Logic and Verification

Prof. Dr. Jürgen Dix

Computational Intelligence Group
TU Clausthal
Summer Term 2019

What Song the Syrens sang,
or what name Achilles assumed
when he hid himself among women,
though puzzling Questions,
are not beyond all conjecture.

Sir Thomas Browne

Acknowledgment

This course grew out of my former AI course (held from 2004–2014 at TU Clausthal). The main new parts are the chapters about model checking, PROLOG, and answer set programming.
The author gratefully acknowledges material and slides (for the Prolog part) provided by Nils Bulling used in his BSc course TI1906 Logic-Based Artificial Intelligence at TU Delft in the summer term 2015.
Many thanks also to Tobias Ahlbrecht who helped with the exercises.

Time and place: Monday, Tuesday 10–12.
Exercises: See schedule (7 exercises in total).

Website

https://www.in.tu-clausthal.de/divisions/cig/cigroot/teaching/summer-2019/logic

There you will find important information about the lecture, documents, exercises et cetera.

Organization: Tobias Ahlbrecht;
Exercise class: A. Mantel, J. Schwede;
Exam: 16. September 2019, 9:00 (tentative)

What is this course about?

This course is about logic based formal methods and how to use them for verification in computer science.

It rigorously defines the language and semantics of

- sentential logic (SL) and
- first-order logic (FOL).

We introduce the notion of a model, and show how hardware and software systems can be modelled within this framework. We also consider

- semantic entailment vs. syntactic derivability

of formulae and how these two notions are related through correctness and completeness. Completeness is formally proved for SL and extensively discussed for FOL (in the form of refutation completeness).

What is this course about? (cont.)

Both logics are applied to important verification problems:

- SL and its extension LTL for verifying linear temporal properties in concurrent systems, and
- FOL for verifying input/output properties of programs (the Hoare calculus).

We show how the **resolution calculus** for FOL leads to **PROLOG**, a programming language based on FOL.

We also show that SL leads to a declarative version of **PROLOG**: Answer Set Programming (**ASP**), that can be used to solve problems expressible on the second level of the polynomial hierarchy.

While **ASP** is not a full-fledged programming language, it can be used for many naturally ocurring problems and it allows to model them in a purely declarative way.

Lecture Overview

1. Introduction (1 lecture)
2. Sentential Logic (SL) (3 lectures)
3. Verification I: Reactive Systems (3 lectures)
4. First-Order Logic (FOL) (2,5 lectures)
5. Verification II: Hoare Calculus (2,5 lectures)
6. From FOL to PROLOG (2 lectures)
7. PROLOG (3 lectures)

1. Introduction

Content of this chapter (1):

Defining AI: There are several interpretations of AI. They lead to scientific areas ranging from Cognitive Science to Rational Agents.

History: We discuss some important philosophical ideas in the last three millennia and touch some events that play a role in later chapters (syllogisms of Aristotle, Ars Magna).

Content of this chapter (2):

Logic-Based AI since 1958: AI came into being in 1956-1958 with John McCarthy. We give a rough overview of its successes and failures.

Rational Agent: The viewpoint to consider an agent as a rationally acting entity.

Content of this chapter (3):

Model checking: Many problems can be modelled with finite state systems and solved by checking whether (1) a given model has a certain property, (2) a theory is satisfiable or (3) a formula is valid.

Verification: Programs are based on an infinite state space and thus require other methods, similar to the Hoare calculus.

1.1 What Is AI?

"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense" (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)
"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)	"A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990) "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

Table 1: Several Definitions of AI

1. Cognitive science

2. "Socrates is a man. All men are mortal. Therefore Socrates is mortal."
   (Famous *syllogisms* by Aristotle.)

   (1) **Informal description** $\rightsquigarrow$
   (2) **Formal description** $\rightsquigarrow$
   (3) Problem solution

   (2) is often problematic due to under-specification

   (3) is deduction (correct inferences): only enumerable, but **not decidable**

3. Turing Test:

`http://cogsci.ucsd.edu/~asaygin/tt/ttest.html`

`http://www.loebner.net/Prizef/loebner-prize.html`

- Standard Turing Test
- Total Turing Test

Turing believed in 1950:

In 2000 a computer with 10^9 memory-units could be programmed such that it can chat with a human for 5 minutes and pass the Turing Test with a probability of 30 %.

4. In item **2.** *correct* inferences were mentioned.

   Often not enough information is available in order to act in a way that makes sense (to act in a provably correct way).
   $\rightsquigarrow$ Non-monotonic logics.

   The world is in general under-specified. It is also impossible to act rationally without *correct* inferences: **reflexes**.

The year 1943:

McCulloch and **W. Pitts** drew on three sources:

1. physiology and function of neurons in the brain,
2. propositional logic due to **Russell/Whitehead**,
3. Turing's theory of computation.

Model of artificial, connected neurons:

- Any computable function can be computed by some network of neurons.
- All the logical connectives can be implemented by simple net-structures.

