxonomy of properties . . . . .	17
1.5.16	When and which formal method to use? . . . . .	17
1.6	Formalisms for specification and verification . . . . .	18
1.6.1	Some formalisms for specification . . . . .	18
1.6.2	Some techniques and methodologies for verification . . . . .	19
1.7	Summary . . . . .	19
1.7.1	Summary . . . . .	19
1.7.2	Ten Commandments of formal methods . . . . .	20
1.7.3	Further reading . . . . .	20
<b>2</b>	<b>Logics</b>	<b>21</b>
2.1	Introduction . . . . .	21

2.2	Propositional logic . . . . .	25
2.3	Algebraic and first-order signatures . . . . .	26
2.4	First-order logic . . . . .	30
2.4.1	Syntax . . . . .	30
2.4.2	Semantics . . . . .	31
2.4.3	Proof theory . . . . .	33
2.5	Modal logics . . . . .	35
2.5.1	Introduction . . . . .	35
2.5.2	Semantics . . . . .	37
2.5.3	Proof theory and axiomatic systems . . . . .	45
2.5.4	Exercises . . . . .	48
2.6	Dynamic logics . . . . .	50
2.6.1	Multi-modal logic . . . . .	50
2.6.2	Dynamic logics . . . . .	51
2.6.3	Semantics of PDL . . . . .	53
2.6.4	Exercises . . . . .	55
<b>3</b>	<b>LTL model checking</b>	<b>57</b>
3.1	Introduction . . . . .	57
3.2	LTL . . . . .	58
3.2.1	Syntax . . . . .	59
3.2.2	Semantics . . . . .	59
3.2.3	Illustrations . . . . .	63
3.2.4	Some more illustrations . . . . .	64
3.2.5	The Past . . . . .	64
3.2.6	Examples . . . . .	65
3.2.7	Classification of properties . . . . .	70
3.2.8	Exercises . . . . .	76
3.3	Logic model checking: What is it about? . . . . .	76
3.3.1	The basic method . . . . .	76
3.3.2	General remarks . . . . .	79
3.3.3	Motivating examples . . . . .	80
3.4	Automata and logic . . . . .	80
3.4.1	Finite state automata . . . . .	80
3.4.2	Büchi Automata . . . . .	90
3.4.3	Something on logic and automata . . . . .	93
3.4.4	Implications for model checking . . . . .	98
3.4.5	Automata products . . . . .	100
3.5	Model checking algorithm . . . . .	109
3.5.1	Preliminaries . . . . .	109
3.5.2	The algorithm . . . . .	110
3.5.3	LTL to Büchi . . . . .	111
3.5.4	Rest . . . . .	115
3.6	Final Remarks . . . . .	115
3.6.1	Something on Automata . . . . .	115
<b>4</b>	<b><math>\mu</math>-calculus model checking</b>	<b>117</b>
4.1	Introduction . . . . .	117

4.2	Propositional $\mu$ -calculus: syntax and semantics . . . . .	124
4.2.1	Syntax . . . . .	124
4.2.2	Background: Fixpoints . . . . .	130
4.2.3	Semantics . . . . .	133
4.3	Model checking . . . . .	134
<b>5</b>	<b>Partial-order reduction</b>	<b>138</b>
5.1	Introduction . . . . .	138
5.2	Independence and invisibility . . . . .	146
5.3	POR for $LTL_{\neg\bigcirc}$ . . . . .	148
5.3.1	Calculating the ample sets . . . . .	161
<b>6</b>	<b>Symbolic execution</b>	<b>165</b>
6.1	Introduction . . . . .	165
6.1.1	Testing and path coverage . . . . .	168
6.2	Symbolic execution . . . . .	171
6.3	Concolic testing . . . . .	176

# Chapter 1

## Formal methods

### 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).

### Contents

1.1	Introduction . . . . .	1
1.2	Motivating example . . . . .	1
1.3	How to guarantee correctness? . . . . .	4
1.4	Software bugs . . . . .	7
1.5	On formal methods . . . . .	11
1.6	Formalisms for specification and verification . . . . .	18
1.7	Summary . . . . .	19

What is it about?

## 1.1 Introduction

This is the “script” or “handout” version of the lecture’s slides. It basically reproduces the slides in a more condensed way but with additional comments added. The slides used class are kept not too full. Additional information and explanations that are perhaps said in the classroom or when using the whiteboard, without being reproduced on the shown slides, are shown here, as well as links and hints for further readings. In particular, *sources* and *bibliographic information* is shown mostly only here.

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

## 1.2 Motivating example

### 1.2.1 A simple computational problem

$$a_0 = \frac{11}{2}$$

$$a_1 = \frac{61}{11}$$

$$a_{n+2} = 111 - \frac{1130 - \frac{3000}{a_n}}{a_{n+1}}$$

Thanks to César Muñoz (NASA, Langley) for providing the example (which is taken from “Arithm’etique des ordinateurs” by Jean Michel Muller. See <http://www.mat.unb>).

[br/ayala/EVENTS/munoz2006.pdf](http://br/ayala/EVENTS/munoz2006.pdf) or <https://hal.archives-ouvertes.fr/enst-00086707>. The definition or specification of it seems so simple that it's not even a "problem". It seems 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 you 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).

### 1.2.2 A straightforward implementation

```

1 public class Mya {
2
3     static double a(int n) {
4         if (n==0)
5             return 11/2.0;
6         if (n==1)
7             return 61/11.0;
8         return 111 - (1130 - 3000/a(n-2))/a(n-1);
9     }
10
11     public static void main(String [] argv) {
12         for (int i=0;i<=20;i++)
13             System.out.println("a(" +i+" ) = "+a(i));
14     }
15 }

```

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)));;

```

### 1.2.3 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
```

One can easily test the program for the shown output (in the document here, every second line is omitted). It's also not a feature of Java. For instance, a corresponding ocaml program shows “basically” the same behavior (the exact numbers are slightly off).

### 1.2.4 Should we trust software?

$a_n$  for any  $n \geq 0$  may be computed by using the following expression:

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

Where

$$\lim_{n \rightarrow \infty} a_n = 6$$

We get then

$$a_{20} \approx 6 \tag{1.1}$$

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 field 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 (in equation (1.1)). 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 may be specified by a bunch of equations, maybe describing 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 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 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 from some perspective can be seen as quite high-level. Then the task would be, to implement a piece of hardware, or firmware or some controller, that "implement" the specification, given by some high-level recursive description in Java (or some other executable format). One can imagine that the Java program is used for *testing* whether that 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.

## 1.3 How to guarantee correctness?

### 1.3.1 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 be conform with the previous equation

(incl.  $\lim_{n \rightarrow \infty} a_n = 6$ )

- **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



A “deadly embrace” is the original term for something that is now commonly called *deadlock*. It’s a classical error condition that occurs in concurrent programs. In particular something that cannot occur in sequential program or in a sequential algorithm. It occurs when two processes obtain 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). Similar for the last requirement, using the word “never”.

### 1.3.2 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 Dijkstra (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*?

Dijkstra’s well-known dictum comes from [12]. 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 as soon only programs in the most trivial “programming languages” can be automatically analysed (i.e., if one does not allow general loops or recursion or if one assumes finite memory). For clarity, one should perhaps be more precise, what can’t be analysed. First of all, the undecidability of problems 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 many: 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).

### 1.3.3 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.

### 1.3.4 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”

The quote is from Peled’s book [27].

## 1.4 Software bugs

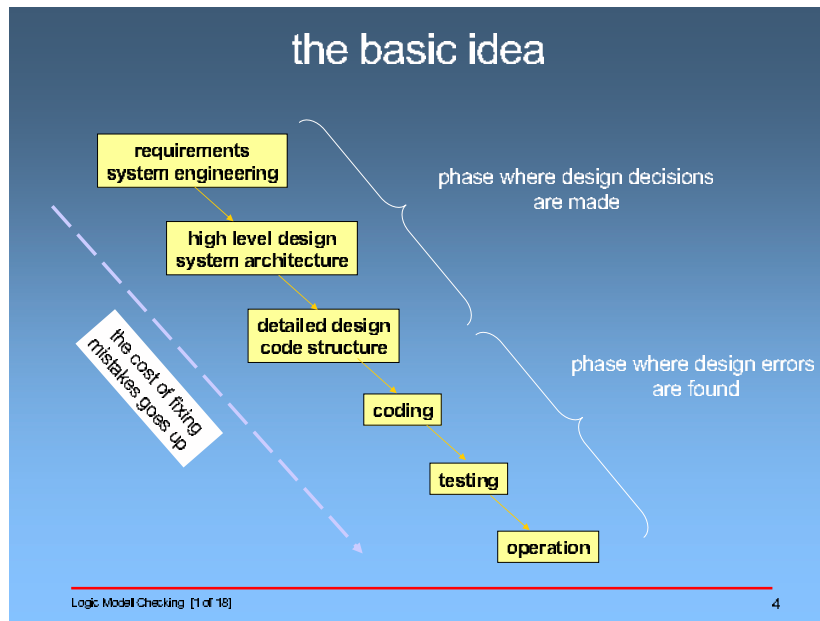
### 1.4.1 Sources of errors

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
- ...

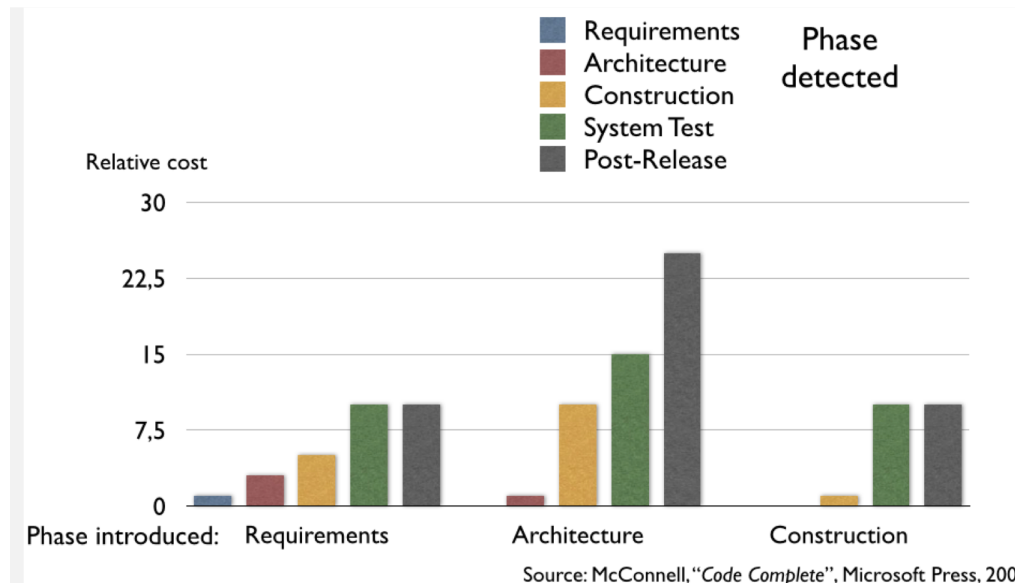
The list of errors is clearly not complete, sometimes a “system” is unusable, even if it’s hard to point with the finger to an error or a bug (is it “a bug or a feature?”). Different kinds of validation and verification techniques address different kinds of errors. Also testing as one (huge) subfield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

### 1.4.2 Errors in the SE process



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 pronounced 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.

### 1.4.3 Costs of fixing defects



The figures are taken from [25] (a quite well-known source). The book itself attributes the shown figures to different other sources.

### 1.4.4 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 direction and was destroyed by the mission control. A software error caused the miscalculation of 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 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 discussed). Note, however, that in some of those cases, the cause of the error is not uncontroversial, 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 when the reasons are unclear are called speculations). Not all one finds on the internet is reliable, there are many folk tales on “funny” software glitches. Many may never see the public light or are openly analysed (especially when concerned with security related issues, military or financial institutions). Of course, when a space craft explodes moments after lift-off or crash-lands on Mars on live transmission from Houston, it’s difficult to swipe it under the carpet. But not even then, it’s easy to nail it down to the/a causing factor not to mention 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 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”.

For instance, 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 machine into punch card. The source code (in the form of punch cards) was, obviously, hard to “read”, so code inspection or code reviews was hard to do at that level. To mitigate the problem of erroneous transcription, machines call *card verifiers* where used. Basically, it meant that two people punched in the same program and verification meant that the result was automatically compared by the verifier.

## 1.5 On formal methods

The slides are inspired by introductory material of the books by K. Schneider and the one by D. Peled ([29, Section 1.1] and [27, Chapter 1]).

### 1.5.1 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)

### 1.5.2 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.

### 1.5.3 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]).

### 1.5.4 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 existence 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 programmers. 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 practitioner 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 artificial 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 search 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, faster computers, but people would still state: a chess playing computer is not intelligent, it’s “just” 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)

### 1.5.5 Another netfind: “bitcoin” and formal methods :-)

	FORMAL SPECIFICATION AND VERIFICATION
Introduction	
Motivation	A significant strength of developing a protocol using a provably correct security model is that it provides a guaranteed limit of adversarial power. One is given a contract that as long as the protocol is followed and the proofs are correct, the adversary cannot violate the security properties claimed.
Sajourn's End	
Proof of Stake	Deeper reflection makes the prior assertion even more significant. Adversaries can be arbitrarily intelligent and capable. To say they are defeated solely through a mathematical model is extraordinary. And, of course, it is not entirely true.
Social Elements of Money	
Designing in Layers – Cardano Settlement Layer	Reality introduces factors and circumstances that prevent the utopia of pure security and correct behavior from existing. Implementations can be wrong. Hardware can introduce attack vectors previously unconsidered. The security model might be insufficient and not conform to real life use.
Scripting	
Sidechains	
Signatures	
User Issued Assets (UIAs)	A judgement call is needed about how much specification, rigor and checking is demanded for a protocol. For example, endeavors like the <a href="#">Sel4 Microkernel project</a> are a prime example of an all out assault on ambiguity requiring almost 200,000 lines of Isabelle code to verify less than 10,000 lines of C code. Yet an operating system kernel is critical infrastructure that could be a serious security vulnerability if not properly implemented.
Scalability	Should all cryptographic software require the same Herculean effort? Or can one choose a less vigorous path that produces equivalent outcomes? Also does it matter if the protocol is perfectly implemented if the environment it runs in is notoriously vulnerable such as on Windows XP?
Cardano Computation Layer	
Regulation	
What is the Point of all of it?	For Cardano, we have chosen the following compromise. First, due to the complex nature of the domains of cryptography and distributed computing, proofs tend to be very subtle, long, complicated and sometimes quite technical. This implies that human driven checking can be tedious and error-prone. Therefore, we believe that every significant proof presented in a white paper written to cover core infrastructure needs to be machine checked.
Science and Engineering	
The Art of Iteration	
Facts and Opinions	
Functional Sins	
Why Haskell?	Second, to verify Haskell code so it correctly corresponds to our white papers, we can choose between two popular options: interfacing with SMT provers via <a href="#">LiquidHaskell</a> and using Isabelle/HOL.
Formal Specification and Verification	SMT (satisfiability modulo theories) solvers deal with the problem of finding functional parameters that satisfy an equation or inequation, or alternatively showing that such parameters do not exist. As discussed by <a href="#">De Moura and Björner</a> , use cases of SMT are various, but the key point is that these techniques are both powerful and can dramatically reduce bugs and semantic errors.
Transparency	
Interoperability	<a href="#">Isabelle/HOL</a> , on the other hand, is a more expressive and diverse tool which can be used to both specify and verify implementation. Isabelle is a generic theorem solver working with higher-order logic constructs, capable of representing sets and other mathematical objects to be used in proofs. Isabelle itself integrates with Z3 SMT prover to work with problems involving such constraints.
The Grand Myopia	
Legacy	
Cryptocurrency Interoperability	Both approaches provide value and therefore we have decided to embrace them both in stages. Human written proofs will be encoded in Isabelle to check their correctness thereby satisfying our machine checking requirement. And we intend on gradually adding Liquid Haskell to all production code in Cardano's implementation throughout 2017 and 2018.
The Maze of Daedalus	
Regulation	
The False Dichotomy	As a final point, formal verification is only as good as the specification one is verifying from and the toolsets available. One of the primary reasons for choosing Haskell is that it provides the right balance of practicality and theory. Specification derived from white papers looks a lot like Haskell code, and connecting the two is considerably easier than doing so with an imperative language.
Metadata	
Authentication and Compliance	
Marketplace DAOs	There is still enormous difficulty in capturing a proper specification and also updating the specification when changes such as upgrades, bug fixes and other concerns need to be made; however, this reality does not in any way diminish the overall value. If one is going to trouble of building a foundation upon provable security, then the implementation should be what was actually proposed on paper.
Sustainability	
Conclusion	
	TRANSPARENCY
	A final question when discussing the science and engineering of developing a cryptocurrency is how to address transparency. Design decisions are not Boolean and ethereal, coming to developers in dreams and then suddenly becoming cannon. They

### 1.5.6 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

### 1.5.7 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

### 1.5.8 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.

- some attempt:

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

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 conjunct should apply also for  $t = 0$ , but that contradicts the first two conjuncts.

So perhaps a correct (?) specification might be

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

### 1.5.9 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)

### 1.5.10 Verifying specifications w.r.t. implementations

Mainly two approaches:

- **Deductive** approach ((automated) theorem proving)
  - Describe the specification  $\varphi_{spec}$  in a formal model (logic)
  - Describe the system's model  $\varphi_{imp}$  in the same formal model
  - Prove that  $\varphi_{imp} \implies \varphi_{spec}$
- **Algorithmic** approach
  - Describe the specification  $\varphi_{spec}$  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  $\varphi_{spec}$

### 1.5.11 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)

...

### 1.5.12 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” [29]. Obviously, one can classify “systems” in many other ways, as well.

### 1.5.13 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

#### 1.5.14 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)

#### 1.5.15 Taxonomy of properties

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

#### 1.5.16 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  $\Rightarrow$  *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.

## 1.6 Formalisms for specification and verification

### 1.6.1 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, ..)
  - $\pi$ -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 referring 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.

### 1.6.2 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

## 1.7 Summary

### 1.7.1 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
- un real complex systems, only part of the system may be formally proved and no single FM can make the task
- our course will concentrate on
  - temporal logic 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)

### 1.7.2 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

### 1.7.3 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 [29]
- 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. [24]



# Chapter 2

## Logics

### Learning Targets of this Chapter

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

### Contents

2.1	Introduction . . . . .	21
2.2	Propositional logic . . . . .	25
2.3	Algebraic and first-order signatures . . . . .	26
2.4	First-order logic . . . . .	30
2.5	Modal logics . . . . .	35
2.6	Dynamic logics . . . . .	50

What  
is it  
about?

## 2.1 Introduction

### Logics

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 a “model”<sup>1</sup> We will 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 seen 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”

<sup>1</sup>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 facing; the rough idea remains the “same” though.

anyway, the installation? The binary? Training? etc.) All of that would be “formalized” in a legal contract readable not even 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, like “is logic as discipline a subfield of math, or is it the other way around”, like “math is about drawing conclusions about some abstract notions and proving things about those, and in order to draw conclusions in a rigorous manner, one should use logical systems (as opposed to hand-waving . . .)”. We are also mostly not much concerned with fundamental questions of *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 the 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 “what exact syntax do you use?”. That only bothers mathematicians (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 the objects of study for formal-method people are, mathematically seen, “dirty stuff”. One tries to argue for the correctness of a program, an algorithm, maybe even an implementation. That often means one does not deal with any elegant mathematical structure but some specific artifact. It’s not about “in principle, the idea of the algorithm is correct”; 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 formalistic argument covering really all cases would not. (Additionally, in making proofs about software, it’s often not about “the remaining 5 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. Proving that a few exemplary interleavings are correct (or test a few) will simply not do the job.

### General aspects of logics

- truth vs. provability
  - when does a formula *hold*, is *true*, is *satisfied*
  - valid
  - satisfiable
- syntax vs. semantics/models
- model theory vs. proof theory

We will encounter different logics. They differ in their syntax and their semantics (i.e., the way particular formulas are given meaning), but they share some commonalities. Actually, the fact that one distinguishes between the *syntax* of a logics and a way to fix the meaning of formulas is common to all the encountered approaches. The term “formula” refers in general to a syntactic logical expression (details depend on the particular logic, of course, and sometimes there are alternative or more finegrained terminology, like *proposition*, or *predicate* or *sentence* or *statements*, or even in related contexts names like *assertion* or *constraint*). For the time being, we just generically speak about “formulas” here and leave terminological differentiations for later. Anyway, when it comes to the semantics, i.e. the meaning, it’s the question of whether it’s true or not (at least in classical settings...). Alternative and equivalent formulations is whether it *holds* or not and whether its *satisfied* or not.

That’s only a rough approximation, insofar that, given a formula, one seldomly can stipulate unconditionally that it holds or not. That, generally, has to do with the fact that formulas typically has fixed parts and “movable” parts, i.e., parts for which an “interpretation” has to be chosen before one can judge the truth-ness of the formula. What exactly is fixed and what is to be chosen depends on the logic, but also on the setting or approach.

To make it more concrete in two logics one may be familiar with (but the lecture will cover them to some extent). For the rather basic *boolean logic* (or propositional logic), one deals with formulas of the form  $p_1 \wedge p_2$ , where  $\wedge$  is a logical connective and the  $p$ ’s here are atomic propositions (or propositional variables, propositional constants, or propositional symbols, depending on your preferred terminology). No matter how it’s called, the  $\wedge$  part is fixed (it’s always “and”), the two  $p$ ’s is the “movable part” (it’s for good reasons why they are sometimes called propositional *variables*...). Anyway, it should be clear that asking whether  $p_1 \wedge p_2$  is true or holds cannot be asked *per se*, if one does not know about  $p_1$  and  $p_2$ , the truth or falsehood is relative to the choice of truth or falsehood of the propositional variables: choosing both  $p_1$  and  $p_2$  as “true” makes  $p_1 \wedge p_2$  true.

There are then different ways of notationally write that. Let's abbreviate the mapping  $[p_1 \mapsto \top, p_2 \mapsto \top]$  as  $\sigma$ , then all of the formulations (and notations) are equivalent

- $\sigma \models \varphi$  (or  $\models_{\sigma} \varphi$ ):
  - $\sigma$  satisfies  $\varphi$
  - $\sigma$  models  $\varphi$  ( $\sigma$  is a model of  $\varphi$ )
- $\llbracket \varphi \rrbracket^{\sigma} = \top$ :
  - with  $\sigma$  as propositional variable assignment,  $\varphi$  is true or  $\varphi$  holds
  - the semantics of  $\varphi$  under  $\sigma$  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 symbols of a chosen signature, however, are generally *not* fixed, at least when doing “logic” and meta-arguments about logics. 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 terminology becomes also a bit non-uniform (depending on the text-book, one might encounter slightly contradicting use of words).

One common interpretation is to call the choice of symbols the *interpretation* or also *model*. To make a distinction one sometimes say, the *model* (let's call it  $M$  is the mathematic structure to part

## 2.2 Propositional logic

A very basic form of logic is known as *propositional* or also *boolean* (in honor of George Boole).<sup>2</sup> 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 the 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, its *model theory* ...) and its *proof theory*. For now, 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 satisfies a formula (its model). Proof theory is about when formulas are "provable" (by a formal procedure or derivation system). Those questions are not independent. A "provable" formula should ideally be "true" as well formula (a question of *soundness*) 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  $\varphi$  is derivable or probable, and  $\models \varphi$  for the formula being "true" (or valid etc.) in a model.

### Syntax

$$\begin{array}{ll} \varphi ::= P \mid \top \mid \perp & \text{atomic formula} \\ \mid \varphi \wedge \varphi \mid \neg \varphi \mid \varphi \rightarrow \varphi \mid \dots & \text{formulas} \end{array}$$

As it is common (at least in computer science), the syntax is given by a grammar. More precisely here, a context-free grammar using a variant of 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 as used in a text file used in a computer. There, more details would have to be fixed to lay down a precise 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. A context-free grammar *is* precise,

<sup>2</sup>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. One can also study intuitionistic versions. One of such logic is known as *minimal* intuitionistic logic, that has implication  $\rightarrow$  as the only constructor.

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 in 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.

## Semantics

- truth values
- $\sigma$
- different “notations”
  - $\sigma \models \varphi$
  - evaluate  $\varphi$ , given  $\sigma \llbracket \varphi \rrbracket^\sigma$

The semantics or meaning of a boolean formula is fixed by defining when it holds. More precisely: a formula typically contains propositional symbols  $p_1, q'$ , etc, whose value need to be fixed (to true or false for each of them). Assuming that we have such a set  $AP$  of propositional variables, then a choice of truth values can be called a *variable assignment*. We use the symbol  $\sigma$  for those, i.e.

$$\sigma : AP \rightarrow \mathbb{B}$$

## Proof theory

- decidable, so a “trivial problem” in that sense
- truth tables (brute force)
- one can try to do better, different derivation strategies (resolution, refutation, ...)
- SAT is NP-complete

*Truth tables* are probably known: it’s an explicit enumeration (often in tabular presentation) of the result of a formula when fixing all boolean inputs as either true or false. That obviously leads to a table of size  $2^n$ , when  $n$  is the number of atomic propositions (the “input”). That’s “brute force”, but it’s a viable way to calculate the intended function, since there are only finately many inputs.

SAT is a closely related, but different problem. It ask whether *there exists* an satisfying to SAT is not a model checking problem. If we see  $\sigma$  as model, then model checking  $\sigma \models \varphi$  is complexity-wise linear (compositional, divide-and-conquer).

## 2.3 Algebraic and first-order signatures

### Signature

- fixes the “syntactic playground”

- selection of
  - *functional* and
  - *relational*
 symbols, together with “arity” or sort-information

## Sorts

- **Sort**
  - name of a domain (like `Nat`)
  - restricted form of type
- single-sorted vs. multi-sorted case
- single-sorted
  - one sort only
  - “degenerated”
  - *arity* = number of arguments (also for relations)

## Terms

- given: signature  $\Sigma$
- set of variables  $X$  (with typical elements  $x, y', \dots$ )

$$\begin{array}{ll}
 t ::= x & \text{variable} \\
 | f(t_1, \dots, t_n) & f \text{ of arity } n
 \end{array} \tag{2.1}$$

- $T_\Sigma(X)$
- terms without variables (from  $T_\Sigma(\emptyset)$  or short  $T_\Sigma$ ): *ground* terms

The definition here makes use of the simple single-sorted case. The 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 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.

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 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 also write in conventional situations like here  $+$  and  $0$  for the *symbols* as well.

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).

In practical situations (i.e., tools), one could allow *overloading*, or other “type-related” complications (sub-sorts for examples). 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) and assume that educated readers know what is meant if we use more conventional concrete notations.

### Substitution

- **Substitution** = *replacement*, namely of variables by terms
- notation  $t[s/x]$

Other notations for substitution exist in the literature. Here, substitution is defined on terms. We will later use (mutatis mutandis) substitution also on first-order formulas (actually, one can use it 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 consider (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).

### First-order signature (with relations)

So far we have covered standard signatures for terms (also known as algebraic signatures). In first-order logic, one also adds a second kind of syntactic material to the signatures, besides function symbols, those are *relational* symbols. Those are intended to be interpreted “logically”. For instance, in a one-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 the form later then the *atomic* formulas of the first-order logic (also called (first-order) predicate logic).

- add **relational** symbols to  $\Sigma$
- typical elements  $P, Q$
- relation symbols with fixed arity  $n$ -ary predicates or relations)
- standard binary symbol:  $\doteq$  (equality)

**Multi-sorted case and a 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, maybe 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 × nat → bool`. Nothing wrong with



that, but one has to be careful not to 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 the 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 operators 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 are part of the logic itself.

**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 two 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 and 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).

**Equality symbol** 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 to 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  $x \text{ eq } x$  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 conjunction, one always interprets the  $\doteq$  as equality.

As a side remark: the status of equality, identity, equivalence etc is challenging from the standpoint of *foundational* logic or maths. 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 investigating 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.

## 2.4 First-order logic

### 2.4.1 Syntax

#### Syntax

- given: first-order signature  $\Sigma$

$$\begin{array}{l} \varphi ::= P(t, \dots, t) \mid \top \mid \perp \quad \text{atomic formula} \\ \quad \mid \varphi \wedge \varphi \mid \neg \varphi \mid \varphi \rightarrow \varphi \mid \dots \quad \text{formulas} \\ \quad \mid \forall x. \varphi \mid \exists x. \varphi \end{array}$$

The grammar shows the syntax for first-order logic. We are not overly obsessed with concrete syntax (here), i.e., we treat the syntax as *abstract syntax*. 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.

**Minimal representation and syntactic variations** The above presentation, as in the proposition or boolean 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, *true* can be defined using *false*, and vice versa. In the case of the boolean constants *true* and *false*, 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 *true* and *false* as propositions or relations with arity 0 and a fixed interpretation. All of that representation 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 over-obsessed with representational questions like that. Of course, if one had to interact with a tool that “supports” for instance first-order logics (like a theorem prover or constraint solver) or if one wanted to implement such a tool oneself, syntactical questions would of course matter and one would have to adhere to stricter standards of that particular tool.

## 2.4.2 Semantics

### First-order structures and models

- given  $\Sigma$
- assume single-sorted case

#### first-order model model $M$

$$M = (A, I)$$

- $A$  some domain/set
- **interpretation**  $I$ , respecting arity
  - $\llbracket f \rrbracket^I : A^n \rightarrow A$
  - $\llbracket P \rrbracket^I : A^n$
- cf. first-order structure

#### First-order structure (left out from the slide)

- single-sorted case here
- domain  $A$  together with functions and relations
- NB: without relations: *algebraic structure*
- many-sorted case: analogously (interpretation respecting sorts)

A model here is a pair, namely a domain of interpretation together with  $I$ , that associates functions and relations appropriately to the corresponding syntactic vocabulary from the signature. A set, equipped with functions and relations is also called a first-order *structure*. Often, the structure itself is also called the model (leaving the association between syntax and its interpretation implicit, as it's obvious in many cases anyway). For instance,

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

is a structure, whose domain are the natural numbers and which 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  $\geq$ . That can be a “model” of a signature  $\Sigma$  with one sort say `Nat` and function symbols `zero`, `succ`, `plus` and `times` and relational symbols `leq` and `geq`. Technically, though the model is the mapping  $I$ . Strictly speaking, nothing would forbid us to interpret the syntax differently in the same structure, for instance setting  $\llbracket \text{times} \rrbracket^I = +$  or  $\llbracket \text{leq} \rrbracket^I = \geq$ .

In this (and similar) cases the association is obvious, thereby sometimes left implicit, and some people also call the structure a *model* of a signature (but for preciseness sake, it should be the structure *and* making clear which elements of the structure belong to what syntax).

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

illustration here, the natural numbers, may be deceptively 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

#### Variable assignment

- given  $\Sigma$  and model

$$\sigma : X \rightarrow A$$

- other names: *valuation*, *state*

#### (E)valuation of terms

- $\sigma$  “straightforwardly extended/lifted to terms”
- how would one define that (or write it down, or implement)?

Given a signature  $\Sigma$  and a corresponding model  $M = (A, I)$ , the value of term  $t$  from  $T_{\Sigma}(X)$ , with variable assignment  $\sigma : X \rightarrow A$  (written  $\llbracket t \rrbracket_{\sigma}^I$ ) is given inductively as follows

$$\llbracket \varphi \rrbracket_{\sigma}^I$$

#### Free and bound occurrences of variables

- quantifiers *bind* variables
- *scope*
- other binding, scoping mechanisms
- variables can *occur* free or not (= *bound*) in a formula
- careful with substitution
- how could one define it?

#### Substitution

- basically:
  - generalize substitution from terms to formulas
  - careful about binders especially don’t let substitution lead to variables being “captured” by binders

**Example**

$$\varphi = \exists x.x + 1 \doteq y \quad \theta = [x/y]$$

**Satisfaction**

$$\models M, \sigma \models \varphi$$

- $\Sigma$  fixed
- in model  $M$  and with variable assignment  $\sigma$  formula  $\varphi$  is true (holds)
- $M$  and  $\sigma$  satisfy  $\varphi$
- minority terminology:  $M, \sigma$  model of  $\varphi$

In seldom cases, some books call the pair of  $(M, \varphi)$  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 independent from the logic). A formula  $\varphi_1$  implies semantically a formula  $\varphi_2$ , if all models of  $\varphi_1$  are also models of formula  $\varphi_2$  (the satisfaction of  $\varphi_1$  implies satisfaction of  $\varphi_2$ ).

Now it depends in if one applies the word “model” to  $M$  or to the pair  $M, \sigma$ . 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.

**Exercises**

- substitutions and variable assignments: similar/different?
- there are infinitely many primes
- there is a person with at least 2 neighbors (or exactly)
- every even number can be written as the sum of 2 primes

**2.4.3 Proof theory****Proof theory**

- how to infer, derive, deduce formulas (from others)
- mechanical process
- soundness and completeness
- *proof* = deduction (sequence or tree of steps)
- theorem
  - syntactic: derivable formula
  - semantical a formula which holds (in a given model)
- (fo)-theory: set of formulas which are
  - derivable
  - true (in a given model)
- soundness and completeness

## Deductions and proof systems

A *proof system* for a given logic consists of

- *axioms* (or *axiom schemata*), which are formulae assumed to be true, and
- *inference rules*, of approx. the form

$$\frac{\varphi_1 \quad \dots \quad \varphi_n}{\psi}$$

- $\varphi_1, \dots, \varphi_n$  are **premises** and  $\psi$  **conclusion**.

## A simple form of derivation

**Derivation of  $\varphi$**  *Sequence* of formulae, where each formula is

- an axiom or
- can be obtained by applying an inference rule to formulae earlier in the sequence.
- $\vdash \varphi$
- more general: set of formulas  $\Gamma$

$$\Gamma \vdash \varphi$$

- proof = derivation
- theorem: derivable formula (= last formula in a proof)

A proof system is

- *sound* if every theorem is valid.
- *complete* if every valid formula is a theorem.

We do not study soundness and completeness for validity of FOL in this course.

## Proof systems and proofs: remarks

- “definitions” from the previous slides: not very formal

in general: a proof system: a “mechanical” (= formal and constructive) way of **conclusions** from axioms (= “given” formulas), and other already proven formulas

- Many different “representations” of how to draw conclusions exist, the one sketched on the previous slide
  - works with “sequences”
  - corresponds to the historically oldest “style” of proof systems (“Hilbert-style”), some would say outdated . . .
  - otherwise, in that naive form: impractical (but sound & complete).
  - nowadays, better ways and more suitable for computer support of representation exists (especially using trees). For instance **natural deduction** style system

for this course: those variations don’t matter much.

## A proof system for prop. logic

We can axiomatize a subset of *propositional logic* as follows.

$$\begin{array}{ll}
 \varphi \rightarrow (\psi \rightarrow \varphi) & (\text{Ax1}) \\
 (\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi)) & (\text{Ax2}) \\
 ((\varphi \rightarrow \perp) \rightarrow \perp) \rightarrow \varphi & (\text{DN}) \\
 \frac{\varphi \quad \varphi \rightarrow \psi}{\psi} & (\text{MP})
 \end{array}$$

## A proof system

*Example 2.4.1.*  $p \rightarrow p$  is a theorem of PPL:

$$\begin{array}{ll}
 (p \rightarrow ((p \rightarrow p) \rightarrow p)) \rightarrow & \text{Ax}_2 \\
 ((p \rightarrow (p \rightarrow p)) \rightarrow (p \rightarrow p)) & (2.1) \\
 p \rightarrow ((p \rightarrow p) \rightarrow p) & \text{Ax}_1 \\
 (2.2) \\
 (p \rightarrow (p \rightarrow p)) \rightarrow (p \rightarrow p) & \text{MP on (1) and (2)} \\
 (2.3) \\
 p \rightarrow (p \rightarrow p) & \text{Ax}_1 \\
 (2.4) \\
 p \rightarrow p & \text{MP on (3) and (4)} \\
 (2.5)
 \end{array}$$

A proof can be represented as a [tree](#) of inferences where the leaves are axioms.

## 2.5 Modal logics

### 2.5.1 Introduction

The roots of logics date back very long, and those of modal logic not less so (Aristotle also hand his fingers in the origin of modal logic and discussed some kind of “paradox” that gave food for thought for future generations thinking about 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”, i.e., *tomorrow* is a form 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 where long debated often with philosophical and/or even religious connotations 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 of 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

$$\Box\varphi \rightarrow \Box\Box\varphi, \quad (2.6)$$

with  $\varphi$  being some ordinary statement (in propositional logic perhaps, for first-order logic). If we are dealing with a logic of belief, is that a valid formula: If I believe that something is true, do I believe that I believe that the thing is? 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 (2.6): 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  $\varphi$ ". Now we may ponder whether that means that then I *also* know that  $\varphi$ ? 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 philosophical questions. The question whether equation (2.6) 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 super-thoroughly been 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*. Models of modal logics are now called *Kripke-structures* (it's basically transition systems).



## Introduction

- **Modal** logic: logic of “*necessity*” and “*possibility*”, in that originally the intended meaning of the *modal* operators  $\Box$  and  $\Diamond$  was
  - $\Box\varphi$ :  $\varphi$  is necessarily true.
  - $\Diamond\varphi$ :  $\varphi$  is possibly true.
- Depending on what we intend to capture: we can interpret  $\Box\varphi$  differently.
  - temporal**  $\varphi$  will always hold.
  - doxastic** I believe  $\varphi$ .
  - epistemic** I know  $\varphi$ .
  - intuitionistic**  $\varphi$  is provable.
  - deontic** It ought to be the case that  $\varphi$ .

We will restrict here the modal operators to  $\Box$  and  $\Diamond$  (and mostly work with a temporal “mind-set”).

### 2.5.2 Semantics

#### Kripke structures

The definition below makes use of the “classical” choice of words to give semantics to modal logic. It’s actually quite simple, based on a relation (here written  $R$ ) on some set. The “points” on that set are called *worlds* which are connected by an *accessibility* relation. 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 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”: one some world some formulas hold, and in a different world, different ones. “Embedded” in the modal logic is an “underlying” logic. We mostly assume *propositional* modal logic, but one might as well consider an extension of first-logic with modal operators. For instance, the student presentation about *runtime verification* will make use of a first-order variant of LTL, called QTL, quantified temporal logic, but that will be later. In this propositional setting, what then is needed for giving meaning to modal formulas is an interpretation of the propositional variables, and that has to be done *per world*.

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

$$W \rightarrow AP \rightarrow \mathbb{B} .$$

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

**Labelling** The valuation function  $V$  associates a propositional valuation to each world (or isomorphically the set of all propositional atoms that are intended to hold, per 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 as *transitions*. That terminology 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. 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 expressed 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 whether consider transitions as labelled or not, there is a “state-labelling” at least, namely the valuation that is needed to interpret the propositions per world or state.

As a side remark: classical automata can be seen as labelled transitions, 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). There, two classical versions that are used in describing hardware (which is a form of propositional logic as well. . .) label the transitions via 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., transitions contain both 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 as well, but in the form that fits to our modal-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 2.5.1** (Kripke frame and Kripke model).

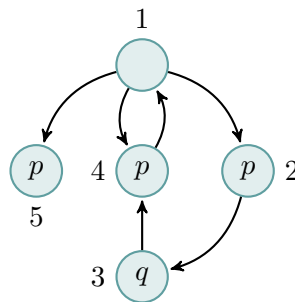
- A *Kripke frame* is a structure  $(W, R)$  where
  - $W$  is a non-empty set of *worlds*, and
  - $R \subseteq W \times W$  is called the *accessibility relation* between worlds.
- A *Kripke model*  $M$  is a structure  $(W, R, V)$  where
  - $(W, R)$  is a frame, and
  - $V$  a function of type  $V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

isomorphically:  $V : W \rightarrow 2^{AP}$

Kripke models are sometimes called *Kripke structures*. The standard textbook about model checking Baier and Katoen [2] does not even mention the word “Kripke structure”, they basically use the word *transition systems* instead of Kripke model with worlds called states (and Kripke frame is called *state graph*). 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  $AP$  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. The usage of “variable” vs. “constant” seem two irreconcilable choices of terminology. Perhaps the use of the word “variable” stresses that the value needs to be fixed by some interpretation, the word “constant” focuses in the fact that those are “0-ary” constructors (or atoms), and this “constant”, not depending on subformulas (once their meaning is fixed). We prefer thinking of  $\top$  and  $\perp$  as the only two propositional constants and we don’t call propositional atoms “constants”.

### Illustration



**Kripke model** Let  $AP = \{p, q\}$ . Then let  $M = (W, R, V)$  be the Kripke model such that

- $W = \{w_1, w_2, w_3, w_4, w_5\}$
- $R = \{(w_1, w_5), (w_1, w_4), (w_4, w_1), \dots\}$
- $V = [w_1 \mapsto \emptyset, w_2 \mapsto \{p\}, w_3 \mapsto \{q\}, \dots]$

The graphical representation is slightly informal (and also later we allow ourselves these kind of “informalities”, appealing to the intuitive understanding of the reader). There are 5 worlds, which are numbered for identification. In the Kripke model, they are referred to via  $w_1, w_2, \dots$  (not as  $1, 2, \dots$  as in the figure). Later, when we often call corresponding entities states, not worlds, we tend to use  $s_1, s_2, \dots$  for typical states. For the valuation, we use a notation of the form  $[\dots \mapsto \dots]$  to denote a finite mappings.

In particular, we are dealing with finites mappings of type  $W \rightarrow 2^{AP}$ , i.e., to subsets of the list of atomic propositions  $AP = \{p, q\}$ . The sets are not explicitly noted in the

graphical illustration, i.e., the set-braces  $\{\dots\}$  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 isomorphic (i.e., equivalent) view on the valuation is, that it is a function of type  $W \rightarrow (AP \rightarrow \mathbb{B})$  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 intended to be evaluate to “false” in that world.

As a side remark: we said, that we are dealing with finite mappings. For the examples, illustrations and many applications, that is correct. However, the definition of Kripke structure does *not* require that there is only a finite set of worlds,  $W$  in general is a *set*, finite or not.

## Satisfaction

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, typically written as  $\models$ . After the introduction and discussion of Kripke models or transition systems, the satisfaction relation should be fairly obvious for the largest part, especially the part of the *underlying logic* (here propositional logic): the valuation  $V$  is made exactly so that it covers the base case of atomic propositions, namely give meaning to the elements of  $AP$  depending on the current world of the Kripke frame. The treatment of the propositional connectives  $\wedge$ ,  $\neg$ ,  $\dots$  is identical to their treatment before. Remains the treatment of the real innovation of the logic, the modal operators  $\Box$  and  $\Diamond$ .

**Definition 2.5.2** (Satisfaction). A modal formula  $\varphi$  is *true* in the world  $w$  of a model  $V$ , written  $V, w \models \varphi$ , if:

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

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

For this lecture, we prefer the former presentation (but actually, it does not matter of course) for 2 reasons. 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 Kripke models more like transition systems, for instance capturing the behavior of computer programs. There, we are not so philosophically interested in speaking of “worlds” that are “accessible” via some

accessibility relation  $R$ , it's more like states in a program, 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  $\varphi$ ”, or “ $\varphi$  holds in state  $w$ ” etc. but of course the interpretation of a modal formula requires that there is always a transition system relative to which it is interpreted.

Often one finds also notations using the “semantic brackets”  $\llbracket \_ \rrbracket$ . Here, the meaning (i.e., truth-ness or false-ness of a formula, depends on the Kripke model as well as the state, which means one could define a notation like  $\llbracket \varphi \rrbracket_w^M$  as  $\top$  or  $\perp$  depending on whether  $M, w \models \varphi$  or not. Remember that we had similar notation in first-order logic  $\llbracket \varphi \rrbracket_\sigma^I$ . We discussed (perhaps unnecessarily so) two isomorphic views 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 (that are 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 supposed to hold, one would have a “valuation”  $\tilde{V} : AP \rightarrow 2^W$ . 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 intuitionistic logics. Kripke also propose “Kripke-models” for that kind of logics (for clarity, I am talking about intuitionistic propositional logics or intuitionistic first-order logic etc, 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  $\varphi$  (in a model)”, for which we write “ $w \models p$ ”, but says “world  $w$  forces a formula  $\varphi$ ”, 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  $\llbracket \varphi \rrbracket^M$  to represent the set of all states in  $M$  that satisfy  $\varphi$ , i.e.,

$$\llbracket \varphi \rrbracket^M = \{w \mid M, w \models \varphi\} .$$

In general (and also other logics),  $\models$ - and  $\llbracket \_ \rrbracket$ -style notations are interchangeable and interdefinable.

### “Box” and “diamond”

- modal operators  $\Box$  and  $\Diamond$
- often pronounced “necessarily” and “possibly”

- mental picture: depends on “kind” of logic (temporal, epistemic, deontic ...) and (related to that) the form of accessibility relation  $R$
- formal definition: see previous slide

The pronunciation of  $\Box\varphi$  as “necessarily  $\varphi$ ” and  $\Diamond\varphi$  as “possibly  $\varphi$ ” are *generic*, when dealing with specific interpretations, they might get more specific meanings and then be called likewise: “in all futures  $\varphi$ ” or “I know that  $\varphi$ ” etc. Related to the intended mindset, one imposes different restrictions in the “accessibility” relation  $R$ . In a temporal setting, if we interpret  $\Box\varphi$  as “tomorrow  $\varphi$ ”, then it is clear that  $\Box\Box\varphi$  (“ $\varphi$  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  $\varphi$ ”, then  $\Box\Box\varphi$  and  $\Box\varphi$  are equivalent. That reflects common sense and reflects what one might think about the concept of “times” and “days”. 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  $\Box\varphi \rightarrow \Box\Box\varphi$  is generally true.

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 (which is a frame with a valuation) 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 valuation (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  $w$ . For that one can write

$$\models \varphi$$

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

$$(W, R) \models \varphi$$

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 “ $\varphi$  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 relation  $R$  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.

It 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 of  $R$  which corresponds to a particular set of axioms traditionally known as S4. See below for some examples.

**Further notational discussion and preview to LTL** Coming back to the informal “temporal” interpretation of  $\Box\varphi$  as either “tomorrow  $\varphi$ ” vs. “now and in the future  $\varphi$ ”, 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. A conventional way would be to see the time as *linear* (a line, fittingly called a *timeline*, connecting the past into the future, with “now” in the middle, perhaps measured in years or seconds etc.) With such a linear picture in mind, it’s also clear that there is no difference between the modal operators  $\Box$  and  $\Diamond$ .<sup>3</sup> In the informal interpretation of  $\Box$  as “tomorrow”, one should have been more explicit that, what was meant is “for all 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 takes  $\bigcirc$ . A formula  $\bigcirc\varphi$  is interpreted as “in the next state or world,  $\varphi$  holds” and pronounced “next  $\varphi$ ” 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)  $\square$  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 have fixed for now, when discussing modal logics in general).

### Different kinds of relations

$R$  a binary relation on a set, say  $W$ , i.e.,  $R \subseteq W \times W$

- reflexive
- transitive
- (right) Euclidian
- total
- order relation
- . . . .

**Definition 2.5.3.** A binary relation  $R \subseteq W \times W$  is

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

$$\forall a. aRa$$

- *transitive* if

$$\forall a b c. aRb \wedge bRc \rightarrow aRc$$

- (right) *Euclidean* if

$$\forall a b c. aRb \wedge aRc \rightarrow bRc$$

---

<sup>3</sup>Characterize as an exercise what *exactly* (not just roughly) the condition the accessibility relation must have to make  $\Box$  and  $\Diamond$  identical.

- *total if*

$$\forall a. \exists b. aRb$$

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.

In that connection a word of caution: first-order logic seems like powerful, the swiss army knife of logics, perfect for formalizing everything if one is patient or obsessive enough. One should be careful, though, FOL has its limitations (and not just because theoremhood is undecidable). In some way, FOL is rather *weak* actually, for instance one cannot even characterize the natural numbers, at least not exactly (one way of getting a feeling for that is: Peano’s axioms, that characterize the natural numbers, are *not* first-order). First-order logic is not strong enough to capture induction, and then one is left with a notation that looks exactly like “natural numbers” but for which one cannot use *induction*. And that is thereby a disappointingly useless form of “natural numbers”. . . In the chapter about the  $\mu$ -calculus, we will touch upon those issues again.

In practice, some people use “applied forms” of first-order logics. For instance, one has a signature that captures the natural numbers, and then one *assumes* that the vocabulary is interpreted by the actual natural numbers as model. The assumption is, as just mentioned, not capturable by first-order logic itself, it’s an external assumption. If one would like to capture that inside a formal logical system (and not just assuming it and explain that by English sentences), one would have to use stronger systems than available in first-order logics. As an example: Hoare-logic was mentioned in the second lecture, which is based traditionally on first-order logic. Those kind of logic is used to talk about programs and those programs contain data stored in variables, like natural numbers and similar things. However, when talking about natural numbers or other data structures in Hoare logic, one is not concerned with “are those really expressible in pure first-order logic”, one is interested in the program verification, so it’s often simply assumed that those are the natural numbers as known from math etc. We may encounter not directly Hoare-logic, but perhaps dynamic logic, which is also a form of (multi)-modal logic. Actually Hoare-logic can be seen as a special case of dynamic logic.

### Valid in frame/for a set of frames

If  $(W, R, V), s \models \varphi$  for all  $s$  and  $V$ , we write

$$(W, R) \models \varphi$$



## Samples

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

## Some exercises

Prove the double implications from the slide before!

**Hints** By “double implications”, the iff’s (if-and-only-if) are meant. In each case there are two directions to show.

- The forward implications are based on the fact that we quantify over *all* valuations and all states. More precisely; assume an arbitrary frame  $(W, R)$  which does *not* have the property (e.g., reflexive). Find a valuation and a state where the axiom does not hold. You have now the contradiction . . .
- For the backward implication take an arbitrary frame  $(W, R)$  which *has* the property (e.g., Euclidian). Take an arbitrary valuation and an arbitrary state on this frame. Show that the axiom holds in this state under this valuation. Sometimes one may need to use an inductive argument or to work with properties derived from the main property on  $R$  (e.g., if  $R$  is euclidian then  $w_1 R w_2$  implies  $w_2 R w_2$ ).

### 2.5.3 Proof theory and axiomatic systems

We only sketch some proof theory of modal logic, basically because 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 in the introductory part, 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. Actually, Hilbert-style may just be around for reasons of tradition, insofar it was historically the first or among the first. Other forms like natural deduction or sequent presentation came later. Actually proposed with the intention to improve on the presentation of the proof system, for instance allowing more “natural” proofs, i.e., formal proofs that resemble more closely to the way, informal proofs are carried out or structured. 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 may in itself not capture the core of the differences, but it’s an important aspect. As discussed, different classes of

frames (transitive, reflexive ...) correspond to axioms or selection of axioms, and we have seen some.

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  $\Box p \rightarrow \Box p$  “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  $\rightarrow$ . The axiom is known as distribution axiom or transitionally also as axiom K. In a way, the given rules are somewhat the *base line* for all classical modal logics. Modal logics with propositional part covered plus necessitation and axiom K are also called *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.

## Base line axiomatic system (“K”)

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

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

$$\frac{\varphi \rightarrow \psi \quad \varphi}{\psi} \text{ MP}$$

$$\frac{\varphi}{\Box\varphi} \text{ G}$$

The distribution axiom K is written as “rule” without premises. The system also focuses on the “new” part, 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).

The sketched logic is also known under the name K itself, so K is not just the name of the axiom. The presentation here is Hilber-style, but there are different ways to make a derivation system for the logic K. On the next slide, there are a few more axioms (with their traditional names, some of which are just numbers, like “axiom 4” or “axiom 5”), and 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 one of the next slides.

## Sample axioms for different accessibility relations

$$\Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi) \quad (\text{K})$$

$$\Box\varphi \rightarrow \Diamond\varphi \quad (\text{D})$$

$$\Box\varphi \rightarrow \varphi \quad (\text{T})$$

$$\Box\varphi \rightarrow \Box\Box\varphi \quad (4)$$

$$\neg\Box\varphi \rightarrow \Box\neg\Box\varphi \quad (5)$$

$$\Box(\Box\varphi \rightarrow \psi) \rightarrow \Box(\Box\psi \rightarrow \varphi) \quad (3)$$

$$\Box(\Box(\varphi \rightarrow \Box\varphi) \rightarrow \varphi) \rightarrow (\Diamond\Box\varphi \rightarrow \varphi) \quad (\text{Dum})$$

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.

There are many more different axioms studied in the literature, how they related and what not. The axiom called DUM is more esoteric (“[among the] most bizarre formulae that occur in the literature” [17]) and actually, there are even different versions of that (DUM<sub>1</sub>, DUM<sub>2</sub> ...).

### Different “flavors” of modal logic

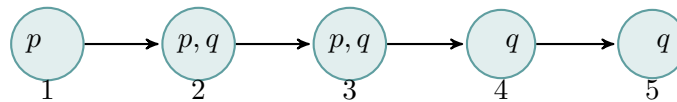
Logic	Axioms	Interpretation	Properties of $R$	
D	K D	deontic	total	
T	K T		reflexive	
K45	K 4 5	doxastic	transitive/euclidean	Concerning the terminology
S4	K T 4		reflexive/transitive	
S5	K T 5	epistemic	reflexive/euclidean	
			reflexive/symmetric/transitive	
			equivalence relation	

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

### 2.5.4 Exercises

#### Some exercises

Consider the frame  $(W, R)$  with  $W = \{1, 2, 3, 4, 5\}$  and  $(i, i + 1) \in R$



Let the “valuation”  $\tilde{V}(p) = \{2, 3\}$  and  $\tilde{V}(q) = \{1, 2, 3, 4, 5\}$  and let the model  $M$  be  $M = (W, R, V)$ . Which of the following statements are correct in  $M$  and why?

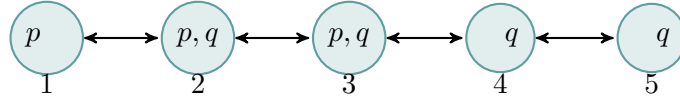
- $M, 1 \models \Diamond \Box p$
- $M, 1 \models \Diamond \Box p \rightarrow p$
- $M, 3 \models \Diamond(q \wedge \neg p) \wedge \Box(q \wedge \neg p)$
- $M, 1 \models q \wedge \Diamond(q \wedge \Diamond(q \wedge \Diamond(q \wedge \Diamond q)))$
- $M \models \Box q$

The answers to the above questions are

- yes
- no
- yes
- yes,
- yes. But why?

### Exercises (2): bidirectional frames

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



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

1.  $M, 1 \models \Diamond \Box p$
2.  $M, 1 \models \Diamond \Box p \rightarrow p$
3.  $M, 3 \models \Diamond(q \wedge \neg p) \wedge \Box(q \wedge \neg p)$
4.  $M, 1 \models q \wedge \Diamond(q \wedge \Diamond(q \wedge \Diamond(q \wedge \Diamond q)))$
5.  $M \models \Box q$
6.  $M \models \Box q \rightarrow \Diamond \Diamond p$

The notion of bidirectional The  $R$  can be separated into two disjoint relations  $R_F$  and  $R_P$ , which one is the inverse of the other.

1.  $M, 1 \models \Diamond \Box p$ : Correct [it was wrongly mentioned as incorrect in an earlier version of the script]
2.  $M, 1 \models \Diamond \Box p \rightarrow p$ : Correct
3.  $M, 3 \models \Diamond(q \wedge \neg p) \wedge \Box(q \wedge \neg p)$ : Incorrect
4.  $M, 1 \models q \wedge \Diamond(q \wedge \Diamond(q \wedge \Diamond(q \wedge \Diamond q)))$ : Correct
5.  $M \models \Box q$ : Correct ... but is it the same explanation as before?
6.  $M \models \Box q \rightarrow \Diamond \Diamond p$

### Exercises (3): validities

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

1.  $\Box \perp$
2.  $\Diamond p \rightarrow \Box p$
3.  $p \rightarrow \Box \Diamond p$
4.  $\Diamond \Box p \rightarrow \Box \Diamond p$

1.  $\Box \perp$ : 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.

As for further reading, [19] and [4] may be good reads.

## 2.6 Dynamic logics

### Introduction

#### Problem

- FOL: “very” expressive but *undecidable*. Perhaps good for mathematics but not ideal for computers.
- !! FOL can talk about the state of the system. But how to talk about *change of state* in a *natural* way?
- modal logic: gives us the power to talk about *changing of state*. Modal logics is natural when one is interested in systems that are essentially modeled as states and transitions between states.

FOL: At least relatively. There are much more expressive logics again. FOL has also some serious restrictions.} By *far* not the most expressive logic there is. We want to talk about programs, states of programs, and change of the state of the computer via executing programming instructions, like assignments. Modal logic can be seen as FOL with one free variable, but we loose the “beauty” of modal logics.

### 2.6.1 Multi-modal logic

#### Multi-modal logic

“Kripke frame”  $(W, R_a, R_b)$ , where  $R_a$  and  $R_b$  are two relations over  $W$ .

**Syntax (2 relations)** *Multi-modal logic* has one modality for each relation:

$$\varphi ::= p \mid \perp \mid \varphi \rightarrow \varphi \mid \Diamond_a \varphi \mid \Diamond_b \varphi \quad (2.7)$$

where  $p$  is from a set of propositional constants (i.e., functional symbols of arity 0) and the other operators are *derived* as usual:

$$\varphi ::= \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg \varphi \mid \Box_a \varphi \mid \Box_b \varphi \quad (2.8)$$

**Rest Semantics:** “natural” generalization of the “mono”-case

$$M, w \models \Diamond_a \varphi \text{ iff } \exists w' : w R_a w' \text{ and } M, w' \models \varphi \quad (2.9)$$

- analogously for modality  $\Diamond_b$  and relation  $R_b$

As *multi-modal* logic: *obvious generalization* of modal logic from before

1. The relations can overlap; i.e., their intersection need not be empty
2. of course: more than 2 relations possible, for each relation one modality.
3. There may be *infinitely* many relations and infinitely many modalities.

Infinitely many modalities are possible. One has to be careful then, though. Infinitely many modalities may pose theoretical challenges (not just for the question how to deal with them computationally). We ignore issues concerning that in this lecture. As a further remark: later there will be PDL and maybe TLA (temporal logic of actions). There are many *actions* involved, which lead to many “modalities”.

## 2.6.2 Dynamic logics

### Dynamic logics

- different variants
- can be seen as special case of multi-modal logics
- variant of Hoare-logics
- here: PDL on **regular** programs
- “P” stands for “propositional”

### Regular programs

**DL** Dynamic logic is a multi-modal logic to talk about programs.

here: dynamic logic talks about **regular programs**

Regular programs are formed syntactically from:

- *atomic* programs  $\Pi_0 = \{a, b, \dots\}$ , which are indivisible, single-step, basic programming constructs
- *sequential* composition  $\alpha \cdot \beta$ , which means that program  $\alpha$  is executed/done first and then  $\beta$ .
- *nondeterministic choice*  $\alpha + \beta$ , which nondeterministically chooses one of  $\alpha$  and  $\beta$  and executes it.
- *iteration*  $\alpha^*$ , which executes  $\alpha$  some nondeterministically chosen finite number of times.
- the special **skip** and **fail** programs (denoted **1** resp. **0**)

### Regular programs and tests

**Definition 2.6.1** (Regular programs). The syntax of *regular programs*  $\alpha, \beta \in \Pi$  is given according to the grammar:

$$\alpha ::= a \in \Pi_0 \mid \mathbf{1} \mid \mathbf{0} \mid \alpha \cdot \alpha \mid \alpha + \alpha \mid \alpha^* \mid \varphi? . \quad (2.10)$$

The clause  $\varphi?$  is called *test*.

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

## Tests

- *simple* Boolean tests:  $\varphi ::= \top \mid \perp \mid \varphi \rightarrow \varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$
- *complex* tests:  $\varphi?$  where  $\varphi$  is a logical formula in *dynamic logic*

## Propositional Dynamic Logic: Syntax

**Definition 2.6.2** (DPL syntax). The formulas  $\varphi$  of *propositional dynamic logic* (PDL) over regular programs  $\alpha$  are given as follows.

$$\begin{aligned} \alpha &::= a \in \Pi_0 \mid \mathbf{1} \mid \mathbf{0} \mid \alpha \cdot \alpha \mid \alpha + \alpha \mid \alpha^* \mid \varphi? \\ \varphi &::= p, q \in \Phi_0 \mid \top \mid \perp \mid \varphi \rightarrow \varphi \mid [\alpha]\varphi \end{aligned} \quad (2.11)$$

where  $\Phi_0$  is a set of atomic propositions.

1. **programs**, which we denote  $\alpha \dots \in \Pi$
2. **formulas**, which we denote  $\varphi \dots \in \Phi$

**Propositional** Dynamic Logic (PDL): because based on propositional logic, only

## PDL: remarks

- Programs  $\alpha$  interpreted as a **relation**  $R_\alpha$   
 $\Rightarrow$  multi-modal logic.
- $[\alpha]\varphi$  defines many modalities, one modality for each program, each interpreted over the relation defined by the program  $\alpha$ .
- The relations of the basic programs are just **given**.
- Operations on/composition of programs are interpreted as operations on relations.
- $\infty$  many complex programs  $\Rightarrow \infty$  many relations/modalities
- But we think of a single modality  $[\dots]\varphi$  with programs inside.
- $[\dots]\varphi$  is the universal one, with  $\langle \dots \rangle \varphi$  defined as usual.

“If program  $\alpha$  is started in the current state, then, *if* it terminates, then in its final state,  $\varphi$  holds.”

## Exercises: “programs”

Define the following programming constructs in PDL:



<b>skip</b>	$\triangleq$	$\top?$
<b>fail</b>	$\triangleq$	$\perp?$
<b>if</b> $\varphi$ <b>then</b> $\alpha$ <b>else</b> $\beta$	$\triangleq$	$(\varphi? \cdot \alpha) + (\neg\varphi? \cdot \beta)$
<b>if</b> $\varphi$ <b>then</b> $\alpha$	$\triangleq$	$(\varphi? \cdot \alpha) + (\neg\varphi? \cdot \mathbf{skip})$
<b>case</b> $\varphi_1$ <b>then</b> $\alpha_1$ ; ...	$\triangleq$	$(\varphi_1? \cdot \alpha_1) + \dots + (\varphi_n? \cdot \alpha_n)$
<b>case</b> $\varphi_n$ <b>then</b> $\alpha_n$		
<b>while</b> $\varphi$ <b>do</b> $\alpha$	$\triangleq$	$(\varphi? \cdot \alpha)^* \cdot \neg\varphi?$
<b>repeat</b> $\alpha$ <b>until</b> $\varphi$	$\triangleq$	$\alpha \cdot (\neg\varphi? \cdot \alpha)^* \cdot \varphi?$
<i>(General while loop)</i>		
<b>while</b> $\varphi_1$ <b>then</b> $\alpha_1$   ...   $\varphi_n$ <b>then</b> $\alpha_n$ <b>od</b>	$\triangleq$	$(\varphi_1? \cdot \alpha_1 + \dots + \varphi_n? \cdot \alpha_n)^* \cdot (\neg\varphi_1 \wedge \dots \neg \wedge \varphi_n)?$

### 2.6.3 Semantics of PDL

#### Making Kripke structures “multi-modal-prepared”

**Definition 2.6.3** (Labeled Kripke structures). Assume a set of labels  $\Sigma$ . A *labeled Kripke structure* is a tuple  $(W, R, \Sigma)$  where

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

is the disjoint union of the relations indexed by the labels of  $\Sigma$ .

for us (at least now): The labels of  $\Sigma$  can be thought as programs

- $\Sigma$ : aka alphabet,
- alternative:  $R \subseteq W \times \Sigma \times W$
- labels  $l, l_1 \dots$  but also  $a, b, \dots$  or others
- often:  $\xrightarrow{a}$ , like  $w_1 \xrightarrow{a} w_2$  or  $s_1 \xrightarrow{a} s_2$

#### Regular Kripke structures

- “labels” now have “structure”
- remember regular program syntax
- interpretation of certain programs/labels fixed,
  - $\mathbf{0}$ : failing program
  - $\alpha_1 \cdot \alpha_2$ : sequential composition
  - ...
- thus, relations like  $\mathbf{0}$ ,  $R_{\alpha_1 \cdot \alpha_2}$ , ... must obey side-conditions

leaving open the interpretation of the “atoms”  $a$ , we fix the interpretation/semantics of the constructs of regular programs

### Regular Kripke structures

**Definition 2.6.4** (Regular Kripke structures). A *regular Kripke structure* is a Kripke structure labeled as follows. For all basic 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.

$$\begin{aligned} R_1 &= Id \\ R_0 &= \emptyset \\ R_{\alpha_1 \cdot \alpha_2} &= R_{\alpha_1} \circ R_{\alpha_2} \\ R_{\alpha_1 + \alpha_2} &= R_{\alpha_1} \cup R_{\alpha_2} \\ R_{\alpha^*} &= \bigcup_{n \geq 0} R_{\alpha}^n \end{aligned}$$

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

### Kripke models and interpreting PDL formulas

Now: add *valuations*  $\Rightarrow$  Kripke model

**Definition 2.6.5** (Semantics). A PDL formula  $\varphi$  is *true* in the world  $w$  of a regular Kripke model  $M$ , i.e., we have attached a valuation  $V$  also, written  $M, w \models \varphi$ , if:

$$\begin{array}{ll} M, w \models p_i & \text{iff } p_i \in V(w) \text{ for all propositional constants} \\ M, w \not\models \perp & \text{and } M, w \models \top \\ M, w \models \varphi_1 \rightarrow \varphi_2 & \text{iff whenever } M, w \models \varphi_1 \text{ then also } M, w \models \varphi_2 \\ M, w \models [\alpha]\varphi & \text{iff } M, w' \models \varphi \text{ for all } w' \text{ such that } wR_{\alpha}w' \\ M, w \models \langle \alpha \rangle \varphi & \text{iff } M, w' \models \varphi \text{ for some } w' \text{ such that } wR_{\alpha}w' \end{array}$$

### Semantics (cont'd)

- programs and formulas: mutually dependent
- *omitted* so far: what relationship corresponds to

$\varphi?$

- remember the intuitive meaning (semantics) of tests

### Test programs

Intuition: tests interpreted as subsets of the identity relation.

$$R_{\varphi?} = \{(w, w) \mid w \models \varphi\} \subseteq I \tag{2.12}$$

More precisely:

- for  $\top?$  the relation becomes  $R_{\top?} = Id$   
(testing  $\top$  succeeds everywhere and is as the **skip** program)
- for  $\perp?$  the relation becomes  $R_{\perp?} = \emptyset$   
( $\perp$  is nowhere true and is as the **fail** program)
- $R_{(\varphi_1 \wedge \varphi_2)?} = \{(w, w) \mid w \models \varphi_1 \text{ and } w \models \varphi_2\}$
- Testing a complex formula involving  $[\alpha]\varphi$  is like looking into the **future** of the program and then deciding on the action to take...

### Axiomatic System of PDL

Take all tautologies of propositional logic (i.e., the axiom system of PL from Lecture 2) and add

Axioms:

$$[\alpha](\phi_1 \rightarrow \phi_2) \rightarrow ([\alpha]\phi_1 \rightarrow [\alpha]\phi_2) \quad (1)$$

$$[\alpha](\phi_1 \wedge \phi_2) \leftrightarrow [\alpha]\phi_1 \wedge [\alpha]\phi_2 \quad (2)$$

$$[\alpha + \beta]\phi \leftrightarrow [\alpha]\phi \wedge [\beta]\phi \quad (3)$$

$$[\alpha \cdot \beta]\phi \leftrightarrow [\alpha][\beta]\phi \quad (4)$$

$$[\phi?]\psi \leftrightarrow \phi \rightarrow \psi \quad (5)$$

$$\phi \wedge [\alpha][\alpha^*]\phi \leftrightarrow [\alpha^*]\phi \quad (6)$$

$$\phi \wedge [\alpha^*](\phi \rightarrow [\alpha]\phi) \rightarrow [\alpha^*]\phi \quad (\text{IND})$$

Rules: take the (MP) modus ponens and (G) generalization of Modal Logic.

### 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)

### 2.6.4 Exercises

The exercises have been placed on a separate sheet.

#### Exercises: Play with binary relations

- **Composition** of relations **distributes** over **union** of relations.  

$$R \circ (\bigcup_i Q_i) = \bigcup_i (R \circ Q_i) \quad (\bigcup_i Q_i) \circ R = \bigcup_i (Q_i \circ R)$$
- $R^* \triangleq I \cup R \cup R \circ R \cup \dots \cup R^n \cup \dots \triangleq \bigcup_{n \geq 0} R^n$

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  $\alpha$  and  $\beta$  are **equivalent** iff they represent the same binary relation  $R_\alpha = R = R_\beta$ .

Show:

1. Two programs  $\alpha$  and  $\beta$  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:

<pre> <b>while</b> <math>\phi_1</math> <b>do</b>   <math>\alpha</math>;   <b>while</b> <math>\phi_2</math> <b>do</b> <math>\beta</math> </pre>	<pre> <b>if</b> <math>\phi_1</math> <b>then</b>   <math>\alpha</math>;   <b>while</b> <math>\phi_1 \vee \phi_2</math> <b>do</b>     <b>if</b> <math>\phi_2</math> <b>then</b> <math>\beta</math> <b>else</b> <math>\alpha</math> </pre>
--	---

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:

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

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

# Chapter 3

## LTL model checking

### 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 . . . . .	57
3.2	LTL . . . . .	58
3.3	Logic model checking: What is it about? . . . . .	76
3.4	Automata and logic . . . . .	80
3.5	Model checking algorithm . . . . .	109
3.6	Final Remarks . . . . .	115

What is it about?

## 3.1 Introduction

In this chapter, we leave behind a bit the “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  $M \models \varphi$ . We do that for a specific modal logic, more precisely, as 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 mathematics or philosophy). We will also cover *one* central way of doing model checking of temporal logics, namely *automata-based* model checking.

### Temporal logic?

- **Temporal logic:** is the/a logic of “time”
- *modal* logic.
- different ways of modeling time.
  - linear vs. branching time
  - time instances vs. time intervals
  - discrete time vs. continuous time
  - past and future vs. future only
  - ...

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 we have introduced modal logics, 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 are variants of temporal logics which handle real-time, including versions of real-time LTL, but they won't (probably) occur in this lecture.

## LTL

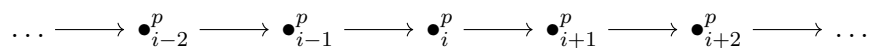
- **linear** time temporal logic
- one central temporal logic in CS
- supported by Spin and other model checkers
- many variations
- We have used FOL to express properties of states.
  - $\langle x : 21, y : 49 \rangle \models x < y$
  - $\langle x : 21, y : 7 \rangle \not\models x < y$
- A computation is a sequence of states.
- To express properties of computations, we need to extend FOL.
- This we can do using temporal logic.

## 3.2 LTL

### LTL: speaking about “time”

In **linear temporal logic (LTL)**, also called *linear-time temporal logic*, we can describe properties as, 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)



Time here is *linear* and *discrete*. One consequently just uses ordinary natural numbers (or integers) to index the points in time. We will mostly only be concerned with 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. Branching time is an alternative to the linear modelling of time, and instead of having discrete point in times, one could have dense time and/or deal with intervals.

### 3.2.1 Syntax

#### 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, like *propositional* logic or *first-order* 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*. The first thing to extend is the syntax: we have formulas  $\psi$  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 following grammar.

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

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.

### 3.2.2 Semantics

Fixing the meaning of LTL formulas means, 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 should be also of linear structure. 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  $AP \rightarrow \mathbb{B}$  here not a valuation, 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  $\mathbb{N} \rightarrow (AP \rightarrow \mathbb{B})$ . This is equivalently represented as an infinite sequence of the form

$$s_0 s_1 s_2 \dots$$

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  $\pi, \pi'$  etc. to refer to them. It's important to remember that paths are *infinite*. As discussed in the lecture: if we allowed *finite* paths, we would lose the kind of nice duality between the  $\diamond$  and  $\square$  operator (that refers to the fact that that  $\neg\square\neg$  is the same as  $\diamond$ , and the other way around).

In that connection: what's  $\neg \bigcirc \neg$ ?

### Some remarks on terminology: paths, states, and valuations

The notions of states and paths ... are slightly differing in the 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 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 there (and some people surely do), it's just that the standard case there is finite sequence, and therefore the word *path* is reserved for those. LTL on the other hand deals with infinite sequences, and consequently uses the word paths for that.

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  $i + 1$ ". 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 transitions or edges in the underlying transition system or graph.

Finally, of course, the conventional notion of path in a graph does not speak 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  $\mathbb{N}_n \rightarrow N$  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  $\pi$  is of type  $\mathbb{N} \rightarrow (AP \rightarrow \mathbb{B})$ , i.e., an infinite sequence of states (or valuations).

## Paths and computations

### Definition 3.2.1 (Path).

- A *path* is an infinite sequence

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

of states.

- $\pi^k$  denotes the *path*  $s_k, s_{k+1}, s_{k+2}, \dots$
- $\pi_k$  denotes the *state*  $s_k$ .

It’s intended (later) that paths represent behavior of programs resp. “going” through a Kripke-model or transition system. A transitions system is a graph-like structure (and may contain cycles), and a paths 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’s 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  $\models_{\text{ul}}$  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).** An LTL formula  $\varphi$  is *true* relative to a path  $\pi$ , written  $\pi \models \varphi$ , under the following conditions:

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

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 one can express remaining operators from the syntax by them. Those other operators are  $\square$ ,  $\diamond$ ,  $R$ , 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-expanded 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  $\square$  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  $\varphi_1$  “releases”  $\varphi_2$ ’s need to hold, i.e.,  $\varphi_2$  has to hold up until and *including* the point where  $\varphi_1$  first holds and if  $\varphi_1$  never holds (i.e., never “releases  $\varphi_2$ ”), then  $\varphi_2$  has to hold forever. If there a point where  $\varphi_1$  is first true and thus releases  $\varphi_2$ , then at that “release point” both  $\varphi_1$  and  $\varphi_2$  have to hold. Furthermore, it’s a “weak” form of a “reverse until” insofar that it’s not required that  $\varphi_1$  ever releases  $\varphi_2$ .

$$\begin{array}{ll}
\pi \models \square\varphi & \text{iff } \pi^k \models \varphi \text{ for all } k \geq 0 \\
\pi \models \diamond\varphi & \text{iff } \pi^k \models \varphi \text{ for some } k \geq 0 \\
\pi \models \varphi_1 R \varphi_2 & \text{iff for every } j \geq 0, \\
& \text{if } \pi^i \not\models \varphi_1 \text{ for every } i < j \text{ then } \pi^j \models \varphi_2 \\
\pi \models \varphi_1 W \varphi_2 & \text{iff } \pi \models \varphi_1 U \varphi_2 \text{ or } \pi \models \square\varphi_1
\end{array}$$

### Validity and semantic equivalence

**Definition 3.2.3** (Validity and equivalence).

- $\varphi$  is (*temporally*) *valid*, written  $\models \varphi$ , if  

$$\pi \models \varphi \text{ for all paths } \pi.$$
- $\varphi_1$  and  $\varphi_2$  are *equivalent*, written  $\varphi_1 \sim \varphi_2$ , if  

$$\models \varphi_1 \leftrightarrow \varphi_2 \text{ (i.e. } \pi \models \varphi_1 \text{ iff } \pi \models \varphi_2, \text{ for all } \pi).$$

*Example 3.2.4.*  $\Box$  distributes over  $\wedge$ , while  $\Diamond$  distributes over  $\vee$ .

$$\begin{aligned}\Box(\varphi \wedge \psi) &\sim (\Box\varphi \wedge \Box\psi) \\ \Diamond(\varphi \vee \psi) &\sim (\Diamond\varphi \vee \Diamond\psi)\end{aligned}$$

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

In a 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  $\perp$ ), 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  $\varphi_1 \rightarrow \varphi_2$  is valid), one could model-check using formula  $\varphi_1$  (which might be easier), and use that to establish that also  $\varphi_2$  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.

However, the two problems —  $M \models \varphi$  vs.  $\models \varphi_1 \rightarrow \varphi_2$  — are not [...]

The next 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 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.

### 3.2.3 Illustrations

$\pi \models \Box p$



$\pi \models \Diamond p$



$$\pi \models \bigcirc p$$

$$\bullet_0 \longrightarrow \bullet_1^p \longrightarrow \bullet_2 \longrightarrow \bullet_3 \longrightarrow \bullet_4 \longrightarrow \dots$$

### 3.2.4 Some more illustrations

$$\pi \models p \, U \, q \text{ (sequence of } p\text{'s is } \textit{finite})}$$

$$\bullet_0^p \longrightarrow \bullet_1^p \longrightarrow \bullet_2^p \longrightarrow \bullet_3^q \longrightarrow \bullet_4 \longrightarrow \dots$$

$$\pi \models p \, R \, q \text{ (sequence of } q\text{s may be infinite)}$$

$$\bullet_0^q \longrightarrow \bullet_1^q \longrightarrow \bullet_2^q \longrightarrow \bullet_3^{p,q} \longrightarrow \bullet_4 \longrightarrow \dots$$

$$\pi \models p \, W \, q$$

The sequence of  $ps$  may be infinite.  $(p \, W \, q \sim p \, U \, q \vee \Box p)$ .

$$\bullet_0^p \longrightarrow \bullet_1^p \longrightarrow \bullet_2^p \longrightarrow \bullet_3^p \longrightarrow \bullet_4^p \longrightarrow \dots$$

### 3.2.5 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  $\mu$ -calculus). 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 to 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 in a way 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.

We focus on the past version of LTL. Doing so, it’s 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  $\mathbb{N} \rightarrow (AP \rightarrow 2)$ ), 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.

It can be noted: one of the student representations this year (the one about run-time verification), will make use of a past-time LTL,

## The past

### Observation

- Manna and Pnueli [24] uses pairs  $(\pi, j)$  of paths and positions instead of just the path  $\pi$  because they have *past-formulas*: formulas without future operators (the ones we use) but possibly with *past operators*, like  $\Box^{-1}$  and  $\Diamond^{-1}$ .

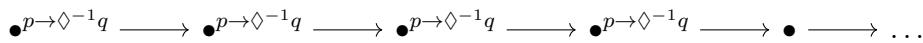
$$\begin{aligned} (\pi, j) \models \Box^{-1}\varphi & \text{ iff } (\pi, k) \models \varphi \text{ for all } k, 0 \leq k \leq j \\ (\pi, j) \models \Diamond^{-1}\varphi & \text{ iff } (\pi, k) \models \varphi \text{ for some } k, 0 \leq k \leq j \end{aligned}$$

- However, it can be shown that for any formula  $\varphi$ , there is a *future-formula* (formulae without past operators)  $\psi$  such that

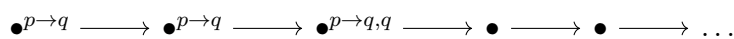
$$(\pi, 0) \models \varphi \text{ iff } (\pi, 0) \models \psi$$

### The past: example

$$\Box(p \rightarrow \Diamond^{-1}q)? \quad (\pi, 0) \models \Box(p \rightarrow \Diamond^{-1}q)$$



$$(\pi, 0) \models q \text{ R } (p \rightarrow q)$$



## 3.2.6 Examples

### Some examples

#### Temporal properties

1. If  $\varphi$  holds initially, then  $\psi$  holds eventually.
2. Every  $\varphi$ -position is responded by a later  $\psi$ -position (**response**)
3. There are **infinitely** many  $\psi$ -positions.
4. Sooner or later,  $\varphi$  will hold *permanently* (**permanence, stabilization**).
5. The first  $\varphi$ -position must coincide or be preceded by a  $\psi$ -position.
6. Every  $\varphi$ -position initiates a sequence of  $\psi$ -positions, and if terminated, by a  $\chi$ -position.

### Formalization of “informal” properties

It can be difficult to correctly formalize informally stated requirements in temporal logic.

#### Informal statement: “ $\varphi$ implies $\psi$ ”

- $\varphi \rightarrow \psi$ ?  $\varphi \rightarrow \psi$  holds in the initial state.
- $\Box(\varphi \rightarrow \psi)$ ?  $\varphi \rightarrow \psi$  holds in every state.
- $\varphi \rightarrow \Diamond\psi$ ?  $\varphi$  holds in the initial state,  $\psi$  will hold in some state.
- $\Box(\varphi \rightarrow \Diamond\psi)$ ? “response”

It is not obvious, which one of them (if any) is necessarily what is intended.

*Example 3.2.5.*  $\varphi \rightarrow \Diamond\psi$ : If  $\varphi$  holds initially, then  $\psi$  holds eventually.

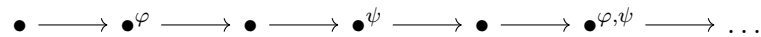


This formula will also hold in every path where  $\varphi$  does not hold initially.



*Example 3.2.6 (Response).*  $\Box(\varphi \rightarrow \Diamond\psi)$

Every  $\varphi$ -position coincides with or is followed by a  $\psi$ -position.



This formula will also hold in every path where  $\varphi$  never holds.



*Example 3.2.7 ( $\infty$ ).*  $\Box\Diamond\psi$

There are infinitely many  $\psi$ -positions.

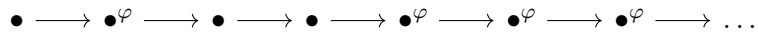


- model-checking?
- run-time verification?

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

**Permanence:**  $\diamond \Box \varphi$

Eventually  $\varphi$  will hold *permanently*.

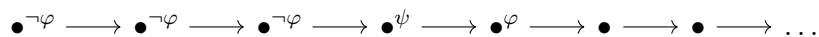


Equivalently: there are *finitely* many  $\neg\varphi$ -positions.

**And another one**

*Example 3.2.8.*  $(\neg\varphi) W \psi$

The first  $\varphi$ -position must coincide or be preceded by a  $\psi$ -position.



$\varphi$  may never hold



**LTL example**

*Example 3.2.9.*  $\Box(\varphi \rightarrow \psi W \chi)$

Every  $\varphi$ -position initiates a sequence of  $\psi$ -positions, and if terminated, by a  $\chi$ -position.



The sequence of  $\psi$ -positions need not terminate.



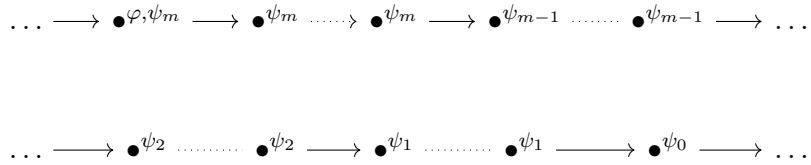
### Nested waiting-for

A **nested waiting-for formula** is of the form

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

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

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



**Explanation** Every  $\varphi$ -position initiates a succession of intervals, beginning with a  $\psi_m$ -interval, ending with a  $\psi_1$ -interval and possibly terminated by a  $\psi_0$ -position. Each interval may be empty or extend to infinity.

### Duality

**Definition 3.2.10** (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*:

- If  $\bullet$  is the dual of  $\circ$  then
- $\circ$  is the dual of  $\bullet$ , thus
- we may refer to two connectives as dual (of each other).

The  $\circ$  and  $\bullet$  operators are not concrete connectives or operators, they are meant as “placeholders”. One can have a corresponding notion of duality for the unary operators  $\diamond$  and  $\square$ , and even for null-ary “operators”.



## Dual connectives

- $\wedge$  and  $\vee$  are duals:

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

- $\neg$  is its own dual:

$$\neg\neg\varphi \sim \varphi.$$

- What is the dual of  $\rightarrow$ ? It's  $\leftarrow$ :

$$\begin{aligned} \neg(\varphi \rightarrow \psi) &\sim \varphi \leftarrow \psi \\ &\sim \psi \rightarrow \varphi \\ &\sim \neg\varphi \rightarrow \neg\psi \end{aligned}$$

## Complete sets of connectives

- 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.,  $\leftarrow$  can be defined), but also subsets of it, so we don't actually need all the connectives.

*Example 3.2.11.*  $\{\vee, \neg\}$  is complete.

- $\wedge$  is the dual of  $\vee$ .
- $\varphi \rightarrow \psi$  is equivalent to  $\neg\varphi \vee \psi$ .
- $\varphi \leftrightarrow \psi$  is equivalent to  $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$ .
- $\top$  is equivalent to  $p \vee \neg p$
- $\perp$  is equivalent to  $p \wedge \neg p$

## Duals in LTL

- What is the dual of  $\Box$ ? And of  $\Diamond$ ?
- $\Box$  and  $\Diamond$  are duals.

$$\begin{aligned} \neg\Box\varphi &\sim \Diamond\neg\varphi \\ \neg\Diamond\varphi &\sim \Box\neg\varphi \end{aligned}$$

- Any other?
- $U$  and  $R$  are duals.

$$\begin{aligned} \neg(\varphi U \psi) &\sim (\neg\varphi) R (\neg\psi) \\ \neg(\varphi R \psi) &\sim (\neg\varphi) U (\neg\psi) \end{aligned}$$

### Complete set of LTL operators

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

We don't need all our temporal operators either.

*Proof.* •  $\diamond\varphi \sim \top U \varphi$

•  $\Box\varphi \sim \perp R \varphi$

•  $\varphi R \psi \sim \neg(\neg\varphi U \neg\psi)$

•  $\varphi W \psi \sim \Box\varphi \vee (\varphi U \psi)$  □

### 3.2.7 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. 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 should 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 role 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 kind of 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 liveness properties.

## Classification of properties

We can classify properties expressible in LTL. *Examples:*

**invariant**  $\Box\varphi$

**“liveness”**  $\Diamond\varphi$

**obligation**  $\Box\varphi \vee \Diamond\psi$

**recurrence**  $\Box\Diamond\varphi$

**persistence**  $\Diamond\Box\varphi$

**reactivity**  $\Box\Diamond\varphi \vee \Diamond\Box\psi$

- $\varphi, \psi$ : non-temporal formulas

The *invariant* is a prominent example of a safety property. Each *invariant* property is also a *safety* property. Some people even use the words synonymously (earlier editions of the lecture), but according to the consensus or majority opinion, one should distinguish the notions. 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.

## Safety (slightly simplified)

- important basic class of properties
- relation to testing and run-time verification
- informally “nothing bad ever happens”

**Definition 3.2.12** (Safety/invariant). • A *invariant* formula is of the form

$$\Box\varphi$$

for some first-order/prop. formula  $\varphi$ .

- A *conditional safety* formula is of the form

$$\varphi \rightarrow \Box\psi$$

for (first-order) formulae  $\varphi$  and  $\psi$ .

Safety formulae express *invariance* of some **state property**  $\varphi$ : that  $\varphi$  holds in every state of the computation.

### Safety property example

**Mutex** *Mutual exclusion* is a safety property. Let  $C_i$  denote that process  $P_i$  is executing in the critical section. Then

$$\Box \neg (C_1 \wedge C_2)$$

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

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

$$\Diamond (C_1 \wedge C_2)$$

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

### Liveness properties (simplified)

**Definition 3.2.13** (Liveness). • A *liveness* formula is of the form

$$\Diamond \varphi$$

for some first-order formula  $\varphi$ .

- A *conditional liveness* formula is of the form

$$\varphi \rightarrow \Diamond \psi$$

for propositional/first-order formulae  $\varphi$  and  $\psi$ .

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

### Connection to Hoare logic

- *Partial correctness* is a safety property. Let  $P$  be a program and  $\psi$  the post condition.

$$\Box (\text{terminated}(P) \rightarrow \psi)$$

- In the case of *full partial correctness*, where there is a precondition  $\varphi$ , we get a *conditional safety* formula,

$$\varphi \rightarrow \Box (\text{terminated}(P) \rightarrow \psi),$$

which we can express as  $\{ \varphi \} P \{ \psi \}$  in Hoare Logic.

### Total correctness and liveness

- *Total correctness* is a liveness property. Let  $P$  be a program and  $\psi$  the post condition.

$$\diamond(\text{terminated}(P) \wedge \psi)$$

- In the case of *full total correctness*, where there is a precondition  $\varphi$ , we get a *conditional liveness* formula,

$$\varphi \rightarrow \diamond(\text{terminated}(P) \wedge \psi).$$

### Duality of partial and total correctness

Partial and total correctness are dual.

Let

$$PC(\psi) \triangleq \Box(\text{terminated} \rightarrow \psi)$$

$$TC(\psi) \triangleq \diamond(\text{terminated} \wedge \psi)$$

Then

$$\neg PC(\psi) \sim PC(\neg\psi)$$

$$\neg TC(\psi) \sim TC(\neg\psi)$$

### Obligation

**Definition 3.2.14** (Obligation). • A *simple obligation* formula is of the form

$$\Box\varphi \vee \diamond\psi$$

for first-order formula  $\varphi$  and  $\psi$ .

- An equivalent form is

$$\diamond\chi \rightarrow \diamond\psi$$

which states that some state satisfies  $\chi$  only if some state satisfies  $\psi$ .

### Obligation (2)

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

*Proof.* This is because of the following equivalences.

$$\Box\varphi \sim \Box\varphi \vee \diamond\perp$$

$$\diamond\varphi \sim \Box\perp \vee \diamond\varphi$$

and the facts that  $\models \neg\Box\perp$  and  $\models \neg\diamond\perp$ . □

## Recurrence and Persistence

### Recurrence

**Definition 3.2.15** (Recurrence). • A *recurrence* formula is of the form

$$\Box\Diamond\varphi$$

for some first-order formula  $\varphi$ .

- It states that infinitely many positions in the computation satisfies  $\varphi$ .

### Observation

A response formula, of the form  $\Box(\varphi \rightarrow \Diamond\psi)$ , is equivalent to a recurrence formula, of the form  $\Box\Diamond\chi$ , if we allow  $\chi$  to be a past-formula.

$$\Box(\varphi \rightarrow \Diamond\psi) \sim \Box\Diamond(\neg\varphi) W^{-1}\psi$$

### Recurrence

**Proposition 3.** *Weak fairness<sup>1</sup> can be specified as the following recurrence formula.*

$$\Box\Diamond(\text{enabled}(\tau) \rightarrow \text{taken}(\tau))$$

### Observation

An equivalent form is

$$\Box(\Box\text{enabled}(\tau) \rightarrow \Diamond\text{taken}(\tau)),$$

which looks more like the first-order formula we saw last time.

### Persistence

**Definition 3.2.16** (Persistence). • A *persistence* formula is of the form

$$\Diamond\Box\varphi$$

for some first-order formula  $\varphi$ .

- It states that all but finitely many positions satisfy  $\varphi$ <sup>2</sup>
- Persistence formulae are used to describe the eventual [stabilization](#) of some state property.

<sup>1</sup>weak and strong fairness will be “recurrent” (sorry for the pun) themes. For instance they will show up again in the TLA presentation.

<sup>2</sup>In other words: only finitely (“but”) many position satisfy  $\neg\varphi$ . So at some point onwards, it’s always  $\varphi$ .

## Recurrence and Persistence

Recurrence and persistence are duals.

$$\neg(\Box\Diamond\varphi) \sim (\Diamond\Box\neg\varphi)$$

$$\neg(\Diamond\Box\varphi) \sim (\Box\Diamond\neg\varphi)$$

## Reactivity

### Reactivity

**Definition 3.2.17** (Reactivity). • A *simple reactivity* formula is of the form

$$\Box\Diamond\varphi \vee \Diamond\Box\psi$$

for first-order formula  $\varphi$  and  $\psi$ .

- A very general class of formulae are conjunctions of reactivity formulae.
- An equivalent form is

$$\Box\Diamond\chi \rightarrow \Box\Diamond\psi,$$

which states that if the computation contains infinitely many  $\chi$ -positions, it must also contain infinitely many  $\psi$ -positions.

## Reactivity

**Proposition 4.** *Strong fairness can be specified as the following reactivity formula.*

$$\Box\Diamond\text{enabled}(\tau) \rightarrow \Box\Diamond\text{taken}(\tau)$$

## GCD Example

### GCD Example

Below is a computation  $\pi$  of our recurring GCD program.

- $a$  and  $b$  are fixed:  $\pi \models \Box(a \doteq 21 \wedge b \doteq 49)$ .
- $at(l)$  denotes the formulae ( $\pi \doteq \{l\}$ ).
- *terminated* denotes the formula  $at(l_8)$ .

States are of the form  $\langle \pi, x, y, g \rangle$ .

$$\begin{aligned} \pi : \quad & \langle l_1, 21, 49, 0 \rangle \rightarrow \langle l_2^b, 21, 49, 0 \rangle \rightarrow \langle l_6, 21, 49, 0 \rangle \rightarrow \\ & \langle l_1, 21, 28, 0 \rangle \rightarrow \langle l_2^b, 21, 28, 0 \rangle \rightarrow \langle l_6, 21, 28, 0 \rangle \rightarrow \\ & \langle l_1, 21, 7, 0 \rangle \rightarrow \langle l_2^a, 21, 7, 0 \rangle \rightarrow \langle l_4, 21, 7, 0 \rangle \rightarrow \\ & \langle l_1, 14, 7, 0 \rangle \rightarrow \langle l_2^a, 14, 7, 0 \rangle \rightarrow \langle l_4, 14, 7, 0 \rangle \rightarrow \\ & \langle l_1, 7, 7, 0 \rangle \rightarrow \langle l_7, 7, 7, 0 \rangle \rightarrow \langle l_8, 7, 7, 7 \rangle \rightarrow \dots \end{aligned}$$

## GCD Example

Does the following properties hold for  $\pi$ ? And why?

- <+(1)->  $\Box terminated$  (safety)
- <+(2)->  $at(l_1) \rightarrow terminated$
- <+(3)->  $at(l_8) \rightarrow terminated$
- <+(4)->  $at(l_7) \rightarrow \Diamond terminated$  (conditional liveness)
- <+(5)->  $\Diamond at(l_7) \rightarrow \Diamond terminated$  (obligation)
- <+(6)->  $\Box(gcd(x, y) \doteq gcd(a, b))$  (safety)
- <+(7)->  $\Diamond terminated$  (liveness)
- <+(8)->  $\Diamond \Box(y \doteq gcd(a, b))$  (persistence)
- <+(9)->  $\Box \Diamond terminated$  (recurrence)

## 3.2.8 Exercises

### Exercises

1. Show that the following formulae 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  $\epsilon$ . Two modalities  $\pi$  and  $\tau$  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 Logic model checking: What is it about?

### 3.3.1 The basic method

#### Logic model checking (1)

- a technique for verifying *finite-state* (concurrent) systems



### Often involves steps as follows

1. Modeling the system
  - It may require the use of *abstraction*
  - Often using some kind of *automaton*
2. Specifying the properties the design must satisfy
  - It is impossible to determine all the properties the systems should satisfy
  - Often using some kind of temporal logic
3. Verifying that the system satisfies its specification
  - In case of a negative result: error trace
  - An error trace may be product of a specification error

This above list gives some ingredients, often used in connection with model checking. It's not to be understood as definition like "that's model checking, end of story."

For instance, there are techniques to model-check infinite systems, i.e., systems with infinite states.

### Logic model checking (2)

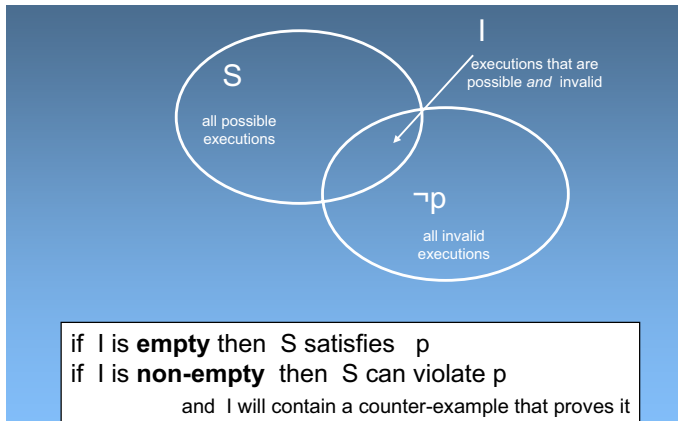
The *application* of model checking at the design stage of a system typically consists of the following **steps**:

1. Choose the properties (correctness requirements) critical to the system you want to build (software, hardware, protocols)
2. Build a model of the system (will use for verification) guided by the above correctness requirements
  - The model should be as small as possible (for efficiency)
  - It should, however, capture everything which is relevant to the properties to be verified
3. Select the appropriate verification method based on the model and the properties (LTL-, CTL\*-based, probabilistic, timed, weighted ...)
4. Refine the verification model and correctness requirements until all correctness concerns are adequately satisfied

### State-space explosion

Main causes of combinatorial complexity in SPIN/Promela (and in other model checkers.)

- The number of and size of buffered channels
- The number of asynchronous processes



### The basic method

- System:  $\mathcal{L}(S)$  (set of possible behaviors/traces/words of  $S$ )
- Property:  $\mathcal{L}(P)$  (the set of valid/desirable behaviors)
- Prove that  $\mathcal{L}(S) \subseteq \mathcal{L}(P)$  (everything possible is valid)
  - Proving language inclusion is complicated
- Method
  - Let  $\overline{\mathcal{L}(P)}$  be the language  $\Sigma^\omega \setminus \mathcal{L}(P)$  of words not accepted by  $P$
  - Prove  $\mathcal{L}(S) \cap \overline{\mathcal{L}(P)} = \emptyset$ 
    - \* there is no accepted word by  $S$  disallowed by  $P$

There are different model checking techniques. We will cover here the **automata-theoretic approach**, which is for instance implemented in the **SPIN** model checker tool.

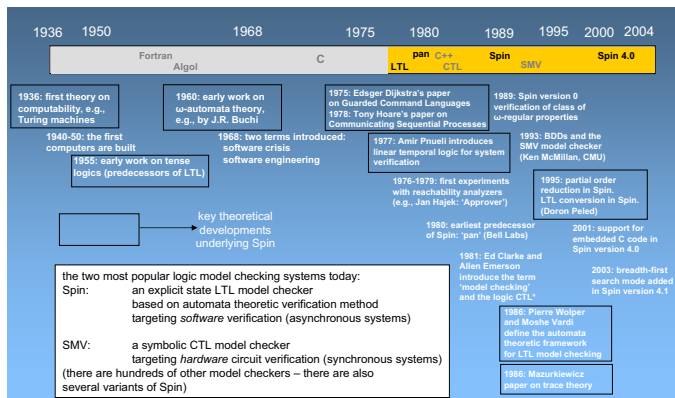
### The basic method

#### Scope of the method

Logic model checkers (LMC) are suitable for *concurrent* and *multi-threading finite-state* systems.

Some of the errors LMC may catch:

- Deadlocks {(two or more competing processes are waiting for the other to finish, and thus neither ever does)}
- Livelocks {(two or more processes continually change their state in response to changes in the other processes)}
- Starvation {(a process is perpetually denied access to necessary resources)}
- Priority and locking problems
- Race conditions {(attempting to perform two or more operations at the same time, which must be done in the proper sequence in order to be done correctly)}
- Resource allocation problems
- Incompleteness of specification
- Dead code {(unreachable code)}
- Violation of certain system bounds
- Logic problems: e.g, temporal relations
- ...



## A bit of history

### 3.3.2 General remarks

#### On correctness (reminder)

- A system is **correct** if it meets its design requirements.
- There is no notion of “absolute” correctness: It is always wrt. a given specification
- Getting the properties (requirements) right is as important as getting the model of the system right

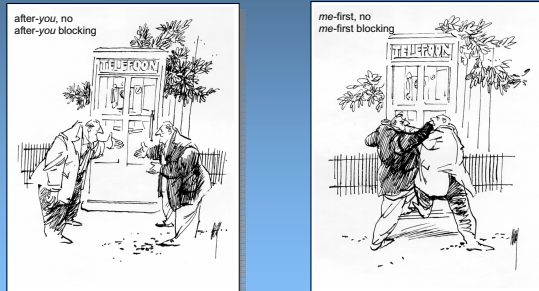
#### Examples of correctness requirements

- A system should not *deadlock*
- No process should *starve* another
- *Fairness* assumptions
  - E.g., an infinite often enabled process should be executed infinitely often
- *Causal relations*
  - E.g., each time a request is send, and acknowledgment must be received (*response* property)

#### On models and abstraction

- The use of *abstraction* is needed for building models (systems may be extremely big)
  - A model is always an abstraction of the reality
- The choice of the model/abstractions depends on the requirements to be checked
- A good model keeps only relevant information
  - A trade-off must be found: too much detail may complicate the model; too much abstraction may oversimplify the reality
- Time and probability are usually abstracted away in LMC

in real-life conflicts ultimately get resolved by *human judgment*.  
computers, though, must be able to resolve it with fixed algorithms



### Building verification models

- Statements about system design and system requirement must be separated
  - One formalism for specifying behavior (**system** design)
  - Another formalism for specifying system **requirements** (correctness properties)
- The two types of statements define a **verification model**
- A model checker can now
  - Check that the behavior specification (the design) is logically consistent with the requirement specification (the desired properties)

### 3.3.3 Motivating examples

#### Distributed algorithms

Two asynchronous processes may easily get blocked when competing for a shared resource

#### A Small multi-threaded program

#### Thread interleaving

#### A simpler example

## 3.4 Automata and logic

### 3.4.1 Finite state automata

#### FSA

**Definition 3.4.1** (Finite-state automaton). A *finite-state automaton* is a tuple  $(Q, q_0, \Sigma, F, \rightarrow)$ , where

```

int x, y, r;
int *p, *q, *z;
int **a;

thread_1(void) /* initialize p, q, and r */
{
    p = &x;
    q = &y;
    z = &x;
}
thread_2(void) /* swap contents of x and y */
{
    r = *p;
    *p = *q;
    *q = r;
}
thread_3(void) /* access z via a and p */
{
    a = &p;
    *a = z;
    **a = 12;
}
    
```

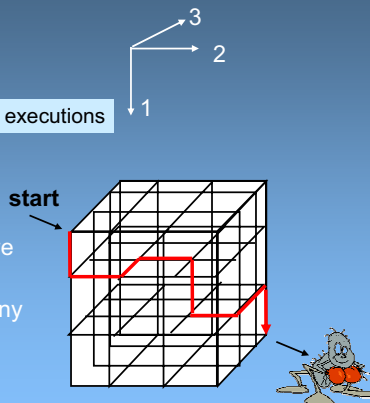
3 asynchronous threads  
accessing shared data  
3 statements each  
how many test runs are needed to  
check that no data corruption can occur?

- the number of possible thread interleavings is...

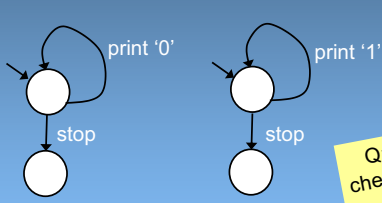
$$\frac{9!}{6! \cdot 3!} \cdot \frac{6!}{3! \cdot 3!} \cdot \frac{3!}{3!} = 1,680 \text{ possible executions}$$

placing 3 sets of 3 tokens in 9 slots

- are all these executions okay?
- can we check them all? should we check them all?
- in classic system testing, how many would normally be checked?



- consider two 2-state automata
  - representing two asynchronous processes
- one can print an arbitrary number of '0' digits, or stop
- the other can print an arbitrary number of '1' digits, or stop



Q: how could a model checker deal with possibly infinite executions?

how many different combined executions are there?  
i.e., how many different binary numbers can be printed?  
how would one test that this system does what we think it does?

- $Q$  is finite set of states
- $q_0 \in Q$  is a distinguished initial state
- the “alphabet”  $\Sigma$  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 called *alphabet* (with a symbol  $\Sigma$ ) 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).

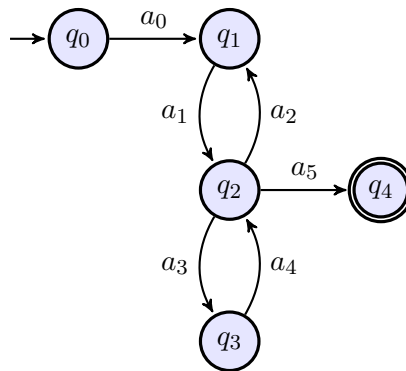
Other names for related concepts is that of a (finite-state) *transition system*. And even Kripke structures or Kripke models can be seen as a variation of the theme, though in a more logical or philosophical context, the *edges* between the workds may not be viewed

as *transitions* or operational steps in a evolving system. In Baier and Katoen [2], they call Kripke structure *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 be 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 is the *acceptance* condition, i.e., the interpretation of the set of accepting states.

In the slides taken from Holzmann. Holzmann uses the following notation:  $\mathcal{A}.S$  denotes the state  $S$  of automaton  $\mathcal{A}$ ,  $\mathcal{A}.T$  denotes the transition relation  $T$  of  $\mathcal{A}$ , and so on. . . . If understood from the context, we will avoid the use of  $\mathcal{A}.$

### Example FSA



The automaton is given by the 6-letter alphabet (or label set)  $\Sigma = \{a_0, a_1, \dots, a_5\}$ , by the 5 states  $q_0, q_1, \dots, q_4$ , 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

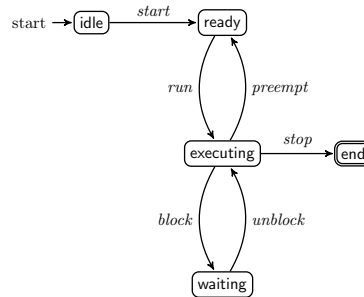
$$\rightarrow = \{(q_0, a_0, q_1), \dots, (q_2, a_5, q_4)\} \subseteq Q \times \Sigma \times Q,$$

but it does not make it more “formal” nor does it add clarity.

### Example: An interpretation

The above automaton may be interpreted as a *process scheduler*:

If course, it’s still the “same” automaton. Using different identifiers for the states —  $q_0, q_1, \dots$  vs. *idle, ready, \dots* — and analogously for the edge labels does not make it into a different automaton. It’s the structure and what it does that matters, not the names chosen to identify elements of the structure. Both automata are *isomorphic* which means “essentially identical”.



### Determinism vs. non-determinism

**Definition 3.4.2** (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  $\Sigma$ .

As it came up in a discussion during the lecture: 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, which 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 automata 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  $A \leftrightarrow B$  can be viewed as total function  $A \rightarrow B_{\perp}$ , where  $B_{\perp}$  represents the set  $B$  extended by an extra error element  $\perp$ .

The automaton corresponding to the process scheduler is *deterministic*.



## Runs

**Definition 3.4.3** (Run). A *run* of a finite state automaton  $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$  is a (possibly infinite) sequence

$$\sigma = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \dots$$

- $q \xrightarrow{a} q'$  is meant as  $(q, a, q') \in \rightarrow$
- each run corresponds to a **state sequence** (a word) over  $Q$  and a **word** over  $\Sigma$

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 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 given contains “full information” insofar that it is a sequence of transitions. It corresponds to the choice of words in Holzmann [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.

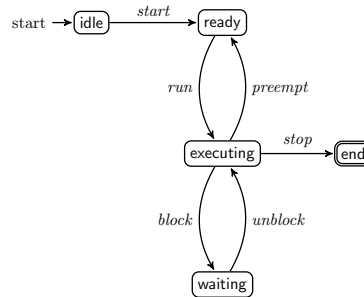
For me, the definition of run as given here is a more “plausible” interpretation of the word. A run or execution (for me) should fix all details that allows to reconstruct or replay what concretely happened. Considering state sequences as run would leave out which *labels* are responsible for that sequence. 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. 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, mainly they 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 “truly concurrent”). Also in connection with weak memory models, such relaxations are common. Those considerations will not play a role in the lecture: runs etc. are *linear* for us (total orderings).

## Example run

- state sequences from runs: idle ready (execute waiting)\*



- corresponding words in  $\Sigma$ :  $start\ run(block, unblock)^*$
- A single state sequence may correspond to more than one word
- non-determinism: the same  $\Sigma$ -word may correspond to different state sequence

### “Traditional” acceptance

**Definition 3.4.4** (Acceptance). An *accepting* run of a finite state automaton  $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$  is a finite run  $\sigma = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} q_n$ , with  $q_n \in F$ .

In the scheduler example from before: a *state sequence* corresponding to an accepting run is

idle ready executing waiting executing end .

The corresponding *word* of labels is

$start\ run\ block\ unblock\ stop$  .

A 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 existence 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 a *word* over  $\Sigma$  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. daemon 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  $q_0 \xrightarrow{a} q_1$  and  $q_0 \xrightarrow{a} q_2$  (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,  $\varphi_1$  describes the language accepted by  $q_1$  and  $\varphi_2$  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 *existence* 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 support acceptance in a best possible way. Of course, they are also “prophetic” in that choosing correctly requires foresight (but angels can do that...). 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  $\forall$ -quantification. The duality between those two forms of non-determinism 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 environment 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 ( $\exists$ ) 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 ( $\forall$ ).

## Accepted language

**Definition 3.4.5** (Language). The *language*  $\mathcal{L}(\mathcal{A})$  of automaton  $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$  is the set of words over  $\Sigma$  that correspond to the set of all the accepting runs of  $\mathcal{A}$ .

- generally: *infinitely* many words in a language
- remember: regular expressions etc.

For the given “scheduler automaton” from before, one can capture the language of finite words by (for instance) the following regular expression

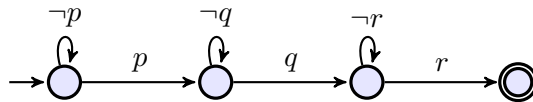
$$\text{start run } ((\text{preempt run})^* \mid (\text{block unblock})^*) \text{ stop .}$$

In the context of language theory, *words* are finite sequences of letters from an alphabet  $\Sigma$ , i.e., a word is an element from  $\Sigma^*$ , and languages are sets of words, i.e., subsets of  $\Sigma^*$ . For LTL and related formalisms, we are concerned with *infinite words* and languages over infinite words.

### Reasoning about runs

**Sample correctness claim (positive formulation)** If first  $p$  becomes true and afterwards  $q$  becomes true, then afterwards,  $r$  can no longer become true

**Seen negatively** It's an **error** if in a run, one sees first  $p$ , then  $q$ , and then  $r$ .



- reaching accepting state  $\Rightarrow$  correctness property **violation**
- accepting state represents **error**

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.

As said, operating with the negated specification, that's typical for the approach. What is specific and atypical is that one can represent the property violation (i.e., the negated formula) referring to *finite* sequences and thus capture it via conventional automata. In the general case, that is not possible. 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 existence 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 [23]. That “bad” in the sentence refers to the *negation* of the original property one wishes to establish (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 path 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).

### Comparison to FSA in “standard” language theory

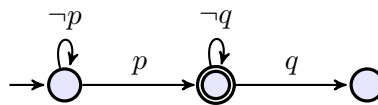
- remember classical FSA (and regular expressions)
- for instance: *scanner* or *lexer*
- (typically infinite) languages of finite words
- remember: accepting runs are finite
- in “classical” language theory: **infinite** words completely out of the picture

### Some liveness property

“if  $p$  then eventually  $q$ .”

### Seen negatively

It’s an **error** if one sees  $p$  and afterwards never  $q$  (i.e., forever  $\neg q$ )



- violation: only possible in an **infinite** run
- not expressible by *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 visiting states *infinitely* often.

In the above example, it would seem that acceptance could be “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  $\epsilon$ -transitions (silent transitions), simply because the construction is more straightforward (compositional). Generalizing Büchi automata to GBAs does not involve  $\epsilon$ -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

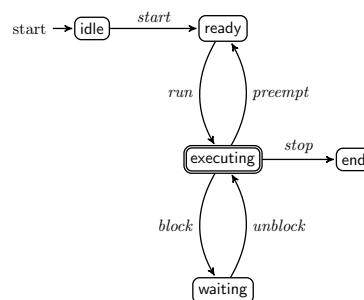
#### Büchi acceptance

- infinite run: often called  $\omega$ -run (“omega run”)
- corresponding acceptance properties:  $\omega$ -acceptance
- different versions: Büchi, Muller, Rabin, Streett, parity etc., acceptance conditions
  - Here, for now: **Büchi acceptance** condition [9] [8]

**Definition 3.4.6** (Büchi acceptance). An *accepting  $\omega$ -run* of finite state automaton  $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$  is an infinite run  $\sigma$  such that some  $q_i \in F$  occurs infinitely often in  $\sigma$ .

Automata with this acceptance condition are called **Büchi automata**.

#### Example: “process scheduler”



- accepting  $\omega$ -runs
- $\omega$ -language

**infinite state sequence**  $\text{idle (ready executing)}^\omega$

$\omega$ -word  $start (run preempt)^\omega$

When describing languages over infinite words, one often uses the symbol  $\omega$  to stand for “infinity” (in other contexts, as well. Actually  $\omega$  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  $\epsilon$ , 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  $\Sigma$ ,  $\Sigma^\omega$  represents all infinite words over  $\Sigma$ . As a side remark: for non-trivial  $\Sigma$  (i.e., with more than 2 letters), the set  $\Sigma^\omega$  is no longer enumerable (it’s a consequence of the simple fact that its cardinality is larger than the cardinality of the natural numbers).

Sometimes, one finds the notation  $\Sigma^\infty$  (or  $(ab)^\infty \dots$ ) to describe infinite *and* finite words. Remember in that context, that the semantics of LTL formulas is defined over *infinite* sequences (paths), only.

In the above example, the “process scheduler” corresponds to one we have seen before. Now, however, with a *different* state marked as accepting. The automaton is meant to illustrate the notion of Büchi-acceptance; it’s not directly mean as some specific logical property (or a negation thereof), nor do we typically think that transition systems representing the “program” work as language acceptors and thus have specific accepting states they have to visit infinitely often.

### Generalized Büchi automata

**Definition 3.4.7** (Generalized Büchi automaton). A *generalized Büchi automaton* is an automaton  $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ , where  $F \subseteq 2^Q$ .

Let  $F = \{f_1, \dots, f_n\}$  and  $f_i \subseteq Q$ . A run  $\sigma$  of  $\mathcal{A}$  is *accepting* if

$$\text{for each } f_i \in F, \text{ inf}(\sigma) \cap f_i \neq \emptyset.$$

- $\text{inf}(\sigma)$ : states visited infinitely often in  $\sigma$
- generalized Büchi automaton: multiple accepting sets instead of only one ( $\neq$  “original” Büchi Automata)
- generalized Büchi automata: *equally* expressive

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  $\omega$ -automata. There are other acceptance conditions (Rabin, Streett, Muller, parity  $\dots$ ), which we will probably 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  $\omega$ -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 automata is also known as  $\omega$ -regular languages.

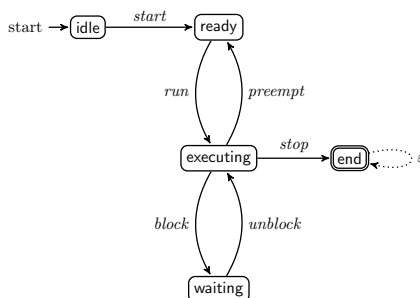
## Stuttering

- treat finite and infinite acceptance uniformly
- finite runs as infinite ones, where, at some point, infinitely often “nothing” happens (**stuttering**)
  - Let  $\varepsilon$  be a predefined nil symbol
  - alphabet/label set extended to  $\Sigma + \{\varepsilon\}$
  - extend a finite run to an equivalent infinite run: keep on stuttering after the end of run. The run must end in a final state.

**Definition 3.4.8** (Stutter extension). The *stutter extension* of a finite run  $\sigma$  with final state  $s_n$ , is the  $\omega$ -run

$$\sigma (s_n, \varepsilon, s_n)^\omega \quad (3.1)$$

## Stuttering example



The “process scheduler” example is now again used with a the “natural end state” as accepting. Examples of accepting state sequences resp. accepting words corresponding to an accepting  $\omega$ -run include the following:

idle ready executing waiting executing end $^\omega$

and

start run block unblock stop $^\omega$

So far, we have introduced the stutter extension of a (finite) 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 be 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.

Note also: the end-state is *isolated*!

It's best, however, *not* to see the automaton itself as Büchi-automaton. Or all states are accepting?

Anyway, here is the "automaton" from before again. Note that we have marked the "end" state as accepting state again. Since the automaton represents the system, perhaps

It again should be noted that the automaton or transition system here is not such much intended or motivated as a "language acceptor": it's supposed to capture (an abstraction of) the behavior of a program or process, and in that setting one typically does not speak of "accepting states", it may have a end state, where the program terminates, but that's often not interpreted as "accepting a finite or infinite language".

### 3.4.3 Something on logic and automata

#### From Kripke structures to Büchi automata

- LTL formulas can be interpreted on sets of infinite runs of Kripke structures
- Kripke structure/model:
  - "automaton" or "transition system"
  - transitions unlabelled (typically)
  - states (or worlds): "labelled", in the most basic situation: sets of propositional variables

We have encountered different "transition-system formalisms". One under the name Kripke models (or Kripke structures), the other one automata (especially those with Büchi acceptance). [2] talk about transition systems instead of Kripke structures (and they allow labels on the transitions, as well). The Kripke structures or transition systems are there to model "the system" whereas the "automaton" is there to describe the behavior (the language, i.e., infinite words over sets of propositions). 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 that 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 satisfaction 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 check that the “language” of the system  $M$  is a subset of the language described by a LTL-specification  $\varphi$ , like  $M \models \varphi$  corresponds to  $\mathcal{L}(M) \subseteq \mathcal{L}(\varphi)$ . 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,  $\mathcal{L}(M)$  is a “language” of infinite words corresponding to sequences of “states” and the state-attached information. On the other hand,  $\mathcal{L}(\varphi)$  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 doing 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 straightforward). This 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 description 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.

### BA vs. KS

- “subtle” differences
- labelled transitions vs. labelled states
- easy to transform one representation into the other
- here: from KS to BA.
  - states: basically the same
  - initial state: just make a unique initial one
  - transition labels: all possible combinations of atomic props

- states and transitions: transitions in  $\mathcal{A}$  allowed if
  - \* covered by accessibility in the KS (+ initial transition added)
  - \* transition **labelled** by the “post-state-labelling” from KS

## KS to BA

Given  $M = (W, R, W_0, V)$ . An automaton  $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$  can be obtained from a Kripke structure as follows

**transition labels:**  $\Sigma = 2^{AP}$

**states:**

- $Q = W + \{i\}$
- $q_0 = i$
- $F = W + \{i\}$

**transitions:**

- $s \xrightarrow{a} s'$  iff  $s \rightarrow_M s'$  and  $a = V(s')$   $s, s' \in W$
- $i \xrightarrow{a} s \in T$  iff  $s \in W_0$  and  $a = V(s)$

We call the “states” here now  $W$  for worlds, to distinguish it from the states of the automaton. We write  $\rightarrow_M$  the accesibility relation in  $M$ , to distinguish it from the “labelled transitions” in the automaton.

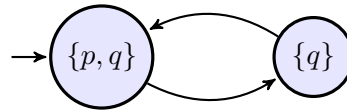
Note: all states are accepting states (which is an ingredient of a Büchi automaton). Basically, the accepting conditions are not so “interesting” and making all states accepting means: I am interesting in *all* behavior as long as it’s infinite. The KS (and thus the corresponding BA) is not there to “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 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.

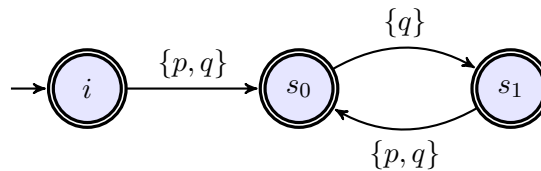
Note also, that the valuations  $V : W \rightarrow (AP \rightarrow \mathbb{B})$  attach to each world or “state” a mapping, that assigns to each atomic proposition from  $AP$  a truth value from  $\mathbb{B}$ . That can be equivalently seen as attaching to each world a *set* of atomic propositions, i.e., it can be seen as of type  $W \rightarrow 2^{AP}$ . Perhaps confusing the assignment of (here Boolean values) to atomic propositions, i.e., functions of type  $AP \rightarrow \mathbb{B}$ , 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: KS to BA**

A Kripke structure (whose only infinite run satisfies (for instance)  $\Box q$  and  $\Box\Diamond p$ ):

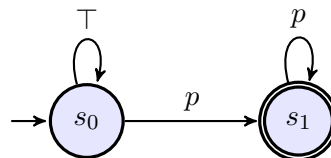


The corresponding Büchi automaton:

**From logic to automata**

- cf. regular expressions and FSAs
- for any LTL formula  $\varphi$ , there exists a Büchi automaton that accepts precisely those runs for which the formula  $\varphi$  is satisfied

**stabilization: “eventually always  $p$ ”,  $\Diamond\Box p$ :**



We will see the algorithm later ...

**(Lack of?) expressiveness of LTL**

- note: analogy with regular expressions and FSAs: **not 100%**
- in the finite situation: “logical” specification language (regexp) correspond fully to machine model (FSA)
- here: LTL is **weaker!** than BAs
- $\omega$ -**regular** expressions +  $\omega$ -regular languages
- generalization of regular languages
- allowed to use  $r^\omega$  (not just  $r^*$ )

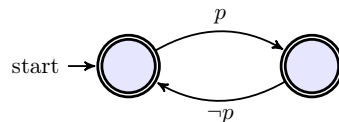
**Generalization of RE / FSA to infinite words**  $\omega$ -regular language correspond to NBAs

There exist a “crippled” form of “infinite regular expressions” that is an exact match for LTL (but not relevant here).

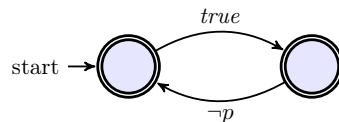
**$\omega$ -regular properties strictly more expressive than LTL**

**Temporal property**  $p$  is always false after an *odd* number of steps

$$p \wedge \Box(p \rightarrow \bigcirc \neg p) \wedge \Box(\neg p \rightarrow \bigcirc p)$$



$$\exists t. t \wedge \Box(t \rightarrow \bigcirc \neg t) \wedge \Box(\neg t \rightarrow \bigcirc t) \wedge \Box(\neg t \rightarrow p)$$



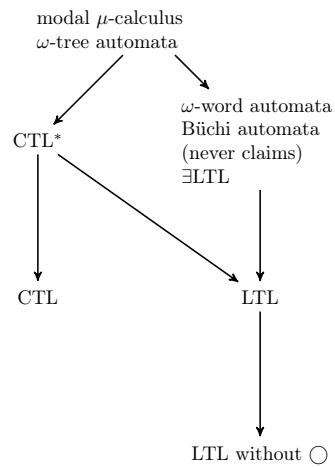
As mentioned, there is a “mismatch” between LTL and Büchi-automata (resp.  $\omega$ -regular expressions): LTL is strictly weaker. One can live with that, of course, but both formalisms seems kind of natural, so one can try to repair the mismatch (or at least analyze the reason of the mismatch). The slides gives an example of an  $\omega$ -regular property which is *not* expressible in LTL (we don’t prove that it’s impossible, though).

To “repair” the mismatch, one could do two things. One can ask, what needs to be taken away from BAs resp.  $\omega$ -regular expressions to make them fit to LTL, resp. is there an automaton model that exactly fits LTL? Alternatively one can ask: what needs to be added to LTL to make it as expressing as BAs? The slide hints at 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 of LTL is sometimes called “existential LTL” or  $\exists$ LTL.

The Spin model checker resp. its input language Promela offers another mechanism that gives the same expressivity to LTL as  $\omega$ -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/>.

## Expressiveness



### 3.4.4 Implications for model checking

#### Core part of automata-based MC

- remember: MC checks “system against formula”  $S \models \varphi$

#### Linear time approach

- $\omega$ -language of the behavior of  $S$  is *contained* in the language allowed by  $\varphi$
- core idea then: instead of

$$\mathcal{L}(S) \subseteq \mathcal{L}(P_\varphi)$$

do

$$\mathcal{L}(S) \cap \overline{\mathcal{L}(P_\varphi)} = \emptyset$$

where  $S$  is a model of the system  $P_\varphi$  represents the property  $\varphi$

Compare: refutation in logic.

#### What’s needed for automatic MC?

$$\mathcal{L}(S) \cap \overline{\mathcal{L}(P_\varphi)} = \emptyset$$

**Algorithms needed for**

1. translation LTL to Büchi
  2. language **emptiness**: are there any accepting runs?
  3. language **intersection**: are there any runs accepted by two or more automata?
  4. language **complementation**
- thankfully: all that is *decidable*

As explained earlier, a minor point is also that one prefers to bridge the mismatch between transition systems on the one hand and Büchi-automata on the other. That's done by slightly massaging the state-labeled TS (or Kripke structure) into a BA representation, as presented before. But that is a minor cosmetic thing.

**How could one do it, then?**

- *system* represented as Büchi automaton  $A$ 
  - The automaton corresponds to the *asynchronous* product of automata  $A_1, \dots, A_n$  (representing the asynchronous processes)

$$A = \prod_{i=1}^n A_i$$

- *property* originally given as an LTL formula  $\varphi$
- translate  $\varphi$  into a Büchi automaton  $B_\varphi$
- check

$$\mathcal{L}(A) \cap \overline{\mathcal{L}(B)} = \emptyset$$

In this slide (and in some of the following) we focus on the pure “logical” aspects of model checking (language inclusion etc.). As discussed, model checking is most profitably used (for various reasons) when establishing or refuting properties of *concurrent systems*. So, when trying to model check  $M \models \varphi$  we are often encountering a situation where the system or model is not given as a “flat” or unstructured big transition system. Instead, it's given by processes running in parallel or concurrently. Therefore,  $M$  consists of a number of individual transition systems or automata, “running in parallel”. In standard “language-theoretical” settings, like using automata modelling lexers or similar, the *parallel* composition of automata plays not much of a role. In language-theoretic terms, it's also known as *shuffle product* of languages. For us, the parallel composition of processes is, of course, important. The particular construction is called here *asynchronous product* of automata. Whether that's a good name, one may debate (later). One can see it as exploring all possible *interleavings* if the steps of the parallel automata or transition system.

### Better avoid complementation

In practice (e.g., in SPIN): avoid automata **complementation**:

- Assume  $A$  as before
- The **negation** of the property  $\varphi$  is automatically translated into a Büchi automaton  $\overline{B}$  (since  $\mathcal{L}(\overline{B}) \equiv \mathcal{L}(B)$ )
- By making the **synchronous** product of  $A$  and  $\overline{B}$  ( $\overline{B} \otimes A$ ) we can check:

$$\mathcal{L}(A) \cap \mathcal{L}(\overline{B}) = \emptyset$$

- If intersection is empty:  $A \models \varphi$ , i.e., “property  $\varphi$  holds for  $A$ ” or “ $A$  satisfies property  $\varphi$ ”
- else:
  - $A \not\models \varphi$
  - **bonus**: accepted word in the intersection **counter example**

### 3.4.5 Automata products

The section discusses two kinds of automata constructions, i.e., composition operators on automata, called (here) *synchronous* and *asynchronous* product. In the context of LTL model checking and the lecture, both serve two different purposes. The synchronous product will capture language *intersection*, intersecting the language of automaton representing an LTL formula (the negated property to verify). It corresponds to the standard product construction perhaps known from standard FSAs. Only now we build the product between Büchi-automata. Actually, we are interested in the product or intersection only between two Büchi-automata, where one is special insofar that it comes from turning the program into such an automaton. Thereby, one of the two automata has *only* accepting states (and never gets really stuck since it’s stuttering). In that sense, the way the synchronous product is used here is *asymmetric*.

The *asynchronous* product here comes from the underlying computation model. The presentation here is based in [20] and thus influenced by the choices as made in the Spin model checker. In Spin, systems consist typically of a number of processes running in parallel. The processes are programmed in a C-like notation. 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 the channels, processes independently do their steps. 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. However, a particular form of concurrency, namely independently or *asynchronously*. 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 synchronously in parallel (with the lock as global synchronizing mechanism). In our setting here, processes run under an interleaving model (“asynchronously”) and the composition of the system with the “formula” is done synchronously.



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 communicate 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*. That's why Spin's "programming language" is basically processes + channel. 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).

### Two kinds of products

- conceptually standard (but see terminating condition = definition of final states)

#### asynchronous

- prog's running in parallel
- **interleaving**
- no synchronization!
- one automaton does something, the others not

#### synchronous

- together with (the automaton representing) the formula
- lock-step
- however: *stuttering*.

Deviating thereby from the classical notion of synchronous product.

### Asynchronous product

**Definition 3.4.9** (Asynchronous product). The *asynchronous product* of two automata  $A_1$  and  $A_2$ , (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
- $F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ or } q_2 \in F_2\}$ .

---


$$\frac{q_1 \rightarrow_1 q'_1}{(q_1, q_2) \rightarrow (q'_1, q_2)} \text{PAR}_1 \qquad \frac{q_2 \rightarrow_2 q'_2}{(q_1, q_2) \rightarrow (q_1, q'_2)} \text{PAR}_1$$


---

We define the product 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 of 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  $\Sigma_1$  and  $\Sigma_2$  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 automaton, 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.

### 3n+1 inspired example

- **3n+1 problem**

- Assume 2 non-terminating asynchronous processes  $A_1$  and  $A_2$ :
- $A_1$  tests whether the value of a variable  $x$  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  $x/2$

**Question** Does the corresponding function *terminate* for all inputs  $x$ ?

- Let  $\varphi$  the following property:  $\Box \Diamond (x \geq 4)$  (negated  $\Diamond \Box (x < 4)$ )

In the problem, “termination” is meant reaching the endless cycle  $4 \rightarrow 2 \rightarrow 1 \rightarrow 4 \dots$

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  $P_1 \parallel \dots \parallel P_n$  consisting of an arbitrary number  $n$  of copies of the same system, where all the  $P_i$  are identical (perhaps up-to their process identity). Think of using model checking mutual exclusion not for an implementation of the the 5-philosophers problem but for the  $n$ -philosophers problem.

### Example: async product

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 mathematical core definition in that the transitions are *labelled*. The labels are **not** 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 accordingly 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 intension 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 is Promela resembles quite C, but that's a 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 assignment. In other words, the above snippet can be simplified to  $x = 3x+1$ . Now, in Spin, the first expression is interpreted as *guard*: in the tradition of C, the expression  $x\%2$  is calculated. In case the outcome is 0, it's interpreted 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 insofar, the transition is enabled or not depending on the “circumstances”). Of course, the circumstances, 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.

The explanation should be enough to understand the example. A few words on “guards” in concurrent programs might still be of more 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 powerful 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  $\sim\text{await}(b)\$$  or similar, making that transparent.

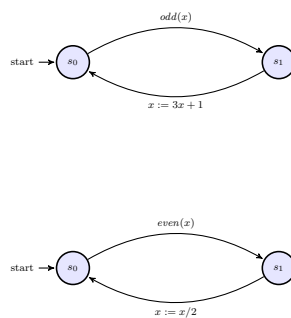
For people experienced in concurrency programming there is another concerning issue wrt. to guards *odd* or similar. The previous discussion concentrated on “syntactical” issues or language pragmatic (like questioning the wisdom of avoiding special syntax). The point here may be even more serious. In the transition systems below (and in the code snippet above), there are two steps, namely first the guard is checked, 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 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 language or language, Spin offers the possibility enforce atomicity (there are slightly different ways which don't concern us here). Basically one can *group* the two steps together,  $[odd(x); x := 3x + 1]$  where we use brackets [ and ] to denote that the execution should be atomic resp., without interleaving (Spin would use keywords like `atomic{...}` or `d_step{...}`). In any case: the shown pattern

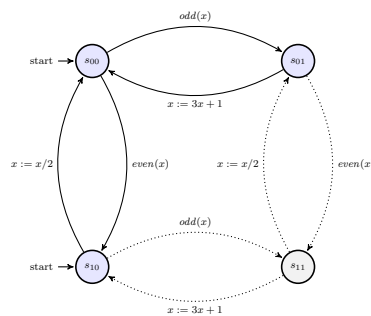
is rather common: in an atomic section, the first one is the guard, and the rest is the effect. It is not so comming (and a bit tricky) to use guards in the middle of an atomic section. Since that pattern is so common, other languages would offer specially syntax for it `await (b) {statements}` and there 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 count then have labels of the general formal  $g \triangleright a$ , 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.

### $A_1$ and $A_2$



### $A_1 \times A_2$



In the product automaton, the state  $s_{11}$  is *unreachable*. When starting at the initial state, the dotted arrows can never be taken.

### Tests or guards on transitions

- guarded commands (thanks to Dijkstra)
- conditional transitions, predicated on a guard
- Promela semantics, an expression statement has to evaluate to non-zero to be executable (*enabled*). So to test whether a variable  $x$  is even, we write  $!(x\%2)$  and  $x\%2$  for checking whether  $x$  is odd.

E.g.: given  $x=4$ ,  $!(4\%2)$  evaluates to  $!(0)$  or written more clearly as  $!(false)$  which is  $(true)$ .

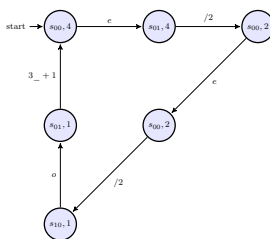
### Example: Pure automaton

We have made use of the more user-friendly 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 *concrete* (non-abstract, non-symbolic) models. So we need to pick a concrete initial value for  $x$ . The picture 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” now consists of the “control-flow state” (for instance  $(s_0, s_0)$  written also  $s_{00}$ ) together with the state of the memory, i.e., the value of  $x$ . In the pure transition system, there are 3 states whose control-flow state, which can be seen as the value of the program-counter(s) is  $s_{00}$ , namely for  $x = 4$ ,  $x = 2$  and  $x = 1$ .

In the states,  $s_{00}$  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 labelling of the edges can be ignored; they are needed only for the non-pure representation, in the pure transition system they are just kept for clarity. 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, 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 reader 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.



### Remarks

- Not all the states in the product necessarily reachable from  $q_0$ . (Their reachability depends on the semantics given to the labels in  $A.L$  The interpretation of the labels depends on Promela semantics as we'll see in a future lecture)

- The transitions in the asyc. product automaton are the transitions from the component automata arranged such that only *one* of the components automata can execute at a time  $\Rightarrow$  interleaving
- Promela has also *rendez-vous* synchronization (A special global variable has to be set)
- Some transitions may synchronize by sending and receiving a message
- For hardware verification, the asynchronous product is defined differently: **each** of the components with enabled transitions is making a transition (simultaneously). And, in my humble opinion, should then better *not* be called asynchronous product, but synchronous.

From Holzmann pp.554. see also Peled pp. 90

### Synchronous product

In the following we define the *synchronous product* for 2 Büchi-automata *in a special case*. The special case is that for one of the automata, all its states are accepting. That 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.

Actually, the general case for two arbitrary BAs is slightly more complex. The slightly tricky part is how to define the accepting states of the product. In principle, part from that, the definition is straightforward. The general intention of such a “product” construction is that the 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 common word is accepted, if both  $A_1$  and  $A_2$  reach a respective accepting state. That is easy. Now that 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 re-visit they resp. 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 synchronous product is slightly more elegant compared to standard BAs.

**Definition 3.4.10** (Synchronous 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:

- $Q = Q_1 \times Q_2$

- $q_0 = (q_{01}, q_{02})$
- $\Sigma = \Sigma_1 \times \Sigma_2$ .
- $\rightarrow = \rightarrow_1 \times \rightarrow_2$
- $(q_1, q_2) \in F$  if  $q_2 \in F_2$

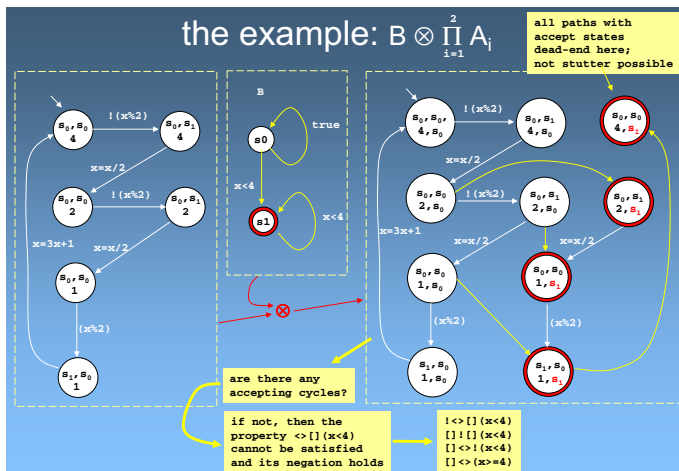
**(finish later) Let the system automaton stutter**

- asymmetric situation
- one automaton: “system”
- second one:
  - “recognizer”
  - automaton that represents the logical LTL formula
- for **system** automating: add *stuttering*
- stutter: a self-loop labeled with  $\varepsilon$  at every every state in without outgoing transitions

**Definition 3.4.11** (Stuttering synchronous product). The *synchronous* product of two finite automata  $P$  and  $B$  (written  $P \otimes B$  is defined as finite state automaton  $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$  where:

- $Q = Q'_1 \times Q_1$ , where  $P'$  is the *stutter closure* of  $P$ 
  - A self-loop labeled with  $\varepsilon$  is attached to every state in  $P$  without outgoing transitions in  $P.T$ )
- $A.s_0$  is the pair  $(P.s_0, B.s_0)$
- $A.L$  is the set of pairs  $(l_1, l_2)$  such that  $l_1 \in P'.L$  and  $l_2 \in B.L$
- $A.T$  is the set of pairs  $(t_1, t_2)$  such that  $t_1 \in P'.T$  and  $t_2 \in B.T$
- $A.F$  is the set of pairs  $(s_1, s_2)$  such that  $s_1 \in P'.F$  or  $s_2 \in B.F$

**Example: synch. product for  $3n + 1$  system and property**



- We require the stutter-closure of  $P$  (as  $P$  is a finite state automaton (the asynchronous product of the processes automata) and  $B$  is a standard Büchi automaton obtained from a LTL formula
- Not all states necessarily reachable from  $q_0$



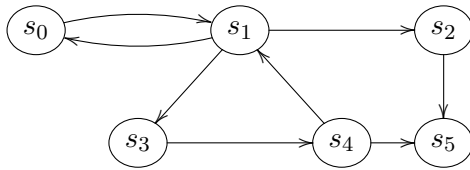


Figure 3.1: Strongly connected component

- Main difference between asynchronous and synchronous products: labels and transitions. for synchronous product:
  - *joint* transitions of the component automata
  - labels are pairs: the combination of the two labels of the original transitions in the component automata
- In general here  $P \otimes B \not\equiv B \otimes P$ , but given that in SPIN  $B$  is particular kind of automaton (labels are state properties, not actions), we have then  $P \otimes B \equiv B \otimes P$

## 3.5 Model checking algorithm

### 3.5.1 Preliminaries

#### Algorithmic checking for emptiness

- for FSA: emptiness checking is easy: *reachability*
- For Büchi:
  - more complex acceptance (namely  $\omega$ -often)
  - simple, one time reachability not enough

$\Rightarrow$  “repeated” reachability

$\Rightarrow$  from initial state, reach an accepting state, and then again, and then again ...

- cf. “lasso” picture
- technically done with the help of SCCs.

#### Strongly-connected components

**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.

maximality.

#### SCC example

- Strongly-connected subsets:  $S = \{s_0, s_1\}$ ,  $S' = \{s_1, s_3, s_4\}$ ,  $S'' = \{s_0, s_1, s_3, s_4\}$
- Strongly-connected components: Only  $S'' = \{s_0, s_1, s_3, s_4\}$

### Checking emptiness

Büchi automaton  $A = (Q, s_0, \Sigma, \rightarrow, F)$  with **accepting run**  $\sigma$

As  $Q$  is finite, there is some suffix  $\sigma'$  of  $\sigma$  s.t. every state on  $\sigma'$  is reachable from any other state on  $\sigma'$

- I.a.w: those set of states is strongly connected.
- This set is reachable from an initial state and contains an accepting state

Checking non-emptiness of  $\mathcal{L}(A)$  is equivalent to finding a SCC in the graph of  $A$  that is reachable from an initial state and contains an accepting state

### Emptiness checking and counter example

- different algos for SCC. E.g.:
  - Tarjan's version of the *depth-first search* (DFS) algorithm
  - SPIN *nested depth-first search* algorithm
- If the language  $\mathcal{L}(A)$  is non-empty, then there is a *counterexample* which can be represented in a finite way
  - It is *ultimately periodic*, i.e., it is of the form  $\sigma_1\sigma_2^\omega$ , where  $\sigma_1$  and  $\sigma_2$  are finite sequences

## 3.5.2 The algorithm

### Model checking algorithm

- Let  $A$  be the automaton specifying the system and  $\overline{B}$  the automaton corresponding to the negation of the property  $\varphi$
1. Construct the intersection automaton  $C = A \cap \overline{B}$
  2. Apply an algorithm to find SCCs reachable from the initial states of  $C$
  3. If none of the SCCs found contains an accepting state
    - The model  $A$  satisfies the property/specification  $\varphi$
  4. Otherwise,
    - a) Take one strongly-connected component  $SC$  of  $C$
    - b) Construct a path  $\sigma_1$  from an initial state of  $C$  to some accepting state  $s$  of  $SC$
    - c) Construct a cycle from  $s$  and back to itself (such cycle exists since  $SC$  is a strongly-connected component)
    - d) Let  $\sigma_2$  be such cycle, excluding its first state  $s$
    - e) Announce that  $\sigma_1\sigma_2^\omega$  is a counterexample that is accepted by  $A$ , but it is not allowed by the property/specification  $\varphi$

### 3.5.3 LTL to Büchi

#### LTL to Büchi

- translation to Generalized Büchi GBA
- cf. Thompson's construction
- *structural* translation
- crucial idea: connect semantics to the syntax.
- compare Hintikka-sets or similar constructions for FOL

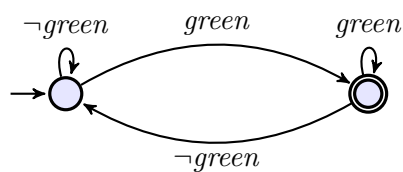
#### Source and terminology: Baier and Katoen [2]

- **transition systems** TS:
  - corresponds to Kripke systems
  - state-labelled (transition labels irrelevant)
  - labelled by sets of atomic props:  $\Sigma = 2^{AP}$
  - “language” or behavior of the TS: (**traces**): infinite sequences over  $\Sigma$

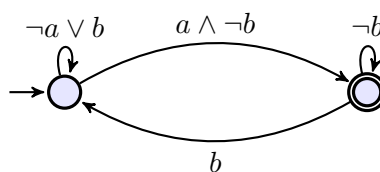
#### Illustrative examples (5.32)

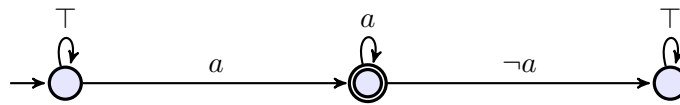
1.  $\Box\Diamond green$
2.  $\Box(request \rightarrow \Diamond response)$
3.  $\Diamond\Box a$

$\Box\Diamond green$



$\Box(request \rightarrow \Diamond response)$



$\diamond \square a$ **Reminder: Generalized NBA**

- equi-expressive than NBA
- used in the construction
- different way of defining acceptance
  - acceptance: set of *acceptance sets* = set of sets of elements of  $Q$ .
  - acceptance: each acceptance set  $F_i$  must be “hit” infinitely often

**Basic idea for  $\mathcal{G}_\varphi$** 

- not the construction yet, but: “insightful” property
- find a mental picture:
  - what are the states of the automaton
  - (and how are they connected by transitions)
- $A_i \in \Sigma$ , sets of atomic props
- $B_i$  : “extended” (by sub-formulas of  $\varphi$ ), i.e.,  $B_i \supseteq A_i$ .

Namely those that are intended to be in the “language of that state”. I.e., the  $B_i$ ’s form the states of  $\mathcal{G}_\varphi$ .

Given  $\sigma = A_0A_1A_2 \dots \in \mathcal{L}(\varphi)$ .

Extension to  $\hat{\sigma} = B_0B_1B_2 \dots$

$$\psi \in B_i \quad \text{iff} \quad \underbrace{A_i, A_{i+1}A_{i+2} \dots}_{\sigma^i} \models \psi$$

$\hat{\sigma} = \text{run}$  (ultimately: state-sequence) in  $\mathcal{G}_\varphi$

**Rest****Cf. FSAs**

- states as “sets” of “words” (language resp. set of ltl formulas)
- cf. Myhill-Nerode
- a bit different, (equivalence on languages of finite words)
- represent states by equivalence classes of words

## Closure of $\varphi$

- related to Fisher-Ladner closure
- See page 276
- “states”  $A_i$  from the mental picture
- what’s a “closure” in general?
- Extending  $A_i$  to  $B_i$  not by *all* true formulas, but only those that could *conceivably play a role* in an automaton checking  $\varphi$
- $\Rightarrow$  achieving “finiteness” of the construction

## How to extend $A_i$ 's

- not by irrelevant stuff (closure of  $\varphi$ ).
- two other conditions:
  - avoid contradictions (*consistency*)
  - include logical consequences<sup>3</sup> (*maximality*)
- maximally consistent sets! (here called *elementary*)
- in one state: local perspective only (but don't forget  $U$ )
- Cf: KS has an interpretation for each  $AP$ , here now (in the intended BA), “interpretation” for *all relevant formulas* “in” each state (subformulas of  $\varphi$  and their negation)

## Elementary sets/maximally consistent sets

- given  $\varphi$
- **elementary**: “maximally consistent set of subsets (of the closure of  $\varphi$ )”
- **consistent**: “no obvious contradictions”
- maximally consistent: sets for formulas  $\psi$  in the closure of  $\varphi$  s.t., there exists *some* path  $\pi$  s.t.  $\pi \models \psi$ .
  - wrt. propositional logic
  - *locally consistent* wrt. until
- “maximal”

**Example:**  $\varphi = a \ U \ (\neg a \wedge b)$

$$\{a, b\} \subseteq \text{closure}(\varphi)$$

$$\{a, b, \neg a, \neg b, \neg(\neg a \wedge b), \neg a \wedge b, \varphi, \neg\varphi\}?$$

---

<sup>3</sup>hence the notion of “closure”

**Example:**  $\varphi = a U (\neg a \wedge b)$

$$\sigma = \{a\}\{a,b\}\{b\}\dots = A_0A_1A_2\dots$$

- Extending (for example):  $A_0$  to  $B_0$
- extending  $\sigma$  to  $\hat{\sigma}$

### Construction of GNBA: general

- given  $AP$  and  $\varphi$
- given  $\varphi$ , construct an GNBA such that

$$\mathcal{L}(B) = \text{words}(\varphi)$$

- 3 core ingredients
  1. **states** = sets of formulas which (are supposed to) “hold” in that state
  2. transition relation: connect the states appropriately,
  3. transitions labelled by sets of  $AP$ .
- labeled transition connected states to match the semantics: for  $\bigcirc\varphi$ :  
go from a state containing  $\bigcirc\varphi$  to a state containing  $\varphi$ . Label the transition with the APs from the start state.

### Transition relation

$$\delta : Q \times 2^{AP} \rightarrow 2^Q$$

- if  $A \neq B \cap AP$ :  $\delta(B, A) = \emptyset$
- if  $A = B \cap AP$ , then  $\delta(B, A)$  is the set  $B'$  such that
  - for every  $\bigcirc\psi \in \text{closure}(\varphi)$ :

$$\bigcirc\psi \in B \quad \text{iff} \quad \psi \in B'$$

- for every  $\varphi_1 U \varphi_2 \in \text{closure}(\varphi)$ :

$$\varphi_1 U \varphi_2 \in B \quad \text{iff} \quad \begin{array}{l} \varphi_2 \in B \\ (\varphi_1 \in B \quad \text{and} \quad \varphi_1 U \varphi_2 \in B') \end{array} \quad \text{or}$$

### Accepting states

$$F_{\varphi_1 U \varphi_2} = \{B \in Q \mid \varphi_2 \in B \text{ or } \varphi_1 U \varphi_2 \notin B\}.$$

### 3.5.4 Rest

## 3.6 Final Remarks

### 3.6.1 Something on Automata

#### Kripke Structures and Büchi Automata

##### *Observation*

- In Peled's book "Software Reliability Methods" Peled [27] the definition of a Büchi automaton is very similar to our Kripke structure, with the addition of acceptance states
  - There is a labeling of the states associating to each state a set of subsets of propositions (instead of having the propositions as transition labels)
- We have chosen to define Büchi Automata in the way we did since this definition is compatible with the implementation of SPIN
  - It was taken from Holzmann's book "The SPIN Model Checker" Holzmann [20]

#### Automata Products

##### *Observation*

- We have defined synchronous and asynchronous automata products with the aim of using SPIN (based on Holzmann's book)
  - The definition of asynchronous product is intended to capture the idea of (software) asynchronous processes running concurrently
  - The synchronous product is defined between an automaton specifying the concurrent asynchronous processes and an automaton obtained from an LTL formula (or obtained from a Promela never claim)
  - The purpose for adding the stutter closure (in the definition of the synchronous product) is to make it possible to verify both properties of finite and infinite sequences with the same algorithm
- I.e., you might find different definitions in the literature!
  - In particular, in Peled's book the automata product is defined differently, since the definition of Büchi automata is different

## Further reading

### Further reading

- The first two parts of this lecture were mainly based on Chap. 6 of Holzmann's book "The SPIN Model Checker"

- Automata products: Appendix A
- The 3rd part was taken from Peled's book

**For next lecture:** Read Chap. 6 of Peled's book, mainly section 6.8 on translating LTL into Automata

- We will see how to apply the algorithm to an example



# Chapter 4

## $\mu$ -calculus model checking

### Learning Targets of this Chapter

The chapter covers an short intro to the (resp. one variant) of the  $\mu$ -calculus and model-checking it. We focus on the most prominent version of the  $\mu$ -calculus for model checking known as modal  $\mu$ -calculus, with a “branching time” interpretation. The logic can be understood as the “prototypical” **logic with fixpoints**, so we’ll have to talk about fixpoints, as well. For model checking, we look at a bit of “game theory” (parity games).

### Contents

4.1	Introduction . . . . .	117
4.2	Propositional $\mu$ -calculus: syntax and semantics . . . . .	124
4.3	Model checking . . . . .	134

What  
is it  
about?

## 4.1 Introduction

The presentation takes information from the handbook article Bradfield and Walukiewicz [7], but cannot cover all of the theoretical background in there (and there’s a lot, due to the rather fundamental nature of the  $\mu$ -calculus). There are many starting points that have led to what here is called  $\mu$ -calculus. In computer science, one important reference point is the article Kozen [22].

### Intro remarks

- rather fundamental logic
- central to  $\mu$ -calculus: **fixpoints**
- many variations (and names)
  - propositional  $\mu$ -calculus
  - modal  $\mu$ -calculus
  - Hennessy-Milner logic with recursion
  - . . . .

**For the lecture: vanilla  $\mu$ -calculus** a plain, propositional modal logic + fixpoints

It's not an exaggeration to say, "the"  $\mu$ -calculus is "fixpoint logic" as it's all about fixpoints. Depending on the starting point, one can (and some did) add fixpoints to, for example, first-order logic and what not. For the model checking lecture, we take as a starting point a *modal logic* (like the ones we discussed in the respective chapter). Actually, we are going for a multi-modal logic. The intuition behind the logic is that the modalities talk about transitions (labelled transitions in fact, as we have a multi-modal logics), not about "knowledge" or "beliefs" etc. One can for sure also think of adding fixpoints with such more philosophical interpretations in mind. However, we focus on Kripke structures resp. transition system representing steps in the executions of programs or systems. Like we mostly did for LTL (and CTL etc. covered by student representation), the starting point is also a *propositional* core logic, underneath the modal part (but that also is a bit orthogonal: there will be a student talk about QTL, quantified temporal logic, which allows first-order quantification. One could also turn that into a  $\mu$ -calculus-like formalism). This set-up here is kind of the vanilla propositional  $\mu$ -calculus. We start by recalling what, very generally, a fix-point of a function is. That's actually pretty simple.

**What's a fixpoint?**

$$f : A \rightarrow A$$

$$f(a) = a$$

A fixpoint of a function  $f$  is thus defined quite simple (the  $a$  from set  $A$  in the above example is a fixpoint if  $f$ ). The next part of the lecture is more a warm-up, as a reminder that there are actually fixpoints "everywhere". It's not always explicitly stated, like "let such-and-such be defined as fixpoint in the following way ...", there are other formulations used, and we have encountered such formulations in the lecture already (perhaps without being aware that underlying the respective definition, there was actually a fix-point construction. The definitions were all on the "meta-level" not as part of a logic, i.e., fixpoints were mostly used (implicitly) to define or talk *about* a logic, we did *not* introduce fixpoints as part of the logics (that what the  $\mu$ -calculus does). Actually, it's not 100% correct when saying that so far the logics did not allow to express fixpoints. It will turn out that temporal operators such as "eventually", "always", "until" are effectively fixpoint constructors. Only that there are no explicit fix-point constructors, so their nature as fixpoints is hidden.

**Fixpoints are everywhere**

**A pedestrian definition of syntax** The set  $\Phi$  of *propositional formulas* is given as follows

- all propositional constants from  $AP$  are formulas
- if  $\varphi$  is a formula, then so is  $\neg\varphi$
- if  $\varphi_1$  is a formula and  $\varphi_2$  is a formula, then so is  $\varphi_1 \wedge \varphi_2$
- if  $\varphi_1$  is a formula and  $\varphi_2$  is a formula, then so is  $\varphi_1 \vee \varphi_2$
- ... [more constructs if wished] ...

Is that even a definition?

### Fixpoints are everywhere

**A pedestrian definition of syntax (reformulated)** The set  $\Phi$  of *propositional formulas* is given as follows

- $AP \subseteq \Phi$
- if  $\varphi \in \Phi$ , the  $\neg\varphi \in \Phi$
- if  $\varphi_1 \in \Phi$  and  $\varphi_2 \in \Phi$ , then  $\varphi_1 \wedge \varphi_2 \in \Phi$
- if  $\varphi_1 \in \Phi$  and  $\varphi_2 \in \Phi$ , then  $\varphi_1 \vee \varphi_2 \in \Phi$
- ... [more constructs if wished] ...

### What about that?

$$\Phi = \{p, q, \dots, p \wedge p, p \wedge q, p \wedge (p \vee q) \dots\} \cup \{\mathfrak{5}, p\#q, \neg\mathfrak{5}, \mathfrak{5} \wedge (q\#q) \dots\}$$

The point of that example is that the set  $\Phi$  of formulas given at the end *satisfies* the conditions, but it's not the one one had in mind. Basically, it means, the sentences as given are *not* a definition: they don't precisely *fix* the set of formulas, they just spell out conditions or constraints on  $\Phi$ . Those constraints corresponds to **closure conditions**. For a condition like “if it so happens that  $\varphi \in \Phi$  for some  $\varphi$ , then it's necessary that also  $\neg\varphi$  in  $\Phi$ ”, one says more shorter that the set of formulas is **closed** under  $\neg$ , which is an *unary* operation or constructor.

### How to fix(-point) it?

Depending to the style of writing or conventions in the field, one finds different ways of removing this ambiguity.

- ... **No other** entities are formulas, i.e. elements of  $\Phi$
- $\Phi$  is the **smallest set** such that 1)  $AP \subseteq \Phi$ , 2) ...
- $\Phi$  is **inductively** given by the following conditions: 1) ...

**“Mu”**  $F(S) = AP \cup \{\neg\phi \mid \phi \in S\} \cup \{\varphi_1 \wedge \varphi_2 \mid \varphi_1 \in S, \varphi_2 \in S\} \cup \dots$

$$\Phi = \mu F$$

The last “fix”, the one labelled “Mu”, approaches a definition based on explicit fixpoints. A function  $F$  is defined, which takes a set (the formal parameter  $S$ ) and produces another set, by adding more elements (actually formulas). The the intended set  $\Phi$  is given as the smallest fixpoint of that function  $F$ , i.e., the smallest set  $\Phi$  such that  $\Phi = F(\Phi)$ . The traditional symbol that represents the smallest fixpoint is  $\mu$ , i.e.,  $\Phi = \mu F$ . As a side remark: the “mu” is a reference to some very famous popular science book GEB (Gödel, Escher, Bach. An Eternal Golden Braid), which in an deep, broad, and entertaining

way builds connections between art and logic, and where recursion (i.e., fixpoints) play a central role, and the book also contains something called MU-puzzle).

### Fixpoints are everywhere, indeed

- grammars (with special syntax)

$$\varphi ::= AP \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \dots$$

- Kleene star and regular expressions:
  - finite words over  $\Sigma$ : written  $\Sigma^*$
  - $a(b+c)^*$
- semantics of programming language
  - while-loop: make single steps, until termination (but not more!)
- data structures
  - the natural numbers are given (“constructed”) by 0 and *succ* as constructor (and not more!)
- proof (and proof trees): a proof is given (“constructed”) from *axioms* and application of *rules* (but not more!)

The list contains some examples which can be interpreted as involving fixpoints. A common denominator of the list is: all of them correspond to *smallest* fixpoints; it always about “the smallest collection closed under something”. The  $\mu$ -calculus later allows two kinds of fixpoints, smallest ones with “ $\mu$ ” as binder and largest fixpoints (“ $\nu$ ”). It’s probably fair to say, that one is used more to smallest fixpoints constructions, generally seen: One is very much used to list as data structure, which corresponds to a smallest fix-point. In some sense (that can be made precise, but is not relevant for the lecture), a correspondent largest fix-point data structure is their “infinite” counterpart, i.e., *streams*. As said, it’s fair to say, that lists are a more common (and simpler?) data structure than streams. As a side remark, as came up during the lecture: infinite data structures, such as streams, can be seen as “lazy” data structure (as in Haskell). Lazy refers to the fact that, obviously, an infinite list (= stream) cannot be kept in memory in its infinite entirety, i.e., treated eagerly. What can be done is treating such an “infinite list” lazily, i.e., handing out the head of the last resp. the rest or tail of the stream piecewise and on demand.

Thinking about those kinds of examples (lists vs. streams, proofs as finite derivation trees vs. infinite proofs using infinite trees?...), the smallest fix-point indeed seems more common and less scary. In the  $\mu$ -calculus later, however, the two fixpoints are completely symmetric. Actually, they are duals in a technical sense (like “ $\neg\mu\neg = \nu$ ” and vice versa). What will be analogous is that smallest fixpoint formulas intuitively talk about *finite* “iterations” and largest fixpoints about *infinite* ones (we will see examples for that). Since the situation is symmetric in that sense, it’s also misleading to think that  $\mu$ -fixpoint necessarily are easier to understand compared to  $\nu$ -fixpoints. Arguably, the simplest useful temporal properties one might want to model-check are *invariants*. In LTL-notation, one can write  $\Box p$  (for “always  $p$  holds”). The proposition  $p$  must always hold, in all infinity, and that may give a hint that it corresponds to an  $\nu$ -fixpoint. The dual formula “eventually  $q$ ” (or  $\Diamond q$  in LTL) is not harder or easier to understand and corresponds dually to a smallest fixpoint: the satisfaction of  $q$  is request within a *finite* time!

## Connection to induction

Previously, the presentation stressed that “fixpoints are everywhere” (by giving examples mostly of data structures and constructions involving smallest fixpoints). As mentioned in that context: in textbooks one not often finds an *explicit* mention of the fixpoints, often other “keywords” are used (“the smallest set such that ...”). One keyword in that context is: “inductively”, like: the natural numbers are given inductively by the constant 0 and the unary function (symbol) *succ*. The use of the word “inductively” is no coincidence: inductively given structures are equipped with a proof principle called **induction**. One presumably is most used to that principle in the context of natural numbers, so much so that induction in that setting is also called *mathematical induction* or *natural induction*. The latter word just means induction over natural numbers. The rather grandiose qualification as being the principle of “mathematical” induction perhaps is meant to differentiate that from “philosophical” induction, the principle to draw conclusions from special cases or instances to a general case. By way of mentioning: the latter interpretation is also connected to what the machine learning people mostly understand when they talk about *induction*. In that kind of interpretation of induction, is often positioned as opposite to *deduction*.

Anyway, we are here only interested in the mathematical proof principle of induction which roughly says (in the case of the natural numbers):

if you need to prove some property **for all** natural numbers, then the way to go for it is:

1. prove that the property holds for 0, and prove that,
2. if it holds for  $n$ , that implies that it also holds for  $n + 1$ .

Note that there are *infinitely* many numbers, each of which is *finitely* constructed though. The two parts of that proof principle are known, of course, as the base case and the induction case or induction step, and the assumption in the induction step that the property holds for  $n$  (which is arbitrary) is also called induction hypothesis.

So far, so familiar (hopefully). Now, the slide contains one well-known and early “formalization” of natural numbers. It’s known as Peano’s axioms or postulates (or Peano-Dedekind). Depending on the source, there may be addition axioms (for instance, stating properties of the equality symbol “=”). We are interested not in that part, we are interested in the natural numbers alone. Relevant for us are in particular axioms 1. and 2. from the slide. Axiom 3. and 4. are not really too relevant for us (they have to do with the fact that the historical axioms also formalized axioms about = as part of the logic and being very explicit about that).

Anyway, the really interesting one is axiom number 5 (here listed in equation (4.1)). It’s often mentioned with this epithet (the famous fifth axiom of Peano... Whether it was in his own original writings numbered as the fifth, I have not checked). What it effectively does is another way of stating that the natural numbers (closed under 0 and successor) is *the smallest set* closed under those. It does so slightly indirectly, referring to another  $S$  is, and it stipulates that *any other set* closed in 0 and successor must be the natural numbers already. Actually, reading the sentence carefully, it states that for each set  $S$ , closed under 0 and successor

$$\mathbb{N} \subseteq S$$

not  $\mathbb{N} = S$ . Effectively, it does not make a difference (see a bit further down below), after we have discussed the formulation of Peano's axiom in the form of equation (4.2).

**remember: one way of formulation** “ $\Phi$  is **inductively** given as follows ...

Natural numbers

1. 0 is a natural number
2. if  $n$  is a natural number, then so is  $\text{succ}(n)$  (written also  $n + 1$ )
3.  $n + 1 = m + 1$ , then  $n = m$
4. there is no natural number  $n$  with  $n + 1 = 0$

**Peano Nr. 5**

If a set  $S$  contains 0 and is closed under successor, then **all** natural numbers are in  $S$ . (4.1)

**Peano Nr. 5: natural induction**

$$\forall \varphi. \varphi(0) \wedge (\forall n. \varphi(n) \rightarrow \varphi(n + 1)) \rightarrow \forall n. \varphi(n) . \quad (4.2)$$

Peano's 5th axiom is also called the **induction axiom**. That is more clearly felt in the second formulation of that principle, the one from equation (4.2). Instead of talking about subsets  $S$  of  $\mathbb{N}$ , that one speaks about a property  $\varphi$  of natural numbers. Both views are equivalent: a subset of  $\mathbb{N}$  can be interpreted as a property and vice versa. Like: the even numbers correspond to the property of “being even” and “being even” characterizes the subset of  $\mathbb{N}$  containing the even numbers.

Before wrapping up the discussion of the slide with some conclusions, let's have a closer look at the first definition of Peano's 5th postulate from equation (4.1), which is insofar informal that it's an English sentence. Apart from that it's actually unambiguous and in that sense formal. Anyway, let's be more explicit, in particular trying to shed light on the notion of *closure*.

For that, let's define the following function (on sets)

$$F(S) = \{0\} + \text{succ}(S) \quad (4.3)$$

It's meant as follows: Take set  $S$  and build  $\text{succ}(S)$  (which is defined as  $\{\text{succ}(s) \mid s \in S\}$ ) and add 0 to the resulting set (the  $+$  stands for “disjoint union”, i.e.,  $\cup$  in case there are no common elements). As an example: If  $S = \{\bullet, 6, a, 0\}$ , then  $F(S) = \{0, \text{succ}(\bullet), \text{succ}(6), \text{succ}(a), \text{succ}(0)\}$ . The 0 and  $\text{succ}$  are best thought of as constructors (i.e., “syntax”), so  $\text{succ}(6)$  is just the constructor  $\text{succ}$  applied not to the element 6. We have no reason to say  $\text{succ}(0)$  is 1, though that might be our intention (and then: what is  $\text{succ}(\bullet)$  supposed to mean, anyhow), Of course arabic numerals in a decimal positional system are a well-suited *notation* or convention to refer to natural numbers, but

that requires extra overhead, here we are concerned about “what are natural numbers” not so much “what’s an efficient way of writing them and working with them”. Now, the English formulation of (4.1) formulates (about the mentioned  $S$ ) that  $S$  contains 0 and is closed under successor. One can also see that as requiring that  $S$  is closed under the nullary constructor 0 (which is a “constant” symbol) and the unary constructor of successor. With this alternative (but equivalent) wording in mind, we can write the postulate from (4.1) a bit more formulaic as follows. A set  $S$  is *closed under  $F$* , if

$$F(S) \subseteq S \tag{4.4}$$

Note that  $S$  plays the role of a “variable” in the condition or constraint from equation (4.4), as the sub-set “equation” formulates a condition on  $S$ . Furthermore, equation (4.4) is almost an equation: it almost requires  $F(S) = S$ , but not quite. If one had this equation instead of the in-equation or subset requirement from (4.4), one has another example of a **fixpoint**:  $F(S) = S$  means that  $S$  has to be a fixpoint of  $S$ . The slight weakening for the closure formulation from equation (4.4), specifies “almost” fixpoint, but not quite. Solutions of such subset inequations in the form of (4.4) are called **pre-fixpoints**, so an  $S$ , satisfying the requirement is a pre-fixpoint of  $F$ . In other words: being a pre-fixpoint of  $F$  and being closed under  $F$  means exactly the same.

Now, as we illustrated with the “grammar example” and the “natural numbers” example: being closed is not good enough for **defining** uniquely the set; what is missing is the “nothing else” part or “the smallest such set” part. So we are after the smallest set  $S$  satisfying the closure condition of (4.4), resp. *the smallest pre-fixpoint of  $F$* . It’s obvious that every fixpoint is also a pre-fixpoint. The convers typically does not hold, there are strictly more pre-fixpoints than fixpoints (the  $\subseteq$  condition is just weaker than requiring  $=$ ). That’s easy enough, and not too interesting. However, we are actually not interested in fixpoints or pre-fixpoints as such. We are interested in the *smallest* such things. Then, what *is* interesting and relevant indeed is the following: under standard circumstances there is **no difference** between **smallest fixpoints** and **smallest pre-fixpoints** for a given  $F$ . In fact, not only that, but there is exactly *one* unique smallest fix-point which at the same is the *one* unique smallest pre-fixpoint. This fact, which is, with a bit of background, straightforward to establish also justifies that one speaks (under standard circumstances) of **the** smallest or **the** least fixpoint (as opposed to some or other “miminal” fixpoint).

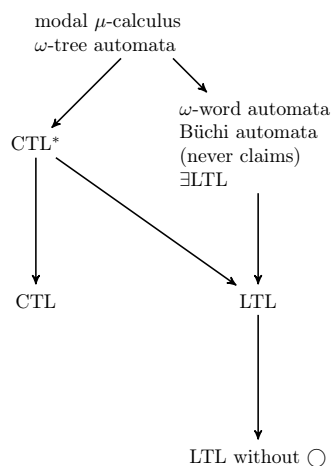
Now, as argued, the two formulations of Peano’s 5th axiom, the ones from equations (4.1) and (4.2) are equivalent and also correspond to a (smallest) fixpoint requirement and, equivalently, allow the proof principle of mathematical induction. Looking especially at the formulation (4.2) where it’s more visible: the formula is **not** a formula of first-order logic: there is quantification over “all propositions”  $\varphi$  in (4.2). Whatever logic it is, it’s **NOT** first-order. One actually should tread rather careful here, it’s slippery ground. The natural impulse could be: Oh, well, quantifying over propositions, I know already, that’s *second-order logic*. Now, what may act as a warning sign here pondering the question of whether the above formula corresponds to a proposition itself. If so, one can have a logic where the quantifiers quantify over propositions *including the quantified formula itself* (the technical term in that connection is *impredicativity*). That kind of self-referential quantification may or may not lead to a catastrophe (catastrophe in the sense that the

whole logic collapses and everything can be proven in that logic, and that mean really everything, including the negation of everything). In other words, the logic could become *inconsistent*. That form if self-referentiality is connected to the concept of whether there is a “set of all sets”. If allows such things, one can introduce contradictions into set-theory (like the set of all sets that don’t contain themselves).

Let’s leave it at that. The importance for us is: Peanos induction axiom corresponds to a fix-point operator, and the discussion about the dangers of internalizing induction resp fixpoint into some logic should be an indication that this yields are some fairly powerful mechanism and it supports the status of the  $\mu$ -calculus as an important logic.

A final remark: the discussion here centered on “natural induction”, i.e., over  $\mathbb{N}$ , but completely analogous principles hold for other inductively given structures. In that case, one also speaks of *structural induction* (induction over the structure) and  $\mathbb{N}$  is just a special case of a particularly simple structure.

## Expressivity



The picture (shown earlier as well) shows again the order of different logics that are relevant in connection with model checking (and some automata model) with the  $\mu$ -calculus sitting on top. There is also a corresponding automaton model which we will not cover in the lecture.

## 4.2 Propositional $\mu$ -calculus: syntax and semantics

### 4.2.1 Syntax

#### Labelled transition systems

A *transition system* is a tuple

$$(S, \rightarrow, \{P_i\}_{i \in \mathbb{N}})$$



- $AP = \{p_0, p_1, p_2, \dots\}$
- $Act: (= \Sigma)$  actions  $a, b', \dots$
- $\rightarrow \subseteq S \times Act \times S$ 
  - $s \xrightarrow{a} s'$ ,  $a$ -transition, from  $s$  to  $s'$
- $P_i \subseteq S$
- note: switch of perspective for “proposition labelling”

Basically, it’s a recap of earlier notions. The notion of (labelled) transition system here corresponds to that what we had before called Kripke model, except that now also the transitions are labelled by letters from an alphabet. Basically we encountered them in connection with multi-modal logics already. That’s not surprising, insofar the flavor of the  $\mu$ -calculus we present here is basically adding fixpoints to a multi-modal logic. In [7], they call the “models” just *transition systems* (not labelled transition systems), also what we introduced as alphabet  $\Sigma$ , a finite set of “symbols” or letters, they call that *action set*, the element of which are consequently called *actions*. Again, that’s just terminology.

### Syntax

$\varphi ::=$	$p \mid \neg p$	props and their negation
	$X$	variables
	$\varphi \wedge \varphi \mid \varphi \vee \varphi$	2 usual boolean connectives
	$[a]\varphi \mid \langle a \rangle \varphi$	(multi)-modal operators
	$\mu X.\varphi \mid \nu X.\varphi$	fix-points

- *true* and *false*:  $p \wedge \neg p$  resp.  $p \vee \neg p$
- variables  $X, Y \dots \in Var$
- actions  $a, b' \dots \in Act$

### Remarks on the syntax

- general negation: missing
  - especially  $\neg X$  not part of the syntax
  - but:  $\neg\varphi$  *definable*
- $\mu$  and  $\nu$  (or  $\sigma$  when unspecific): **binding** operators
  - free and bound occurrences of variables
  - renaming of bound variables ( $\alpha$ -renaming)
- some well-formedness conditions
  - don’t reuse variables
  - guardedness

We don't go into details about those well-formedness conditions (they are important for some technical developments, only). But the concept of free and bound occurrences of variables should be familiar. Guardedness, just to give an impression, wants to avoid "immediate recursions" like  $\mu X.X$  or also  $\nu X.X \vee p$  (these are examples where  $X$  occurs unguarded). Those formulas are not contradictory, they have a meaning, but one tries to keep them out as they make some troubles in establishing results. Fortunately, some have proved that one can ignore them (insofar that one can always transform a formula into an equivalent one that avoids such situations; that particular transformation is not obvious, though...). Anyway, whereas  $\mu X.X$  is not meaningless per se (just not nice to handle in establishing some results), formulas like  $\mu X.\neg X$  are **meaningless** in that it cannot be given an interpretation in a manner that "makes sense". The cause of the problem is, that the body of the fixpoint construction, the formula  $\neg X$  in that case, does not represent a *monotone* function. By **monotone** we understand *monotonously increasing*.

During the lecture there had been comments that in some fields, the terminology of monotone functions capture both *monotonously increasing* and *monotonously decreasing* functions. For us, monotone refers to the increasing case (or at least not-decreasing), only. The "decreasing" situation, we would call *antitone*. Now, fixpoints are guaranteed to exist only in case the involved function is monotone, and that's the same for  $\mu$  and for  $\nu$  (it's not like: for  $\mu$ , the function need to be monotone, for  $\nu$ , on the other hand, antitone). Non-monotonic functions are out of the picture for us. That's why the syntax does not allow free-form negation, as there would be a danger to write formulas that act non-monotone. The syntax as given results in monotone functions, only. The (small) price to pay is that the syntax becomes slightly less compact than it would be otherwise: for each constructor, one also has to include its dual.

### What about the variables?

- propositional  $\mu$ -calculus
  - $X, Y \dots$  *different* from first-order logic variables
  - variables here represent
    - formulas (from  $\mu$ -calculus), resp.
    - semantically: *sets of states*
- ⇒ **second-order** flavor!

By "second-order" flavor we mean that we have variables that represent formulas (resp. sets of states captured by formulas). And we also (in some way) "quantify" over them, only not by existential or universal quantification  $\exists$  and  $\forall$ , but by the fix-point binders  $\mu$  and  $\nu$ . At any rate, the variables  $X, Y' \dots$  are of quite a different nature from a setting if we had not the propositional  $\mu$ -calculus, but had a first-order logic setting as underlying logic (with term variables  $x, y'$ ). We see that also in the notion of *valuation* (when discussing the semantics). A valuation (or variable assignment) maps values or "semantics" to variables. Now, valuations will map variables from *Var* to set of states.

### Preview on the semantics

- given transition system  $\mathcal{M}$

### Satisfaction relation

$$s \models_{\mathcal{M}} \varphi$$

$$s \models_{\mathcal{M}}^{\mathcal{V}} \varphi \quad (4.5)$$

**Equivalently: semantic function** semantics of  $\varphi$  in transition system  $\mathcal{M}$  and with valuation  $\mathcal{V}$ :

$$\llbracket \varphi \rrbracket_{\mathcal{V}}^{\mathcal{M}} \in 2^S \quad (4.6)$$

$$\mathcal{V} : Var \rightarrow 2^S$$

There is not much “new” on the slides, compared to what we had in earlier chapters. Instead of the satisfaction relation  $\models$ , we define the semantics as “semantic function”, traditionally written like  $\llbracket \_ \rrbracket$ . That’s just more convenient when later talking about fix-points. The fixpoints will be built over functions (as usual), and therefore, writing down the semantics using the “semantical function” formulation comes in more handy.

Formulas are interpreted over a given transition system  $\mathcal{M}$ , i.e., the semantics of a formula  $\varphi$  is a set of states in that given  $\mathcal{M}$ . Since formulas may contain variables, the interpretation is additionally relative to choosing values for those free variables. So we assume a *valuation*  $\mathcal{V}$  for those, with  $\mathcal{V} : Var \rightarrow 2^S$ , which assigns “semantics” (= sets of states) to each variable. That’s why the semantic function from (4.6) or the satisfaction relation from (4.5) mentions  $\mathcal{V}$  (besides  $\mathcal{M}$ ).

### Semantics (no fix-points)

The semantics for the logic without fixpoints is rather straightforward. That fragment is sometimes also called Hennessy-Milner logic (without recursion). It’s basically the multi-modal logic we have encountered earlier (in preparation of dynamic logic.)

$$\varphi ::= p_i \mid \neg p_i \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [a]\varphi \mid \langle a \rangle \varphi$$

$$\begin{aligned} \llbracket true \rrbracket_{\mathcal{V}}^{\mathcal{M}} &= S & \llbracket false \rrbracket_{\mathcal{V}}^{\mathcal{M}} &= \emptyset \\ \llbracket p_i \rrbracket_{\mathcal{V}}^{\mathcal{M}} &= P_i & \llbracket \neg p_i \rrbracket_{\mathcal{V}}^{\mathcal{M}} &= S - P_i \\ \llbracket \varphi_1 \wedge \varphi_2 \rrbracket_{\mathcal{V}}^{\mathcal{M}} &= \llbracket \varphi_1 \rrbracket_{\mathcal{V}}^{\mathcal{M}} \cap \llbracket \varphi_2 \rrbracket_{\mathcal{V}}^{\mathcal{M}} & \llbracket \varphi_1 \vee \varphi_2 \rrbracket_{\mathcal{V}}^{\mathcal{M}} &= \llbracket \varphi_1 \rrbracket_{\mathcal{V}}^{\mathcal{M}} \cup \llbracket \varphi_2 \rrbracket_{\mathcal{V}}^{\mathcal{M}} \\ \llbracket [a]\varphi \rrbracket_{\mathcal{V}}^{\mathcal{M}} &= \{s \in S \mid \forall s'. s \xrightarrow{a} s' \Rightarrow s' \in \llbracket \varphi \rrbracket_{\mathcal{V}}^{\mathcal{M}}\} \\ \llbracket \langle a \rangle \varphi \rrbracket_{\mathcal{V}}^{\mathcal{M}} &= \{s \in S \mid \exists s'. s \xrightarrow{a} s' \wedge s' \in \llbracket \varphi \rrbracket_{\mathcal{V}}^{\mathcal{M}}\} \end{aligned}$$

## Fixpoints in LTL

- $\Box p?$
- $\Diamond p$
- $p U q$

Earlier, we made a big story around the fact that there are “fixpoints everywhere”. Examples were mostly drawn from some constructions in “math” but not as integral part of a logic. Of course, the  $\mu$ -calculus is kind of a the prototypical logic which offers fixpoints inside the logic, not just externally on the meta-level. But what about the other temporal logic we covered, LTL? On the next slide, we have a look at, for example, the “always” operator,  $\Box$ .

### Reconsider for instance $\Box p?$

- fix-point equation for “always”?

$$\Box p = p \wedge \bigcirc \Box p. \quad (4.7)$$

**choose**  $\Box p = false$

$$false = p \wedge \bigcirc false \quad (4.8)$$

Let’s take  $\Box$  for illustration. We are currently using LTL, which is a linear logic. The  $\mu$ -calculus in its general form a branching logic, but the arguments or illustrations work equally for a linear logic. Actually, one can also interpret the  $\mu$ -calculus over linear transitions systems, only. Anyway, “always  $p$ ” can be intuitively explained as

“always  $p$ ” means  $p$  now and, after one step “always  $p$ ”.

That’s of course a recursive “definition” in that the thing one tries to define, “always  $p$ ” is define in terms of itself (we should thought be careful with the word definition, as there may be more than one solution to that equation; and indeed there are). It also corresponds to the *unrolling* feature of  $\Box p$ , as given in equation (4.7). Seeing it this way means, we have been given  $\Box p$  already. The chapter about LTL gave a definition of the semantics of  $\Box p$ ; maybe it was given indirectly, in that we focused on  $U$  alone, but anyway, it was somehow defined, and the definition would state that a path satisfies  $\Box p$ , if  $\forall i$ , the suffix of the path starting at  $i$  satisfies  $p$ . So far, so familiar. Point being, equation (4.7) states a fact about  $\Box p$  whose definition is given already.

However, we can try to use the equation also as **defining**  $\Box p$ . In that case we should probably write better

$$”\Box p” = p \wedge \bigcirc ”\Box p” ,$$

to make clear that ” $\Box p$ ” is something that we want to define and which we intend to satisfy the stated (recursive) equation. Since we want to “solve” that equation (to define  $\Box p$ ), it’s probably even better to write

$$X = p \wedge \bigcirc X \quad (4.9)$$

i.e., to use a variable  $X$  (as opposed to " $\Box p$ ") to represent the (yet to define) thing for which we want the equation to hold. That, of course, can be seen as a fix-point equation (in the discussion here, we dispense with the distinction of fixpoints (using  $=$ ) and pre-fixpoints (using  $\subseteq$  or implication).

Now, we can try to solve it. We have heard about the importance of smallest fix-points extensively. The smallest imaginable solution (in terms of sets of states) would be the empty set. That corresponds alternatively to the proposition *false*. So, let's fill in *false* for the unknown  $X$  (or interpreting semantically  $X$  as  $\emptyset$ ), checking if  $\text{false} \stackrel{?}{=} p \wedge \bigcirc \text{false}$  (equation (4.8) from the slide). Well,  $\bigcirc \text{false}$  is equal to *false*, which means *false* is indeed a solution to the fixpoint equation (4.9). And, being the empty set, it's definitely the *smallest* fixpoint.

That's a disappointment, since we set out to give a fixpoint definition of  $\Box p$ , not of *false*. Now, since the smallest fixpoint was a failure, the only hope is the largest fixpoint (nobody needs fixpoints somewhere in the middle...) As it turns out, the largest fixpoint is what we are after, it corresponds to  $\Box p$ . So we can define

$$\Box p = \nu X. p \wedge \bigcirc X . \quad (4.10)$$

That's all fine and good, but one may feel uneasy, for instance by the obvious "failure" of  $\mu X. p \wedge \bigcirc X$ . Maybe in this case it's clear that it can't be the smallest fixpoint (as it's *false*), so it must in all plausibility be the largest. But maybe there are situations where the smallest fixpoint does not collapse to *false* but makes some form of sense, it's only not the one we planned for. How do we know that the definition from equation (4.10) is what we want? At least it *is* a definition, because we know that in the given circumstances, the greatest fixpoint is unique (as is the smallest fixpoint which turned out to be *false*). We could leave it like that: we defined  $\Box p$  by (4.10) and that just it. However, we already had an independent definition of "always  $p$ " not involving  $\mu$  in an apparent way, and beside that we have an intuition what "always  $p$ " is supposed to mean. How can we make sure that our intention is captured by the definition from (4.10)? Or in general: what the intuition, when should one choose  $\mu$  and when  $\nu$ ?

Now, it's tricky to communicate "intuition", intuition builds up by doing things and exercising and repeating things. After a while things seem "natural" and intuitively clear, and one forgets that one had been struggling with the concept in the beginning. That's also when it becomes hard to communicate the idea, because stating "it's intuitively clear that ..." is not helpful. But many things in math are like that and for many things we have forgotten that they may not seem clear at the beginning. Notions about "infinity" are notoriously "unclear" (and philosophically debated, like in which sense does the "infinite" exist, etc). In some sense we get used to it:  $\mathbb{N} = \{0, \text{succ}(0), \text{succ}(\text{succ}(0))\dots\}$  (or more commonly  $\{0, 1, 2, \dots\}$ , and no one loses sleep about the "...", we all know that there "are" infinitely many natural numbers (maybe on a computer only up to MAXINT, but, well, in principle, you know...)). We are just used to deal conceptually with infinite sets without breaking a sweat (like natural numbers, the set of propositional formulas as give

by a grammar, the set  $\Sigma^*$  and those examples from before). It may be noted that even other mathematical facts needed time “to get used to” (historically). One is 0. There had been a time where people had scruples to “have” or work with 0, like: “how can there be or should there be there be a symbol for the absense of something”. Zero stands for “nothing” and one cannot meaningfully compute with nothing (there was a time also where the people had the concept of 0 but did not use a symbol for it, like leaving the place empty. That did not help readability much. And it took time to “get used to it” and have the courage to write 0 for something that “actually” (?) does not exists. Same with the imaginary numbers (numbers like  $\sqrt{-1}$  that do not really exist, or do they?)

The examples about infinite sets ( $\mathbb{N}$ ,  $\Sigma^*$  ...) correspond perhaps not coincidentally to *smallest* fixpoints. It's about *infinite sets* build up by using finite constructions for each elements.  $a^* = \{\epsilon, a, aa, aaa, \dots\} = \{a^n \mid n \in \mathbb{N}\}$  which we understand as the set containing words using  $a$ , where the words are of **arbitrary length**  $n$ . Implicit in the word “arbitrary length” is that it not really “arbitrary” in the sense “ $n =$  infinite length”. Not arbitrary in this radical sense, just ordinary arbitrariness please, like  $n \in \mathbb{N}$ .  $\mathbb{N}$  is infinite, that's fine, but  $n$  please not (because infinite numbers don't really exist, or do they?..).

This discussion is meant less historical, but more of giving intuition about  $\mu$  and  $\nu$ . The smallest fixpoints are of the form as in  $a^n$  with  $n \in \mathbb{N}$ . They may describe a infinite collection of entities (like words or numbers) each of which is finite.  $\nu$ -based definition corresponds to constructions where the single elements are conceptually *infinite* (or at least include infinite elements). One example we have actually seen was  $\Sigma^\omega$ , the set of infinite words over  $\Sigma$ .

## 4.2.2 Background: Fixpoints

### Orders, lattices, etc.

as a reminder:

- partial order  $(L, \sqsubseteq)$
- *upper bound*  $l$  of  $Y \subseteq L$ :
- *least upper bound* (lub):  $\bigsqcup Y$  (or *join*)
- dually: lower bounds and greatest lower bounds:  $\bigsqcap Y$  (or *meet*)
- **complete lattice**  $L = (L, \sqsubseteq) = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \perp, \top)$ : a partially ordered set where meets and joins exist for *all subsets*, furthermore  $\top = \bigsqcup \emptyset$  and  $\perp = \bigsqcap \emptyset$ .

There are also other forms of lattices, for instance, if one only needs joins, but not meets, one can get away with a semi-lattice, and there are many more variations. For the lecture, we generally simply assume *complete lattices* and thus, the montone framework is happy. In particular, if we are dealing with *finite* lattices, which is an important case, we don't need to consider *infinite* sets, and “standard” lattices with binary meets and joins (and least and largest elements) are complete already.

## Fixpoints

given complete lattice  $L$  and monotone  $f : L \rightarrow L$ .

- **fixpoint:**  $f(l) = l$

$$Fix(f) = \{l \mid f(l) = l\}$$

- $f$  *reductive* at  $l$ ,  $l$  is a **pre-fixpoint** of  $f$ :  $f(l) \sqsubseteq l$ :

$$Red(f) = \{l \mid f(l) \sqsubseteq l\}$$

- $f$  *extensive* at  $l$ ,  $l$  is a **post-fixpoint** of  $f$ :  $f(l) \sqsupseteq l$ :

$$Ext(f) = \{l \mid f(l) \sqsupseteq l\}$$

### Define “lfp” / “gfp”

$$lfp(f) \triangleq \bigsqcap Fix(f) \quad \text{and} \quad gfp(f) \triangleq \bigsqcup Fix(f)$$

The last display just gives the names to the two elements of the lattice defined by the corresponding right-hand sides. We know that those elements are existing thanks to the fact that  $L$  is a complete lattice (and it’s very easy to see that meets and joins are unique, that means the  $lfp(f)$  and  $gfp(f)$  are well-defined elements of the lattice). The chosen names somehow suggest that the two thusly defined elements are the least fixpoint, resp. the greatest fixpoint of the monotone function  $f$ .

But, so far  $lfp$  and  $gfp$  is just a suggestive choice of name. It requires a separate *argument* that the elements are actually fixpoints, and the least, resp. the largest fixpoint as that as well. Finally, if we take it really serious, an argument should be found that allows to speaking of *the* least fixpoint. If there is more than one least fixpoint, one should avoid talking about “the least fixpoint” (same for the largest fixpoint). The argument for uniqueness of least fixpoints (or for greatest fixpoint) is very simple though, similar to arguing for the uniqueness of “the least upper bound” etc.

If one would carry out the argument, i.e., the proof, that all fits together in the sense that the  $lfp(f)$  and  $gfp(f)$  defined above *are actually* the least fixpoint and the largest fixpoint, and if one would carefully keep track of what is actually needed to make the proof go through step by step, then one would see that *every single condition* for being a complete lattice is needed (plus the fact that  $f$  is monotone). If one removes one condition, the argument fails! Conversely that means the following: We are interested in uniquely “best approximations” (least or greatest fixpoints depending in whether it’s a may or a must analysis),

and, having a monotone  $f$ , a complete lattices **is exactly what guarantees that those fixpoints exists**. Exactly that, nothing less and nothing more, If your framework has monotone functions and is based on a complete lattice, it works. If not, it does not work, very simple.

That explains the importance of lattices and monotone function. Also, I would guess that historically, the *need* to assure existence of fixpoints has led Tarski (the mathematician whose concepts we are currently covering) exactly to the definition of lattice, not the other way around (“oh, someone defined some lattice, let’s see what I can find out about them, perhaps I could define some  $lfp(f)$  like above and see if I could prove something interesting about it, perhaps it’s a fixpoint?”. But as said, that is speculation.

Having stressed the importance of complete lattices, for fairness sake it should be said that there’s also a place for analyses which fail to meet those conditions. In that case, one might not have a (unique) best solution. Perhaps even worse (and related to that), one might need *combinatorial* techniques (like backtracking), i.e., checking all possible solutions to find an acceptable one. If that happens, the *cost* of the analysis may explode. To avoid that one may give up to look for a “best solution” and settle for a “good enough” one and heuristics that hopefully find an acceptable one efficiently, or even throw the towel and give up “soundness”. Anyway and fortunately, plenty of important analyses fit well into the monotone framework with its lattices, its unique best solution and —perhaps best of all— its efficient solving techniques. Therefore this lecture will cover only those here. Those are called classical *data flow analyses*.

### Tarski’s theorem

**Core** Perhaps core insight of the whole lattice/fixpoint business: not only does the  $\sqcap$  of all pre-fixpoints uniquely exist (that’s what the lattice is for), but —and that’s the trick— *it’s a pre-fixpoint itself* (ultimately due to monotonicity of  $f$ ).

**Theorem 4.2.1.**  $L$ : complete lattice,  $f : L \rightarrow L$  monotone.

$$\begin{aligned} lfp(f) &\triangleq \sqcap Red(f) \in Fix(f) \\ gfp(f) &\triangleq \sqcup Ext(f) \in Fix(f) \end{aligned} \quad (4.11)$$

- Note:  $lfp$  (despite the name) is *defined* as glb of all pre-fixpoints
- The theorem (more or less directly) implies  $lfp$  is the *least* fixpoint

### Fixpoint iteration

- often: iterate, approximate least fixed point from below  $(f^n(\perp))_n$ :

$$\perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq \dots$$

- not assured that we “reach” the fixpoint (“within”  $\omega$ )

$$\begin{aligned} \perp \sqsubseteq f^n(\perp) \sqsubseteq \sqcup_n f^n(\perp) &\sqsubseteq lfp(f) \\ gfp(f) \sqsubseteq \sqcap_n f^n(\top) \sqsubseteq f^n(\top) &\sqsubseteq \top \end{aligned}$$

- additional requirement: **continuity** on  $f$  for all **ascending chains**  $(l_n)_n$

$$f\left(\bigsqcup_n l_n\right) = \bigsqcup_n f(l_n)$$



### 4.2.3 Semantics

#### Semantics of formulas with free variables

- apply the FP theorem (Knaster-Tarski)
- assume  $\mu X.\varphi(X)$  or  $\nu X.\varphi(X)$
- $\mathcal{M}$  with state set  $S$ : fixed
- consider semantics of body  $\varphi(X)$ ,
  - assume (for simplicity), only one free variable  $X$

$$\llbracket \varphi(X) \rrbracket^{\mathcal{M}} : 2^S \rightarrow 2^S$$

- 2 welcome facts
  1.  $2^S$  a **complete lattice**
  2. the function is **monotone** (and also **continuous**, under reasonable assumptions)
- general case:  $\varphi$  may have more variables than just  $X$

$$f(S') = \llbracket \varphi \rrbracket_{\mathcal{V}[X \mapsto S']}^{\mathcal{M}} : 2^{S'} \rightarrow 2^{S'}$$

with  $S' \subseteq S$

#### Semantics of the fixpoints

$$\begin{aligned} \llbracket \mu X.\varphi \rrbracket_{\mathcal{V}}^{\mathcal{M}} &= \bigsqcap \{S' \subseteq S \mid \llbracket \varphi \rrbracket_{\mathcal{V}[X \mapsto S']}^{\mathcal{M}} \subseteq S'\} && \text{(lfp)} \\ &= \bigsqcap \{S' \subseteq S \mid f(S') \subseteq S'\} \end{aligned}$$

$$\begin{aligned} \llbracket \nu X.\varphi \rrbracket_{\mathcal{V}}^{\mathcal{M}} &= \bigsqcup \{S' \subseteq S \mid S' \subseteq \llbracket \varphi \rrbracket_{\mathcal{V}[X \mapsto S']}^{\mathcal{M}}\} && \text{(gfp)} \\ &= \bigsqcup \{S' \subseteq S \mid S' \subseteq f(S')\} \end{aligned}$$

where  $f(S') = \llbracket \varphi \rrbracket_{\mathcal{V}[X \mapsto S']}^{\mathcal{M}}$

#### Alternation of fixpoints

- expressivity of  $\mu$ -calculus: due to fix-points
- more precisely: “nesting” of fix.points
- even more precisely: **alternation-depth of nested fixpoints.**
- compare: direct recursion vs. mutual recursion
- similarly: “ $\mu^2 = \mu$ ”
- technical definition of nesting: not 100% immediate
  1.  $(\mu X.\varphi) \wedge (\nu X.\varphi_2)$ : no nesting
  2.  $\mu X.\mu Y \varphi(X, Y)$ : no alternation
  3.  $\nu X.((\mu Y.p \vee \langle b \rangle Y) \wedge [a]X)$ : ??
  4. ???

The definition is not 100% obvious. In particular, the formula number 3 is **not** nested in the way covered by the definition. Naively it is: the  $\mu$  fixpoint is inside the  $\nu$  fixpoint, so there seems to be nesting. Also there is alternation. But: as said, the definition is slightly more complex. The situation in formula 3 does not count as proper nesting, in that the two “loops” are somehow “independent”. The inner loop is *completely enclosed* in the outer loop. Why that may be called a form of nesting, but it’s not the “proper” nesting that gives the  $\mu$ -calculus its power.

## 4.3 Model checking

### Game

**Definition 4.3.1** (Game). A game is a triple  $G = (V, T, Acc)$  where

1.  $V$  are *nodes* partitioned between two players, Adam and Eve:  $V = V_A + V_E$ .
2.  $T \subseteq V \times V$  is a *transition relation* determining the possible successors of each node, and
3.  $Acc \subseteq V^\omega$  is a set defining the *winning condition*
  - node: aka **position**
  - $Acc$ : *winning condition*

### Playing a game like this

- two-player game
- two kinds of nodes (Eve’s and Adam’s)
- game “moves” through positions
  - in one of Eve’s nodes: **Eve chooses**
  - analogous for Adam
- **winning**:
  - winning condition: from the perspective of Eve
    - \* infinite path through  $G$ : if  $Acc$  satisfied, Eve wins, otherwise Adam
  - a player “stuck”: loses as well
  - no draw possible
- winning *node*:  $\exists$  winning strategy

**strategy  $\theta$  (for Eve)** Given  $G$ . For each sequence of nodes  $\vec{v}$ , ending in a node  $v \in V_E$ : choose  $\theta(\vec{v}) = v'$ , such that  $v \rightarrow v'$

### Games as general framework

- “game theory”: broad field with many applications
- here: game used for
  - semantics, logics, model-checking
- situation often: open systems

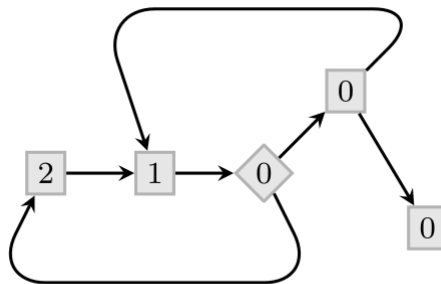
- environment context  $\leftrightarrow$  program
- attacker  $\leftrightarrow$  system
- controllable  $\leftrightarrow$  non-controllable parts

**Game** Different, players with own “goals” (conflicting or at least different) and local “influence” or control.

### Rest

- many variations
  - 2-player, multi-player
  - zero-sum games (no win/win situation in those...)
  - restricted information
  - probabilistic (“mixed”) strategies (Nash-equilibrium!)
  - ...

### Game example



- nodes or positions in the graph
  - Eve: “diamond”-shaped
  - Adam: “box”-shaped
- winning condition (here): Eve wins, if the game passes through “2” infinitely many times
- numbers in the nodes: “ranks” (see later)

Eve (in this example) has only one node, where she can choose going up or down. If she chooses uniformly each time to go down, she wins, as she passes infinitely often through the node marked with 2. This condition is, in this example, the winning condition *Acc* (which is formulated from the perspective of Eve). If she goes up, Adam then get’s a choice. He could actually prevent Eve from ever reaching 2 again, but in doing so, and going down, he will be stuck, and thereby loose immediately (getting stuck means loosing). Therefore, he is forced to take the loop back.

Note: even if Eve cannot control or know what Adam does, she can rely on the fact that he acts “rational” in the sense that he wants to make moves with the goal of winning (or not loosing)

BTW: not loosing is actually the same thing as winning since in the games we are looking at, the outcome is binary or “boolean”: either Adam wins and Eve loses, or the other

way around. We use a game formulation to determine if a property holds in a state or not, and that is either the case or not, as in classical logic.

### Positional strategies

- strategy in general  $\theta(\vec{v}) = v'$
- in the example: strategy of Eve: can be dependent on the “current node” only  
 $\Rightarrow$  **memoryless** or **positional**

$$\theta(v) = v'$$

### Parity games

given game  $G$

**Parity winning condition** ranking:  $\Omega : V \rightarrow \{0, 1, \dots, d\}$

$$Acc = \{\vec{v} \in V^\omega \mid \limsup_{i \rightarrow \infty} \Omega(v_i) \text{ is even}\}$$

Mostowski [26], Emerson and Jutla [13]

### PWC theorem

- every position is winning for one of the two players
- it's winnable by a **positional** strategy
- it's **decidable** who wins

### Model checking $\mu$ -calculus and parity games

#### Verification game

$$\mathcal{M}, s \models \varphi \quad \Rightarrow \quad \mathcal{G}(\mathcal{M}, \varphi)$$

Eve wins from position cor-  
 $\mathcal{M}, s \models \varphi$  iff responding to  $s$  and  $\varphi$  in  
 $\mathcal{G}(\mathcal{M}, \varphi)$

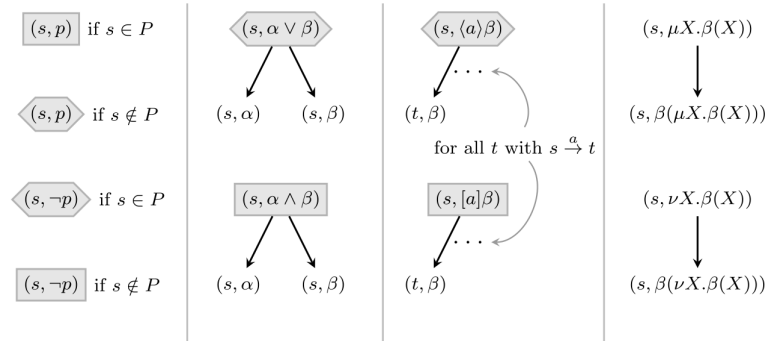
- $\mathcal{V}$ : valuation for free vars, i.e., game  $\mathcal{G}_{\mathcal{V}}(\mathcal{M}, \varphi)$
- **positions** in the game

$$(s, \psi)$$

$\psi$ : 1 formula from the closure of  $\varphi$

**Intention of the construction** Eve has winning strategy from  $(s, \psi)$  iff  $\mathcal{M}, \mathcal{V}, s \models \psi$

### Rules of the verification game



### Ranking

- ranking positions with fp- formulas  $\nu.\psi$  or  $\mu.\psi$ 
  1.  $\mu$ -formula odd.
  2.  $\nu$ -formula even

### Rest

- additionally: subformulas should must no higher ranks than the surrounding formula
- standard way: connect to **alternation depth**

### Rest

$$\begin{aligned} \Gamma(\psi) &= 2 \lfloor \text{adepth}(X)/2 \rfloor && \text{for } \psi = \nu X. \psi'(X) \\ \Gamma(\psi) &= 2 \lfloor \text{adepth}(X)/2 \rfloor + 1 && \text{for } \psi = \mu X. \psi'(X) \\ \Gamma(\psi) &= 0 && \text{otherwise} \end{aligned}$$

# Chapter 5

## 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

5.1	Introduction . . . . .	138
5.2	Independence and invisibility	146
5.3	POR for $LTL_{\circlearrowleft}$ . . . . .	148

## 5.1 Introduction

The material here is based on chapter 10 from the book [11] or the handbook article [28].

### State space explosion problem

- model checking in general “intractable”
- fundamental limitation: combinatorial explosion
- state space: exponential in problem size
  - in particular in *number of processes*

All analyses based on “exploration” or “searching” suffer from the fact that problems become unmanageable when confronted with realistic problems. That’s also true for other approaches like SAT/SMT (but 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  $\varphi$ ”), 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  $\varphi$ . 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, given a program there may be other sources that make a raw state exploration unmanageable. If the problem depends in 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. It's 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's sometimes 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 covered in this section is a technique, specially made to work well to reduce 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  $\mu$ -calculus it was hinted at that the alternation-depth is connected to the complexity of the model checking problem in that context. 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 conjunction, one can check the individual properties one by one.

### Battling the state space explosion

Since it's such a major road block, it's clear that many different techniques have been proposed, investigated, and implemented to address it. The list includes some major ones, of course also clever implementation and data representations and other programming-related techniques are used.

- symbolic techniques
- BDDs
- abstraction
- compositional approaches
- symmetry reduction
- special data representations
- "compiler optimizations": slicing, live variable analysis ...
- parallelization
- here: *partial order reduction*

### "Asynchronous" systems and interleaving

- remember: synchronous and asynchronous product (in connection with LTL model checking)
- asynchronous: software and asynchronous HW
- synchronous: often HW, global clock
- **interleaving** (of steps, actions, transitions ...)

Partial-order reduction is most effective in asynchronous systems. The distinction is for systems with different parts working in parallel or concurrently, and one can make that distinction for HW or software. In HW, synchronous behavior can be achieved by a global HW clock, that forces different components to work in lock step. The global clock is used to *synchronize* the 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 “asynchronous” 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-)acceptance 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?

- partial-order semantics
- what is *concurrent* (or parallel) execution?
- “causal” order
- “*true*” concurrency vs. *interleaving* semantics
- “math” fact: PO equivalent set of all linearizations
- “reality” fact: POR only “approximates” that math-fact
- perhaps better name for POR: “**COR**”:

### commutativity-based reduction

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* (we encountered it in connection with lattices when dealing with the  $\mu$ -calculus).

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 different 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 concurrent 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-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 semantics (let alone 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 described 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 for from easy to start upfront with a partial-order based semantics for all programs.

POR reduction 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: it's intuitively clear that, 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", i.e., running the model checker, and compromises need to be done 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, or something. 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 leave it out." Of course, the criterion should not be trival like: I leave out an edges 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 concrete details when to explore and edge and when not depend also on the *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 it's not worth in checking that, one may conservatively explore both orderings anyway, perhaps it's not worth the effort.

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 [28] who makes that argument).

**Exploiting “equivalences”** Instead of checking all “situations”,

- figure which are **equivalent** (also wrt. to the property)
- check only one (or at least not all) **representatives** per equivalence class
- see also *symmetry reduction*
- 8 queens problem
- POR: equivalent *behaviors*

### (Labelled) transition systems

- basically unchanged,
  - assume initial states
  - states labelled with sets  $2^{AP}$
  - state-labelling function  $L$
  - transitions are as well
- alternatively multiple transition relations: instead of  $\xrightarrow{\alpha}$ , we also see  $\alpha$  as relation

$$(S, S_0, \rightarrow, L)$$

### Determinism and enabledness

- remember:  $\xrightarrow{\alpha}$  **deterministic**
- in that case: also write  $s' = \alpha(s)$  for  $s \xrightarrow{\alpha} s'$  (or  $\alpha(s, s')$ )

**Enabledness**  $\xrightarrow{\alpha}$  **enabled** in  $s$ , if  $s \xrightarrow{\alpha}$

Otherwise  $\xrightarrow{\alpha}$  *disabled* in  $s$ .

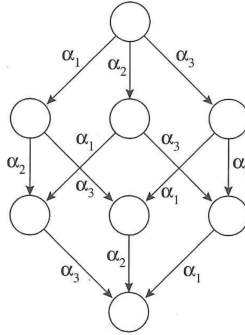
- path  $\pi$ :

$$s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \dots$$

- not necessarily infinite

## Concurrency in asynchronous systems

- independent transitions
- arbitrary orderings or linearizations (= interleavings)
- [actions themselves assumed atomic / indivisible]



- raw math calculation:  $n$  transition relations
  - $n!$  different orderings
  - $2^n$  states

## Reducing the state space

- goal: **pruning** the state space

### Super-unrealistic:

1. generate explicitly the state space by DFS
2. then prune it (remove equivalent transitions & states)
3. then model check the property

### unrealistic (but for presentation reasons)

1. generate explicitly the reduced state space (using modified DFS)
2. then model check the property

### Modified DFS: ample set

- standard DFS: basically *recursion* (probably with explicit stack)
- exploration: explore “successor states”, i.e.,

follow **all enabled** transitions

- graph exploration (not tree): check for *revisits*

**Modification/improvement** Don't explore *all* enabled transitions.

follow **enough enabled** transition

- ample: think “sufficient” or “enough”
- **ample** set of transitions in a state  $\subseteq$  set of enabled transitions in a state

## Modified DFS

```

1  hash(s0);
2  set on_stack(s0);
3  expand_state(s0);

4  procedure expand_state(s)
5      work_set(s) := ample(s);
6      while work_set(s) is not empty do
7          let α ∈ work_set(s);
8          work_set(s) := work_set(s) \ {α};
9          s' := α(s);
10         if new(s') then
11             hash(s');
12             set on_stack(s');
13             expand_state(s');
14         end if;
15         create_edge(s, α, s');
16     end while;
17     set completed(s);
18 end procedure

```

## Ample sets

### General requirements on *ample*

1. pruning with ample does not change the outcome of the MC run (**correctness**)
  2. pruning should, however, cut out a *significant* amount
  3. calculating the ample set: not too much *overhead*
- so far:
    - quite wishy-washy, only general idea
    - “unrealistic” (as mentioned)
  - details also dependent on the “programming language”
  - alternatives of *ample sets* with analogous ideas (the names are not really indicative of how all that works):
    - sleep sets
    - persistent sets
    - stubborn sets
    - ...

**With a little help of the programmer ...**

- for instance: Spin
- Spin: early adoptor of POR
- reduce the amount of interleavings

**atomic** atomic block executed indivisibly

**D\_step** deterministic code fragment executed indivisibly.

**Rest**

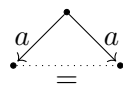
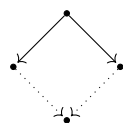
- D\_step more strict than atomic (eg. wrt. goto statements)

**5.2 Independence and invisibility****2 relations between relations**

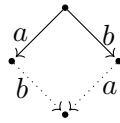
- we have labelled transitions (resp. multiple relations)
- 2 important conditions for POR
  - one connects *two relations*
  - one connects one relation with the property to verify

**Independence** roughly: the order of 2 independent transitions does not matter.

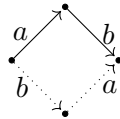
**Invisible** Taking a transition does not change the satisfaction of relevant formulas

**Determinism, confluence, and commuting diamond property****Determinism****Diamond prop.**

**Comm. d-prop.**



**“Swapping” or commuting**



and vice versa

**Independence**

- assume: transition relations  $\xrightarrow{\alpha_i}$  *deterministic*
- write  $\alpha_i(s)$  for  $s \xrightarrow{\alpha_i}$

**Definition 5.2.1** (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  $(\xrightarrow{\alpha_1}, \xrightarrow{\alpha_2}) \in I$

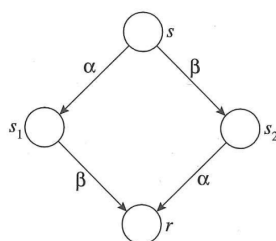
**Enabledness** If  $\alpha_1, \alpha_2 \in \text{enabled}(s)$ , then  $\alpha_1 \in \text{enabled}(\alpha_2(s))$

**Commutativity:** if  $\alpha_1, \alpha_2 \in \text{enabled}(s)$ , then

$$\alpha_1(\alpha_2(s)) = \alpha_2(\alpha_1(s))$$

- *dependence relation:*  $D = (\rightarrow \times \rightarrow) \setminus I$

**Is that all?**



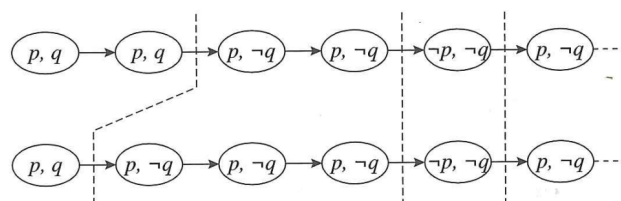
**2 issues**

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.

**Visibility**

- $L : S \rightarrow 2^{AP}$
- $\xrightarrow{\alpha}$  is **invisible** wrt. to a set of  $AP' \subseteq AP$  if for all  $s_1 \xrightarrow{\alpha} s_2$

$$L(s_1) \cap AP' = L(s_2) \cap AP'$$

**Blocks and stuttering**

stuttering equivalent paths

- **block**: finite sequence of intentially labelled states
- stuttering (in this form): important for *asynchronous* systems

**Stutter invariance** An LTL formula  $\varphi$  is *invariant under stuttering* iff for all pairs of paths  $\pi_1$  and  $\pi_2$  with  $\pi_1 \sim_{st} \pi_2$ ,

$$\pi_1 \models \varphi \quad \text{iff} \quad \pi_2 \models \varphi$$

**Next-free LTL**

- $\bigcirc$  breaks stutter invariance
- $LTL_{\neg\bigcirc}$ : “next-free” fragment of LTL (often also  $LTL_{\neg X}$ )

**Stuttering**

- Any  $LTL_{\neg\bigcirc}$  property is invariant under *stuttering*
- Any LTL property which is invariant under stuttering is *expressible* in  $LTL_{\neg\bigcirc}$

**5.3 POR for  $LTL_{\neg\bigcirc}$** **POR for  $LTL_{\neg\bigcirc}$** 

- general useful and fruitful setting for POR
- of course: one may look more specific for specific formulas
- in that setting:



**Correctness of POR** Ample sets prune the (DFS) search. **Goal:**

$$\mathcal{M}, s \models \varphi \quad \text{iff} \quad \mathcal{M}^{\succ_s}, s \models \varphi$$

- note: “iff”
- mainly a condition on *paths*

**Path representatives** each path  $\pi_1$  in  $\mathcal{M}$  starting in  $s$  is represented by an **equivalent** path  $\pi_2$  in  $\mathcal{M}^{\succ_s}$ , starting in  $s$

### Conditions on selecting ample sets

#### 4 conditions for selecting ample set

- each pruned path can be “reordered” to an which is explored (using **independence**). include a condition covering end-states
- make sure that the reordering (pre-poning) does not change the logical status (**stuttering, visibility**)
- “fairness”: make use not to prune “relevant” transitions by letting the search **cycle** in irrelevant ones.

The discussion is based on the presentation in [11].

**C1** That’s said to be the most complex condition. It seems to have been ediscussed (maybe even under different names) in the literature. It is furthermore stressed, that the condition explicitly refers to the *full* unreduced state graph and paths throughout those. Path here are alternating state-transition sequences. Note that unlike in “basic semantics” for LTL, the paths are also path *through* the transition system or graphs. It’s important, insofar that the states in the graph are relevant, in particular also which other actions are alternatively enabled, resp. which alternative actions are in the ample sets at a given state. So, a path (or execution) not a fully linear structure, it’s a bit like the picture of “barbed wire” in the definition of bisimulation, or of resusal sets etc.

Anyway, the condition C1 states

Along *every* path in the *full* state graph that starts at  $S$ , the following condition holds: a transition that is dependent on a transition in  $\text{ample}(s)$  cannot be executed without a transition in  $\text{ample}(s)$  occurring first.

The choice of word “depending on” is a bit unfortunate. Dependence, which is the opposite of independence, is *symmetric*. “Interdependence”, to my ears, would be a better word. Also the status of “events” are unclear, it’s all quite abstract, but independence means: not disabling, therefore dependence can mean disabling (but there’s no talk about “enabling”). Let’s postpone that a bit.

Note that the condition also does not speak about *individual* transitions. It talks about “some” transitions. Timewise, there are 3 points in time relevant: now, in state  $s$ , in the

future, and in between those 2 timepoints. But remember: the condition talks about *all* paths starting in  $s$ , the condition and the 3 time points refer to all those.

As a consequence of the definition: in a state, all enabled ones but not ample are independent from the ample ones. That's indeed a direct consequence of the condition. 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.

The definition of “ample” has two aspects. One is the next-step condition, that's the one use in the algorithm. Ample transitions are the prioritized ones, the one in the focus. For the next-step picture, it's black and white: there is a set of enabled transitions. The ample ones are explored, the rest not. We are talking here in the “unrealistic” setting that we have the unrestricted graph at hand and the ample set allows to restrict the DFS in the described manner. Choosing one element locally from the ample set may *disable* an alternative, so there may be a *choice* inherent at a given point. Of course, we are here not dealing with single *executions* or *runs*, but with model checking. So the DFS will come back eventually and explore relevant alternatives as well.

Now, that was the next-step perspective on the ample sets. There is also the temporal aspect, capture in condition C1: for *all* runs starting somewhere, the relevant ones takes some form of priority as well. The condition is a bit more tricky than just having the relevant ones in a state before the irrelevant ones. The reason is, that transitions can enable as well as disable other transitions. However, that's not universally true. Independent transitions cannot *disable* each other (enabling is allowed). The ample set (the relevant transitions) are closed under interdependence. As stated above in the lemma, it's an easy consequence of the “definition” of the relevant and irrelevant next steps, that the two sets are independent.

It's straightforward to prove: *2 swappable and deterministic relations that don't disable each others satisfy the commuting diamond property.*

This statement forbid disabling. The requirement about prohibitin enabling is used if one wants to prove the reverse implication. But it does not seem to enter the picture here. That sounds also a bit dubious to call two relations independent if one may enable the other. It seems to be related to a subtle distinction on the formulation of “swappability” and the condition of commutativity as part of the independence here. The requirement corrending to “no-enabling” is covered by the fact that commutativity in the formulation of [11] requires that the to relations involved for which the commutation holds are both enabled.

Now, coming back to the definition of condition C1. The ultimate goal which one hopes co capture with C1 is of course correctness of the modified DFS, i.e., one does not miss out relevant paths.

The argument looks at path in the unreduced graphs (which is what C1 talks about, and likewise in principle also C0, except that it does not even talk paths, just about states, so in that perspective there is no difference between the reduced graph and the non-reduced one). Alright. If we taken an arbitrary path in the non-reduced graph, starting at a point, one is not forced to start with a relevant transition (from the ample set). That's what the reduced graph would do. The liberal behavior can do things outside the ample set. Of

course, we should be aware that that it's not “the” ample set, the one where we start the path. After doing a step, we are in a different state, and that state may have a different ample set (and a different set of low-priority transitions, since the set of enabled transitions can surely change). However, by taking an irrelevant  $\beta$ , we know that it's independent (as required by C1) and therefore it cannot disable the relevant ones. In the successor states, the situation is new, but the set of relevant transitions cannot get smaller. They may be other irrelevant transitions, but again according to C1, those (perhaps different) transitions are likewise subject to the independence requirement, now in the changed state. Still, they cannot disable the “older” or “inherited” transitions, so they keep on lingering, as long as one takes non-relevant transitions. One can also not discard them by running into a dead end. That's forbidden by C0. That means, for an element  $\alpha$  of the relevant transitions in a state. In a path starting there, there will finitely many  $\beta$ s followed by an  $\alpha$ , or there are infinitely many  $\beta$ s. Again, the definition is semi-fixed on the state  $s$  with respect to the  $\alpha$ s, but the  $\beta$ s are not defined relative to  $s$ .

Anyway, assume we are at state  $s$  and look at paths starting there. Now, the path can have two forms, either it has a prefix  $\vec{\beta}\alpha$  or it's infinite  $\vec{\beta}\dots$ , where  $\alpha$  is in the ample set of  $s$  and the  $\beta_i$  are all independent for all transitions in the ample set of  $s$  (which includes  $\alpha$ ). Note that at this particular point the text does not require that the  $\beta$  are outside  $ample(s)$  (which would also be a weaker requirement than requiring that the actions are from  $enabled(s) \setminus ample(s)$ , of course). At that point in the text, the only requirement is that the  $\beta$  are independent from  $ample(s)$  (but  $ample(s)$  may contain elements which are among themselves independent, that means,  $\alpha$  may actually *not* be the first element from  $ample(s)$  that is being chosen in the first sequence). Similar remarks apply to the second form of an infinite  $\beta$ -sequence.

The text may not require that, but either intuitively it's assumed (because that's the interesting case), resp. that additional assumption is added in the further discussion. Maybe they are just slightly sloppy. Anyway, without adding the additional requirement, the sequence  $\vec{\beta}\alpha$  may actionally be taken by the modified DFS exploration (it's only not interesting). Interesting is only the case where the  $\beta$ s are all *not* from  $ample(s)$ .

In that case where  $\alpha$  is the *first* element from  $ample(s)$ , the sequence  $\beta_0, \dots, \beta_m\alpha$  is indeed *not* chosen by the modified DFS (in case  $m > 0$ , which is also not mentioned in the text).

So, let's refer to sequences written  $\vec{\beta}$  and starting in  $s$  containing *no* members of the relevant set  $ample(s)$ .

The formulation in the book is a bit shifting with the

The sequences of the form  $\vec{\beta}\alpha$  may *not* be taken by the modified DFS. Technically, the book here is imprecise, it states that sequences of that form *are* in fact not taken. That seems not correct. It's not a real error. The argument that follows states that instead of this sequence, another one is taken. That's true independent from the question whether the original sequence  $\vec{\beta}\alpha$  is taken by the algo or not. Therefore one could say, the given argument covers only the interesting case where the algorithm does not take that sequence. The text also assumes that none of the elements of  $ample(s)$  are taken in  $\vec{\beta}$ . Also that does

⊗

#

cycle condition Willems and Wolper [30]

1. Choice, conflict, and interdependence  
Choices are covered, I think.
2. Transitions, relations, and events

**Reordering conditions ( $C_0, C_1$ )** **$C_0$ : stop at a dead end, only**

$$ample(s) = \emptyset \text{ iff } enabled(s) = \emptyset$$

**$C_1$**  Along every path in  $\mathcal{M}$  starting at  $s$ , the following condition holds: a transition **dependent** on a transition in  $ample(s)$  cannot be executed without a transition from  $ample(s)$  occurring *first*.

- easy fact:  $ample(s) \bowtie \neg ample(s)$

**Form of paths in  $\mathcal{M}^{\succ s}$** 

- consequence of  $C_1$ : two forms of paths

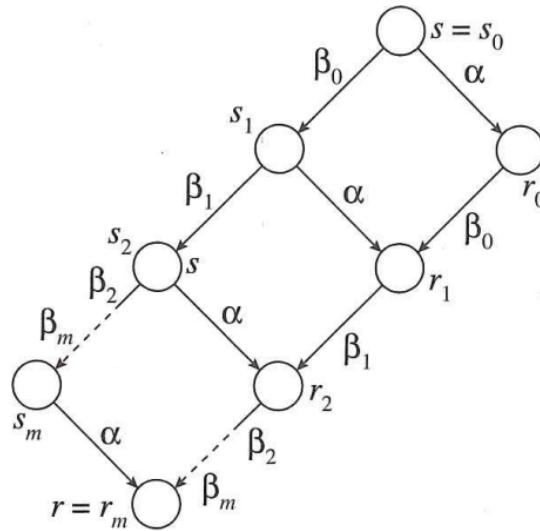
**Blocks**

1. with prefix  $\beta_0\beta_1 \dots \beta_m\alpha$ 
    - $\alpha \in ample(s)$
    - $\beta_i \bowtie ample(s)$
  2. without such prefix:
    - infinite  $\beta_0\beta_1\beta_2 \dots$
    - $\beta_i \bowtie ample(s)$
- assume: all  $\beta_i \notin ample(s)$
  - same as  $\beta_i \in \neg ample(s)$ ?

**Commutation****path  $\vec{\beta}\alpha$  in  $\mathcal{M}$ , starting in  $s$** 

- $\alpha \in ample(s), \beta_i \notin ample(s)$

1. Pic



## 2. Paths

- $\pi_1 = \vec{\beta}\alpha$
- $\pi_2 = \alpha\vec{\beta}$
- $\pi_1 \in \mathcal{M}$  implies  $\pi_2 \in \mathcal{M}$  (and vice versa)
- **what about  $\mathcal{M}^{\infty}$ ?**:  $\pi_1 \notin \mathcal{M}^{\infty}$  ( $m > 0$ ) and  $\pi_2 \in \mathcal{M}^{\infty}$

**Explanations** The assumptions of *independence* means that, in the original transition system  $\mathcal{M}$  the following holds: if (starting in  $s$ ) path  $\pi_1$  is possible, then so is  $\pi_2$ , both ending in the same end state. The reason is that part of the condition of independence is that actions can be swapped or commuted. So, as far as their *existence* in  $\mathcal{M}$  is concerned,  $\pi_1$  and  $\pi_2$  are “equivalent” (and all the “intermediate” paths as well, like  $\beta'\alpha\beta''$ ).

In the pruned system  $\mathcal{M}^{\infty}$ , things change. In particular, the “upper” path  $\pi_1$  which puts  $\alpha$  at the end, does not exist (in case  $m > 0$ ): we assumed that in particular,  $\beta_0 \notin \text{ample}(s)$ , so already the first step is not possible.

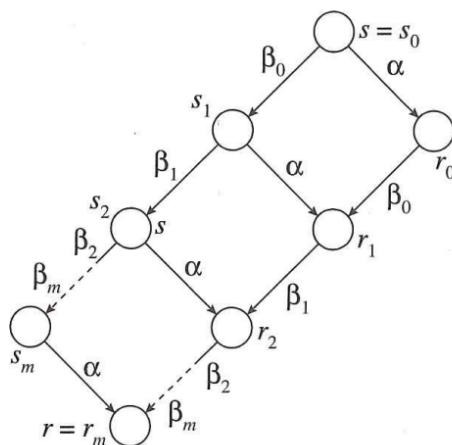
Now, as said, both paths are interchangeable wrt. their existence: if one path exists, it’s guaranteed that the other exists, and vice versa and they have the same start and end state ( $s$  and  $r$  in the picture). But are they interchangeable also wrt. to the intermediate, visited state, in particular, are the two paths interchangeable wrt. the property we model check? Well, one path visits  $s_0, s_1, \dots, s_m, r$  the other one  $s, r_0, \dots, r_m$  (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 through the states  $r_i$  or the states  $s_i$ ?

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  $\alpha$ , i.e.

$$s_i \xrightarrow{\alpha} r_i$$

Now, whether  $\pi_1$  or  $\pi_2$  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  $\alpha$  is invisible (with respect to the atomic propositions)

**Does it make a difference how to go from  $s$  to  $r$ ?**



- $\pi_1$  and  $\pi_2$  (and intermediate mixtures): “interchangable”
- start and end point equal
- but: does it matter which one is taken
  - wrt. the logical **property**, i.e.,
  - does it matter which **intermediate states** are visited?

$$s_i \xrightarrow{\alpha} r_i$$

**Explanation** The answer is clearly *no, it does not matter* provided that the satisfaction 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 does 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.

### Invisibility of transitions

- remember: **invisibility** if transitions (by sets of atomic propositions)

**C<sub>2</sub> (invisibility)** If  $s$  is not fully expanded, then every  $\alpha \in ample(s)$  is *invisible*.

Partial order reduction allows to ignore steps. In the way it's presented here, locally, per state, the (DFS) exploration focuses “ample sets” of the enabled next steps, and omits the rest. That's interesting only if really some elements are ignored. As shown in the above example, we have to be careful when ignoring transitions (for instance the  $tsb_0$ ).

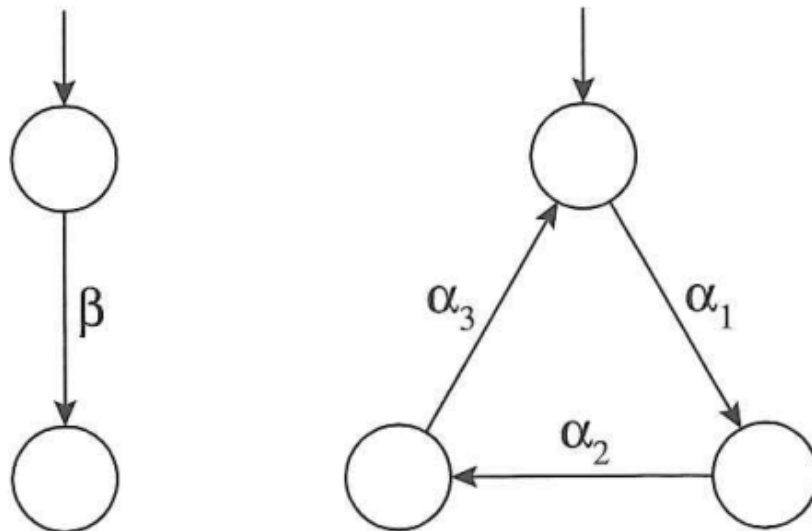
If we ignore transitions, the non-ignored transitions must all be *invisible*.

As for terminology on the slide: A state  $s$  is **fully expanded** if  $ample(s) = enabled(s)$ . 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.

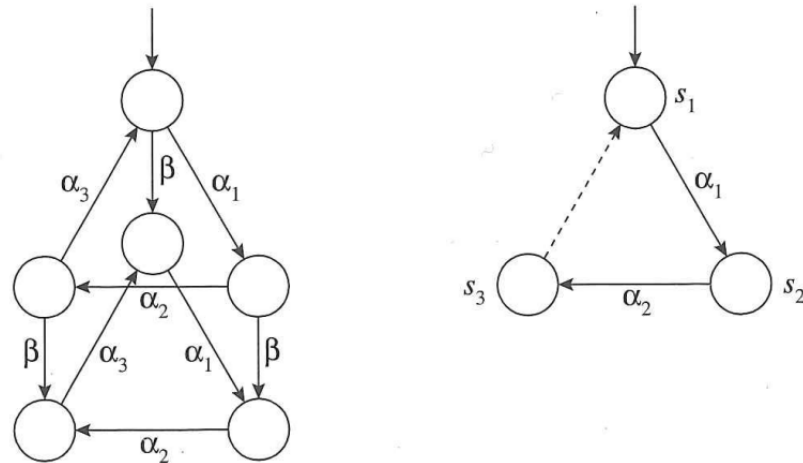
**Is that all?**

**Pics**

1. Two concurrent procs



2.  $\mathcal{M}$  and  $\mathcal{M}^{\times\infty}$



### Is that all?

The previous condition insisted on invisibility of an action  $\alpha$ , in case one omits alternatives. The picture shown previously illustrated that, that if  $\alpha$  is invisible, the uncovered path (in the picture) with  $\alpha$  at the end can be reordered with  $\alpha$  at the beginning *without* omitting intermediate states with different logical status. That last condition about invisibility took care about *one* form of paths that are required as consequence of condition  $C_1$ , namely the one with finitely many  $\beta_i$ 's (not in the ample set of a state  $s$ ) followed by one  $\alpha$  from the ample set of  $s$ .

The final condition  $C_3$  needed for the correctness of pruning the exploration to focus on the ample-sets has to do with the second form of paths that follow as consequence of  $C_4$ , namely the ones with infinitely many  $\beta_i$ , and *never* any  $\alpha$  from the ample set.

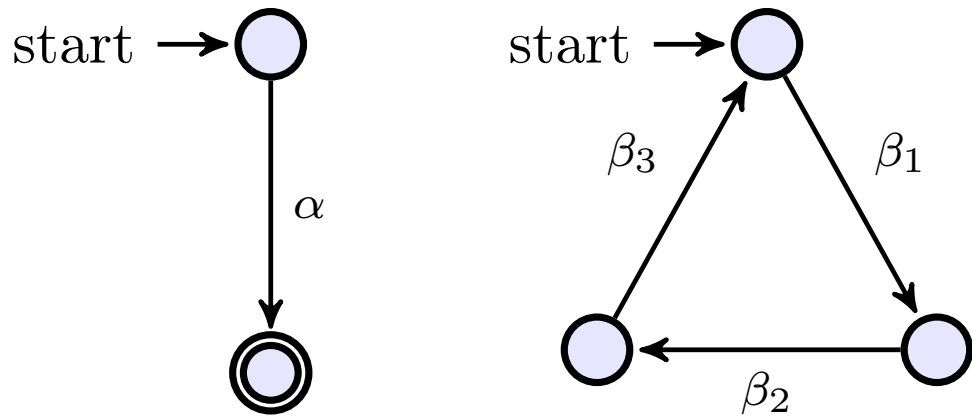
Based in the intuition of the ample sets we can already intuitively see that one has to be careful there. The ample set in a state represent the transitions that should be explored, and the rest from  $\neg ample(s)$  are the one that intended to be ignored (because one can argue that they are equivalently covered otherwise during the exploration). Now, a transition in the ample set of a state marks it as “the transition needs to be explored”. Postponing it “forever” is not the way to go.

The condition  $C_3$  (like the other conditions) is not a condition on the behavior or the form of paths (like “don't look at paths where transitions  $\alpha \in ample(s)$  are postponed forever”), it's a general 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).

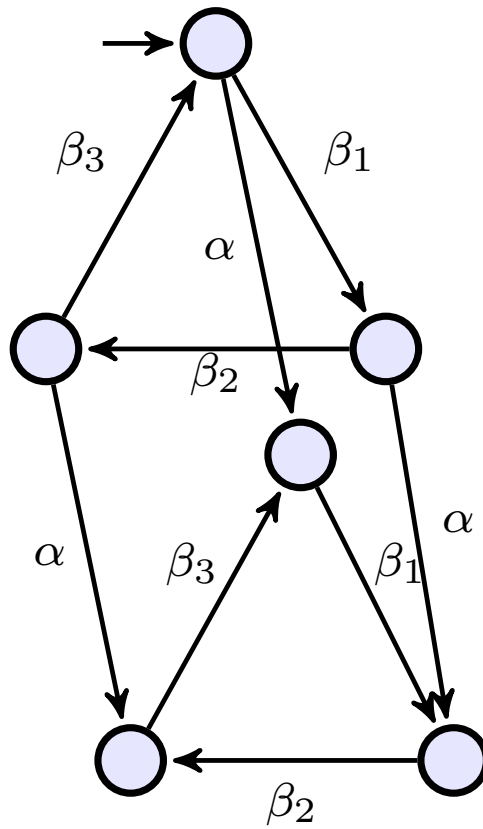
### Pics

1. Two concurrent procs

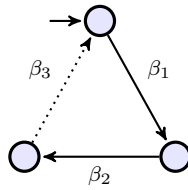




2.  $\mathcal{M}$



$\mathcal{M}^{\approx}$



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.

We make it very simple; the goal is

- $\alpha$  is **visible**
- the  $\beta_i$ s are **invisible** (the second process stuttering)

Intuitively it can be achieved by assuming that there is one boolean variable, initially say “false”, and the process to the left sets it to “true” via its transition  $\overset{\alpha}{\rightarrow}$ . The “label”  $\alpha$  may represent the assignment  $p := \text{true}$ . The other process does not do anything (except spinning around, cycling through its three states).

The first pc show two processes running in parallel in an asynchronous fashion, i.e., interleaving their steps. The overall combined behavior is given by the transition system  $\mathcal{M}$ , with 6 states. In that  $\mathcal{M}$  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*  $\alpha$ , i.e., not including this transition in any of the ample sets. I.e., each state has an one element ample set  $\text{ample}(s) = \{\beta_i\}$ , and  $\alpha$  is not included anywhere.

It’s easy to check that this choice satisfies  $\mathbf{C}_0$  (trivially, since no ample set or enabled set is empty),  $\mathbf{C}_1$  (since  $\alpha$  is assumed to be independent from the  $\beta_i$ s; remember that  $\mathbf{C}_1$  speaks about paths in  $\mathcal{M}$ , not in  $\mathcal{M}^{\times_s}$ ). And finally  $\mathbf{C}_2$  is satisfied as well, as the example is constructed in such a way that the  $\alpha_i$  are all **invisible**, as required by  $\mathbf{C}_2$ .

Transition  $\alpha$  is visible (it does not stutter), so taking it matters wrt. the verification. However, the ample sets chosen as given, leads to explorations in  $\mathcal{M}^{\times_s}$  ignoring  $\alpha$ .

The last condition  $\mathbf{C}_3$  on the next slide exclude 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 ensure is that a situation as in the example cannot occur. That  $\overset{\alpha}{\rightarrow}$  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. It would allow (as in the example) to construct a path running through this cycle where the transition is constantly enabled

but always in  $\neg \text{ample}(s_i)$ , so no state “takes responsibility” to at least one time, explore that edge. In the example, the neglected edge  $\alpha$  is a **visible** one. But the requirement stating “do not systematically neglect an edge” also applies to invisible ones as well. Even if an edge 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  $\xrightarrow{\alpha}$ , what 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 behavior. 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 “chooses” 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 fair). 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 enabled*. The chosen infinite path  $\beta_1\beta_2\beta_3\beta_1\dots$  has an infinite sequence of points where  $\alpha$  is *constantly* enabled. Weak fairness requires that one cannot have an action (like  $\alpha$ ) enabled infinitely long without also taking it. fairness

Strong fairness says: if 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.

A final side remark (to be relevant perhaps for POR): as part of the illustration example, the chosen  $\beta_i$  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 amount of do-nothing steps (here formulated by having the  $\xrightarrow{\alpha_i}$  as invisible). They have no influence on the satisfaction of formulas. More practically, one can see them 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).

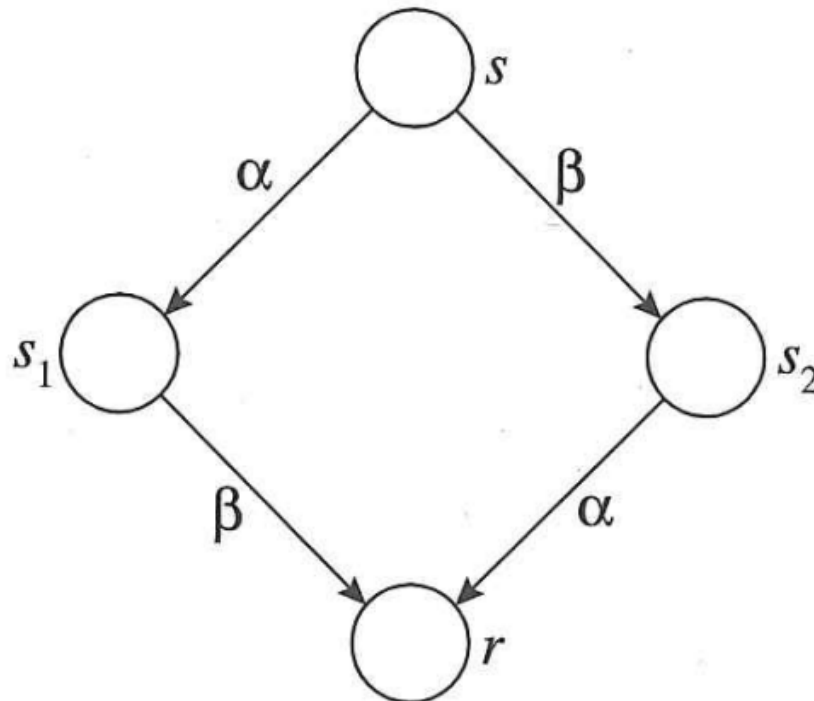
Either way: infinitely many do-nothing or skip or stutter steps is seen as a simple and discrete form of so called Zeno-behavior. That’s in honor of an old Greek philosopher Zeno of Elea, who is remembered for some speculative paradoxes (retold by Aristotle), often concerning infinitely many (smaller and smaller time) steps. The most well-known of those is probably the tale of Achilles and the tortoise, racing against each other.

**Cycle condition C<sub>3</sub>**

**C<sub>3</sub>** A cycle is not allowed if it contains a state in which some transition  $\alpha$  is enabled but never included in  $ample(s)$  for any state  $s$  on the cycle.

**Remember the 2 issues****Repetition**

## 1. Illustration



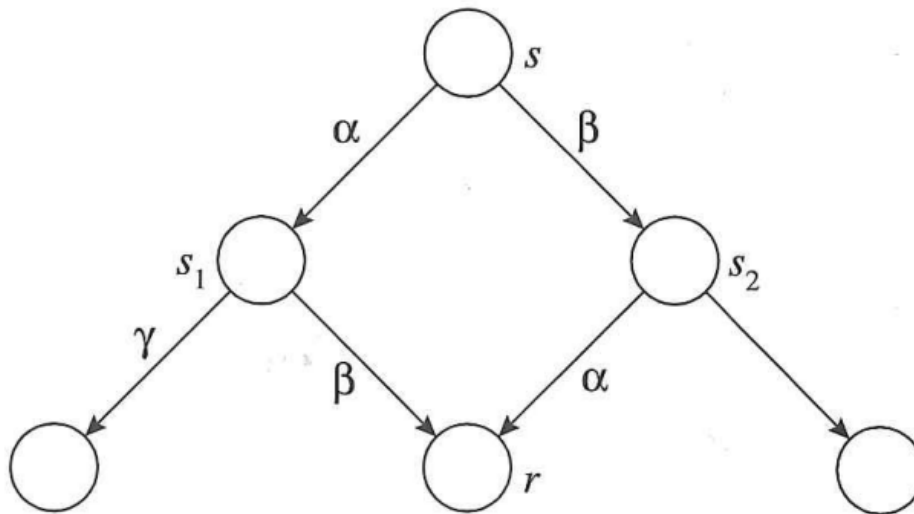
## 2. Text

- a) satisfaction depends in choosing path via  $s_1$  or  $s_2$ ?
- b) forgotten successors?

**Rest**

- assume:  $s_1$  is omitted ( $\beta \in ample(s)$ , but not  $\alpha$ )

## 1. issue 2



2. the conditions imply
- a)  $ss_2r \sim_{st} ss_1r$
  - b)  $ss_1s'_1 \sim_{st} ss_2rr'$

### 5.3.1 Calculating the ample sets

#### Complexity

- checking conditions on-the-fly
- $C_0$ : easy
- $C_1$ : tricky
  - refers to  $\mathcal{M}$ , not  $\mathcal{M}^\infty$
  - checking  $C_1$ : equivalent to reachability checking
- strengthen  $C_3$ :

#### sufficient for $C_3$

- at least one state along each cycle must be fully expanded

#### Rest

- since we do DFS: watch out for “back edges”:  $C'_3$ : If  $s$  is not fully expanded, then no transition in  $ample(s)$  may reach a state that is on the search *stack*

### 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

### Notions, notations, definitions

- we write now  $\alpha$  for  $\xrightarrow{\alpha}$
- fixed, finite set of processes  $i$  (called  $P_i$ )
- $T_i$ : those transitions that “belong to”  $P_i$
- some more easy definitions
  - $pc_i(s)$ : value of program counter of  $i$  in state  $s$
  - $pre(\alpha)$ :
    - \* transition whose execution *may* enable  $\alpha$
    - \* can be over-approximative
  - $dep(\alpha)$ : transitions interdependent with  $\alpha$
  - $current_i(s)$
  - $T_i(s)$

### When are transitions (inter)dependent

- note: dependence is *symmetric*! (good terminology?)

**Shared variables** pairs of transitions, that *share* a variables which is changed (or written?) by at least one of them

**Same process** pairs of transitions belonging to the *same process* are interdependent. In particular  $current_i(s)$

### Message passing

- 2 sends to the same channel or message queue
- 2 receives from the same channel
- **Note** send and receive independent (also on the same channel).
- side remark: rendezvous is seen/ can be seen a joint step of 2 processes

**Transitions that may enable  $\alpha$  ( $pre\alpha$ )**

$$pre(\alpha) \supseteq \{\beta \mid \alpha \notin enabled(s), \beta \in enabled(s), \alpha \in enabled(\beta(s))\}$$

- assume  $\alpha$  is an action from  $P_i$
- $pre(\alpha)$  includes
  - “local predecessor” of  $i$  (“program order”)
  - **shared variables**: if enabling conditions of  $\alpha$  involves shared variables: the set contains *all other transitions* that can change these shared variables
  - **message passing**: if  $\alpha$  is a send (reps. receive), the  $pre(\alpha)$  contains transitions of other processes that receive (resp. send) on the channel

**Ample**

```
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
```

**Check C<sub>2</sub>**

```
function check_C2(X) =
  for all  $\alpha \in X$ 
  do if visible( $\alpha$ )
    then false
    else true
```

**Check C'<sub>3</sub>**

```
function check_C3'(s, X) =
  for all  $\alpha \in X$ 
  do
    if on_stack( $\alpha(s)$ )
    then false
    else true
```

**Check C<sub>1</sub>**

```
function check_C1 (s, Pi) =  
  for all Pj ≠ Pi  
  do  
    if      dep(Ti(s)) ∩ Tj ≠ ∅  
      ∨  
      pre(currenti(s) \ Ti(s)) ∩ Tj ≠ ∅  
    then return false  
  end forall;  
  return true
```



# Chapter 6

## Symbolic execution

### Learning Targets of this Chapter

The chapter gives an not too deep introduction to *symbolic* execution and *concolic* execution.

### Contents

6.1	Introduction . . . . .	165
6.2	Symbolic execution . . . . .	171
6.3	Concolic testing . . . . .	176

What  
is it  
about?

## 6.1 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 clipped in from that presentation. More material may be found in the survey paper [3].

### Introduction

- **symbolic** execution: “old” technique [21]
- natural also in the context of **testing**
- **concolic** execution: extension
- used also in compilers
  - code generation
  - optimization

### Code example

```

1 f(int x, int y) {
2   if (x*x*x > 0) {
3     if (x > 0 && y == 10) {
4       fail();
5     }
6   } else {
7     if (x > 0 && y == 20) {
8       fail();
9     }
10  }
11
12  complete();
13 }
```

The code does not have any particular purpose, except that it will be used to discuss testing, symbolic execution, and a concept called 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, loops pose challenges for symbolic execution. The problems are similar to the challenges to bounded model checking, which was covered by one of the earlier student presentations. BMC is a technique which shares some commonalities with symbolic execution: both are making use of SAT/SMT solving.

### How to analyse a (simple) program like that?

- **testing**
- “verification” (whatever that means)
  - could include code review
- model-checking? Hm?
- symbolic and concolic execution (see later)

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., unmanageably 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 work 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  $\pm 1$  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 is considered out-dated by some, but widely used anyway).

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 criterial. 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 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 (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.

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*.

Programs typically contain more control structure than just one single condition. 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 conjunction.

There is much more to say about symbolic execution as a field, but that's the core idea in a nutshell.

### 6.1.1 Testing and path coverage

#### Testing

- maybe **the** most used method for ensuring software (and system) “quality”
- broad field
  - many different testing goals, techniques
  - also used in combination, in different phases of software engineering cycle
- here: focus on

#### “white-box” testing

- AKA structural testing
- program code available (resp. CFG)
- also focus: *unit* testing

#### Goals

- detect errors
- check corner cases
- provide high (“code”) **coverage**

#### (Code) coverage

- note: typically a non-concurrent setting (unit testing)
- different coverage criteria
  - nodes
  - edges, conditions
  - combinations thereof
  - path coverage
- defined to answer the question

When have I tested “enough”?

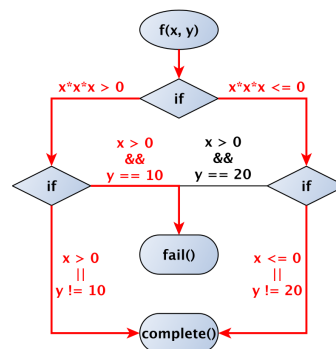
## path coverage

- ambitious to impossible (loops)
- note: still not *all reachable states*, i.e., not verified yet

As mentioned earlier, *path* coverage is often considered as too ambitious as coverage criterion. Of course, sometimes tests cannot cover 100% of the simpler criteria 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 program may actually terminate, but that’s something one cannot see in the CFG, where one can see just a cycle in the graph.

## Path coverage

The picture shows the control-flow graph of the program from the beginning.



In the presentation slides, there is a number of overlays (not reproduced here) that show different paths from the start node till one of the terminal nodes, in connection with the program code. The paths are marked red in the picture, and there are 3 paths marked that way. There are actually 4 different paths in the CFG, but one of them reaching the failure end state via the route on the right does not correspond to a possible execution. That’s easy to see, insofar that would imply a value for  $x$  satisfying the constraint

$$(x^3 \leq 0) \wedge (x > 0 \wedge y = 20) . \quad (6.1)$$

The corresponding path is thus not colored in red in the picture. The constraint from equation (6.1) is an example of a *path condition* or, synonymously, *path constraint*, namely the one for a path, which happens to be unrealizable as the constraint is unsatisfiable.

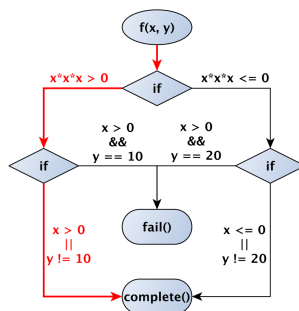
## Random testing

- perhaps most naive way of testing
- generating random inputs
- **concrete** input values
- **dynamic** executions of programs
- *observe actual* behavior and
- compare it against *expected behavior*

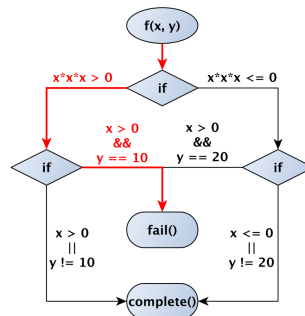
The slide called the approach a “naive” way of testing, but it does mean it has not been used and is being used (and evaluated and compared to other forms, it has been refined etc.) So it has its place.

## Random testing

If one applied the random testing approach to the program from before, one needs to generate random input for the two inputs  $x$  and  $y$ . Two generated values for that pair could be  $(700, 500)$  and  $(-700, -500)$ , for instance. The red path in the figure shows the path the program takes for the first choice. For the second choice, a different path would be chosen, also leading to successful completion.



## One path so far missed



As mentioned, there is an unrealizable 4th path, which we should not count that 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 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).

## 6.2 Symbolic execution

### Symbolic execution

- **symbols** instead of concrete values
- use of **path conditions**, aka **path constraints**

- cf. connection to SAT and SMT
- constraint solver computes real values

Basically we have introduced the core idea of symbolic execution already earlier. Perhaps it's worth interesting that, like in BMC, it's about SMT-solving (not just SAT solving). We are dealing with boolean combinations of constraints over specific domains with specific *theories* (like integers, or arrays, etc.), that corresponds to data types used in the programming language used for the programs we are analyzing. From the presentations about BMC and constraint solving, we also are aware, that theories may easily lead to undecidability of constraint solving. Integers with only addition have a decidable theory (known as Presburger arithmetic). 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 realistic. Indeed, one can work with a theory capturing those "realistic" integer, also "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 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. First we finish up with symbolic execution by looking at a super-simple example (but without adding new technical content to the material, it's more like rubbing it in a bit more).

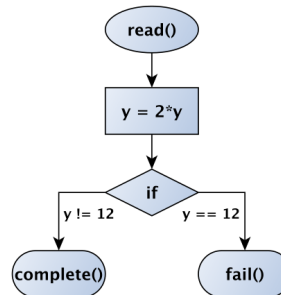


## Simple example

```

1 | y = read();
2 | y = 2 * y;
3 |
4 | if (y == 12) {
5 |     fail();
6 | }
7 |
8 | complete();

```



- in the code: *assignments* not equations ( $y := \text{read}()$ )
- introduce variable  $s$  for  $\text{read}()$
- assignments
  - $y := \text{read}() \Rightarrow y = s$
  - $y := 2*y \Rightarrow y = 2s$
- branching point in line 4
  - right:  $2s = 12$
  - left:  $2s \neq 12$

The code is even simpler than the previous one. The code uses C-like notation where assignment are written using the  $:=$ -symbol. To differentiate imperative assignment in the programming language from declarative equations used in constraints, the slide writes  $:=$  for the former. The difference should be clear by 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.

## Which input leads to the error?

### Constraint solver

Solve the path constraint  $2s = 12$

- child's play: the solution is  $s = 6$
- but: requires solver that can do “arithmetic”, including multiplication

The point about “multiplication” has been mentioned 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  $2x + 17y < z$  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 inequations (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*.

## In summary

### Symbolic execution for dummies

- take the code (resp. the CFG of the code)
- collect all paths into **path conditions**
  - big conjunctions of all conditions along each the path
  - each condition  $b$  will have
    - \* one positive mention  $b$  in one continuation of the path
    - \* one negated mention  $\neg b$  in the other continuation
- solve the constraints for paths leading to errors with an appropriate SMT solver
  
- works best for loop-free program
- cf. also SSA
- but there is another problem as well (see next)

The remark about SSA may be ignored. SSA stands for static single assignment, a widely used intermediate representation in compilers. It's not the same as path conditions, but shares some commonalities in the treatment of variables. Variables are in most languages not single-assignment, they may be overwritten). However, the SSA format (among other things) introduces “versions” of the source level variables which makes them single-assignment (actually, not really dynamically single assignment, but statically single-assignment). The differentiation between static SA and “real” SA is relevant only when deadlining without loops. In loop-free programs, the SSA format transforms the code into some version which is really single assignment. In that way variables become declarative, like variables in a constraint system and the representation of variables for instance in BMC. This has different advantages when it comes to optimization and analysis of the code, which explains the wide usage of that concept. It's outside the scope of this lecture, though.

### Complex condition $x^3$

- non-linear constraint
- in general **undecidable**
- most constraint solvers throw the towel
- for instance: execution stops, no path covered

Coming back to the code example from the beginning of the chapter, we see that this time, the numerical constraints involved are not linear anymore. So, we are definitely leaving the safe ground of decidable theories.

## What can one do?

What can one do (beyond throwing the towel and accept that SE won't cover all paths)?

- “static analysis”: abstracting
  - cover both path approximately
- theorem proving? one cannot sell that to testers

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 may run into trouble when solving them. By “static analysis” I mean techniques like data flow analysis (or more generally abstract interpretation). Characteristic is there, that one *approximates*. 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 the analysis then 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 legitimize 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), though it's not fundamentally forbidden, it's just too expensive to do in many application. As a consequence, they are less precise, i.e., more approximative. Though problems with undecidability 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  $\{negative, 0, positive\}$ , 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  $x^3$ ) but a constraints in linear arithmetic. 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, therefore 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* analysis. 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.

**Concolic testing** Concrete & Symbolic = "concolic"

## 6.3 Concolic testing

### Concolic testing

- here following *DART*
- combination of two techniques

### Random testing

- concrete values
- dynamic execution

### Symbolic execution

- symbols, variables
- static analysis
- other name: **Dynamic symbolic execution** (DSE)

The slogan is

Execute dynamically & explore symbolically

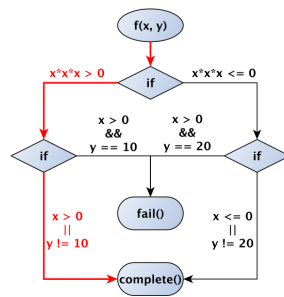


Figure 6.1: Dart (1)

### Dart overview

The following slides show how DART combines random testing and symbolic execution into a *concolic execution* framework. In the slides, different runs are shown in a series of overlays. The script version does not show all the overlays step by step. It just shows, for each iteration, one complete path only. The example is taken from Section 2.5 from Godefroid et al. [16]. It shows how DART could handle the program from before, which involves non-linear constraints. Because of that, standard SMT solvers may not be able to tackle it, since often one restricts to decidable theories (like linear constraints). Also standard overapproximation techniques (“predicate abstraction”) may not be able to precisely analyze a program like that. They would be unable to figure out that fail state is unreachable taking the path “via the right-hand side”. The best they would do is that it “may be reachable”, thus reporting an error that is actually not possible. The overapproximation thus leads to *false alarms*. False alarms are problematic if the user drowns in them. The “tester” will 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 imprecise. Complex programming structures, especially wild pointer manipulations and spaghetti code, by also *dynamic aspects* such as higher-order functions, dynamic or late binding etc. confuses not just the programmer but also lead to radically approximative (= unusable) results. Things get worse when adding concurrency to the mix . . .

For the example. Figure 6.1 shows a possible first run of the DART tool. It starts like random testing, picking an random input, say

$$(x, y) = (700, 500) . \quad (6.2)$$

This input leads to the path marked in red in Figure 6.1. 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 pickig  $x > 0$  and  $y = 10$  is highly unlikely in the first run. Anyway, we start as shown in Figure 6.1.

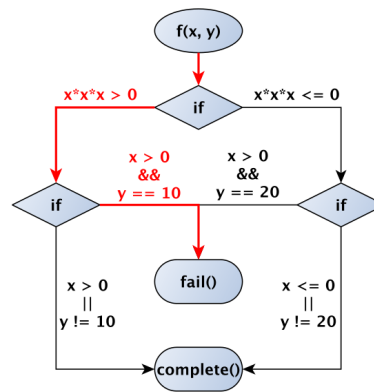


Figure 6.2: Dart (2)

While doing the concrete execution with that input, 2 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 that are 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 chickens 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  $x^3 > 0$ . 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  $y \neq 10$  is true. In this 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 (6.2) led to the *concrete execution* from Figure 6.1, 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 6.2.

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 6.3. 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  $x_n = -700$ .

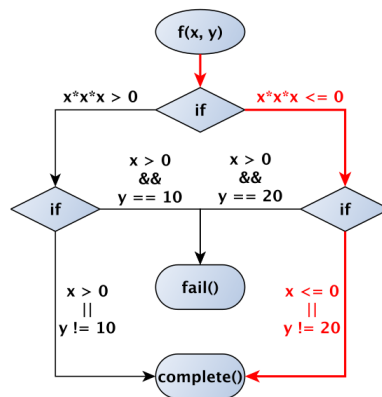


Figure 6.3: Dart ( $n$ )

The conditional leads to the additional constraint  $x_n > 0 \wedge y = 20$ , but that means we have

$$x_n = -700 \wedge x_n > 0 \wedge y = 20 \tag{6.3}$$

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 is 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, the DART method cannot generate new paths any more, it has covered all 3 possible path and the one unrealizable was “covered” insofar that it has been half-symbolically and half-concretely evaluated (see equation (6.3)). 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 illustration, the result would be the same. So with very high probability (and in short time), the method will achieve that coverage.

**Afterthoughts to the example** 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 improbable 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 probability 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).



## 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] Blackburn, P., de Rijke, M., and Venema, Y. (2001). *Modal Logic*. Cambridge University 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] Bradfield, J. and Walukiewicz, I. (2018). The mu-calculus and model checking. In [10].
- [8] Büchi, J. R. (1960). Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92.
- [9] 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.
- [10] Clarke, E. C., Henzinger, T. A., Veith, H., and Bloem, R., editors (2018). *Handbook of Model Checking*. Springer Verlag.
- [11] Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.
- [12] Dijkstra, E. W. (1969). Notes on structured programming. Technical Report EWD-249, Technological University, Eindhoven.
- [13] Emerson, E. A. and Jutla, C. (1991). Tree automata, mu-calculus, and determinacy. In *IEEE Symposium on Foundations of Computer Science (FOCS’91)*.
- [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] Kozen, D. (1983). Results on the propositional  $\mu$ -calculus. *Theoretical Computer Science*, 27:333–354.
- [23] Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143.
- [24] Manna, Z. and Pnueli, A. (1992). *The temporal logic of reactive and concurrent systems—Specification*. Springer Verlag, New York.
- [25] McConnell, S. (2004). *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA.
- [26] Mostowski, A. W. (1984). Regular expressions for infinite trees and a standard form of automata. In Skowron, A., editor, *Computation Theory*, volume 208 of *Lecture Notes in Computer Science*, pages 157–168. Springer Verlag.
- [27] Peled, D. (2001). *Software Reliability Methods*. Springer Verlag.
- [28] Peled, D. (2018). Partial-order reduction. In [10].
- [29] Schneider, K. (2004). *Verification of Reactive Systems*. Springer Verlag.
- [30] Willems, B. and Wolper, P. (1996). Partial-order methods for model checking: From linear time to branching time. In *Proceedings of LICS '96*, pages 294–303. IEEE, Computer Society Press.

## Index

- $\Sigma$ , 82
- $\Gamma \vdash \varphi$ , 34
- $\Sigma$ , 30, 33
- $\doteq$ , 29
- $I$  (interpretation), 31
- $\Box\varphi$ , 42
- $\Diamond\varphi$ , 42
- $V$  (valuation), 38
- $\models$ , 24, 40, 59
- $\llbracket \_ \rrbracket$ , 41
- $[t/x]$ , 28
- $\pi \models \varphi$ , 61
- $\sigma$ , 33
- $\sigma$  (variable assignment), 32
  
- abstract syntax, 26, 30
- abstraction, 139
- accessibility relation, 39
- algebraic signature, 28
- algorithmic verification, 19
- alphabet, 82
- always (temporal operator), 63
- ample set, 144
- angelic choice, 86
- arity, 27, 29
- asynchronous product, 101
- asynchronous vs. synchronous, 140
- axiom, 34
  
- BDD, 139
- binder, 32
- bound variable, 28
- branching time, 58
- Büchi acceptance, 90
  
- closure, 122
- complete lattice, 130
- completeness, 25, 34
- complexity, 138, 139
- compositionality, 139
- concrete syntax, 26, 30
- confluence, 143
- contradiction, 24
- correctness, 4
- coverage, 167
  
- daemononic choice, 86
- data abstraction, 139
  
- deadlock, 5
- decidable, 5
- dependence, 142
- depth-first search, 144
- deterministic finite state automaton, 84
- deterministic relation, 143
- DFA, 82
- Dijkstra's algorithm, 141
- Dijkstra's dictum, 5
- dining philosopher, 5
- discrete time, 58
- domain, 31
- dynamic logics, 50
  
- enabledness, 143
- equality symbol, 29
- eventually (temporal operator), 63
  
- finite mapping, 39
- finite state automaton, 82
- finite state machine, 82
- finite-state automaton, 80
- first-order logics, 24
- first-order model, 31
- first-order signature, 24, 28, 30
- first-order structure, 31, 32
- floating point number, 2
- forcing, 41
- formal method, 21
- formula, 23
- free and bound variable occurrence, 32
- free variable, 28
- free variable, 28
  
- game, 134
- grammar, 120
  
- halting problem, 5, 6
  
- independence, 142
- interactive system, 17
- interleaving, 101, 140
- interleaving concurrency, 141
- interpretation, 24, 31
- invariant, 71
  
- join, 130
  
- Kleene star, 120

- Kripke frame, 37, 38
- Kripke structure, 36, 37, 53, 82
- lattice, 130
  - complete, 130
- Leslie Lamport, 71
- linear time, 58
- liveness, 71
- logics
  - syntax, 23
- LTL, 58
- many-sorted, 31
- meet, 130
- modal logics, 36
- modality, 36
- model, 21
- model theory, 23
- modus ponens, 46
- multi-modal logics, 36
- multi-sorted signature, 28
- next (temporal operator), 63
- NFA, 82
- normal modal logic, 46
- ontology, 32
- parallel composition, 101
- partial order, 85, 130, 140
- partial-order reduction, 139
- path, 60
- path condition, 170
- path constraint, 170
- predicate, 28
- predicate logic, 28
- Presburger arithmetic, 172
- product
  - asynchronous, 101
- proof, 33
- proof theory, 23, 33
- proof tree, 120
- propositional logic, 24
- QTL, 37
- quantifier, 28, 32
- rank, 135
- reactive system, 5, 17
- real-time system, 17
- regular Kripke structure, 53
- release (temporal operator), 64
- Rice's theorem, 5
- rule, 34
- run, 85
- S4, 42
- safety, 71
- safety property, 88
- satisfaction, 40
- satisfaction relation, 40
- satisfiability, 23
- Saul Kripke, 36
- SCC, 109
- scope, 32
- semantic equivalence, 63
- semantical implication, 33
- semantics, 40
- semi-decidable, 5
- shared variable, 142
- signal processing, 2
- signature, 27
  - algebraic, 28
  - first order, 28
  - first-order, 30
  - functional, 27
  - multi-sorted, 28
  - relational, 27
  - signature, 24
  - single-sorted, 28
- simulation, 6
- single-sorted signature, 27, 28
- sort, 27, 28
- soundness, 25, 34
- specification, 6
- Spin, 58
- SSA, 174
- state, 32, 60
- state-space explosion problem, 139
- structural testing, 166
- structure
  - first-order, 32
- stutter, 92
- stutter extension, 92
- substitution, 28
- symbolic model checking, 139
- symmetry reduction, 139

- syntactic sugar, 46
- syntax
  - abstract, 26
  - concrete, 26
- Tarski's fixpoint theorem, 132
- tautology, 24
- temporal logic, 5
- temporal logics, 57
- test, 54
- testing, 5, 6
  - structural, 166
  - white-box, 166
- theorem proving, 19
- time
  - branching, 58
  - linear, 58
- total order, 85, 141
- trace, 111
- transformational system, 17
- transition function, 84
- transition relation, 84
- transition system, 39, 82, 124
- true concurrency, 85, 141
- truth, 23
  
- undecidable, 5
- until (temporal operator), 64
  
- valid, 62, 63
- validation, 6
- validity, 23
- valuation, 32, 39, 54
- variable, 24
- variable assignment, 32
- verification, 6
  
- weak until (temporal operator), 64
- white-box testing, 166