The year 1956:

Two-month workshop at Dartmouth organized by **McCarthy**, **Minsky**, **Shannon** and **Rochester**.

Idea:

Combine knowledge about automata theory, neural nets and the studies of intelligence (10 participants)

Newell und **Simon** show a reasoning program, the Logic Theorist, able to prove most of the theorems in Chapter 2 of the *Principia Mathematica* (even one with a shorter proof).

But the *Journal of Symbolic Logic* rejected a paper **authored by *Newell, Simon and Logical Theorist***.

Newell and Simon claim to have solved the venerable mind-body problem.

The year 1958: Birthyear of AI

The term

Artificial Intelligence

is proposed as the name of the new discipline.

McCarthy joins **MIT** and develops:

1. *Lisp*, the dominant AI programming language
2. *Time-Sharing* to optimize the use of computer-time
3. *Programs with Common-Sense*.

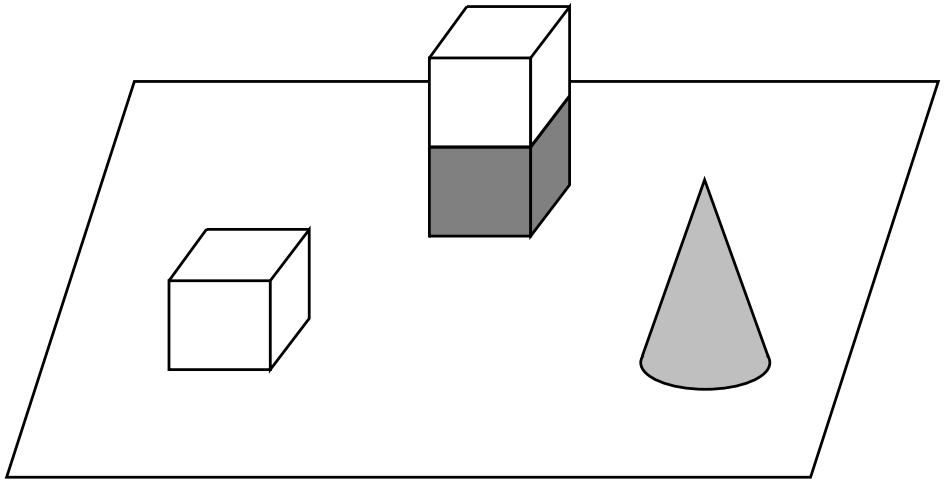
   Advice-Taker: A hypothetical program that can be seen as the first complete AI system. Unlike others it embodies general knowledge of the world.

The years 1960-1966:

McCarthy concentrates on knowledge-representation and reasoning in formal logic ($\rightsquigarrow$ Robinson's Resolution, $\rightsquigarrow$ Green's Planner, $\rightsquigarrow$ Shakey).

Minsky is more interested in getting programs to work and focusses on special worlds, the Microworlds.

Blocksworld is the most famous microworld.



From the 70'ies to the 90'ies

'73: **PROLOG (Colmerauer, Kowalski)**

'74: Relational databases, SQL (**Codd**)

81-91: Fifth generation project (Japan)

'91: **Dynamic Analysis and Replanning Tool (DART)** paid back DARPA's investment in AI during the last 30 years.

From the 90'ies until today

'93: Nine Men's Morris ("Mühle") solved (Gasser (ETH)). #: 10^{10}.

'97: IBM's Deep Blue wins against Kasparow. #: 10^{46}.

'98: NASA's remote agent program: **Deep Space 1**.
Many modules have been model-checked.

From the 90'ies until today (2)

'07: Checkers ("Dame") solved: **Chinook** by J. Schaeffers. #: 10^{20}.

'10: IBM's Watson winning Jeopardy

`http://www-05.ibm.com/de/pov/watson/`

'16: Google's AlphaGo winning **4: 1** against **Lee Sedol** in a professional Go match.
#: 10^{170}.

`https://de.wikipedia.org/wiki/AlphaG`

'17: Google's AlphaZero as a program learning arbitrary games by itself. Even Chess, as one particular instance, and achieving high Elo-ranking.

`https://de.wikipedia.org/wiki/AlphaZero`

1.2 Logic: From Plato To Zuse

450 BC : **Plato, Socrates, Aristotle**

Sokr.: *"What is characteristic of piety which makes all actions pious?"*
Aris.: *"Which laws govern the rational part of the mind?"*

800 : **Al Chwarizmi** (Arabia): Algorithm

1300 : **Raymundus Lullus**: Ars Magna

1646–1716: **G. W. Leibniz:**
Materialism, uses ideas of Ars Magna to build a machine for simulating the human mind

1805 : **Jacquard:** Loom

1815–1864: **G. Boole:**
Formal language,
Logic as a mathematical discipline

1792–1871: **Ch. Babbage:**
Difference Engine: Logarithm-tables
Analytical Engine: with addressable memory,
stored programs and conditional jumps

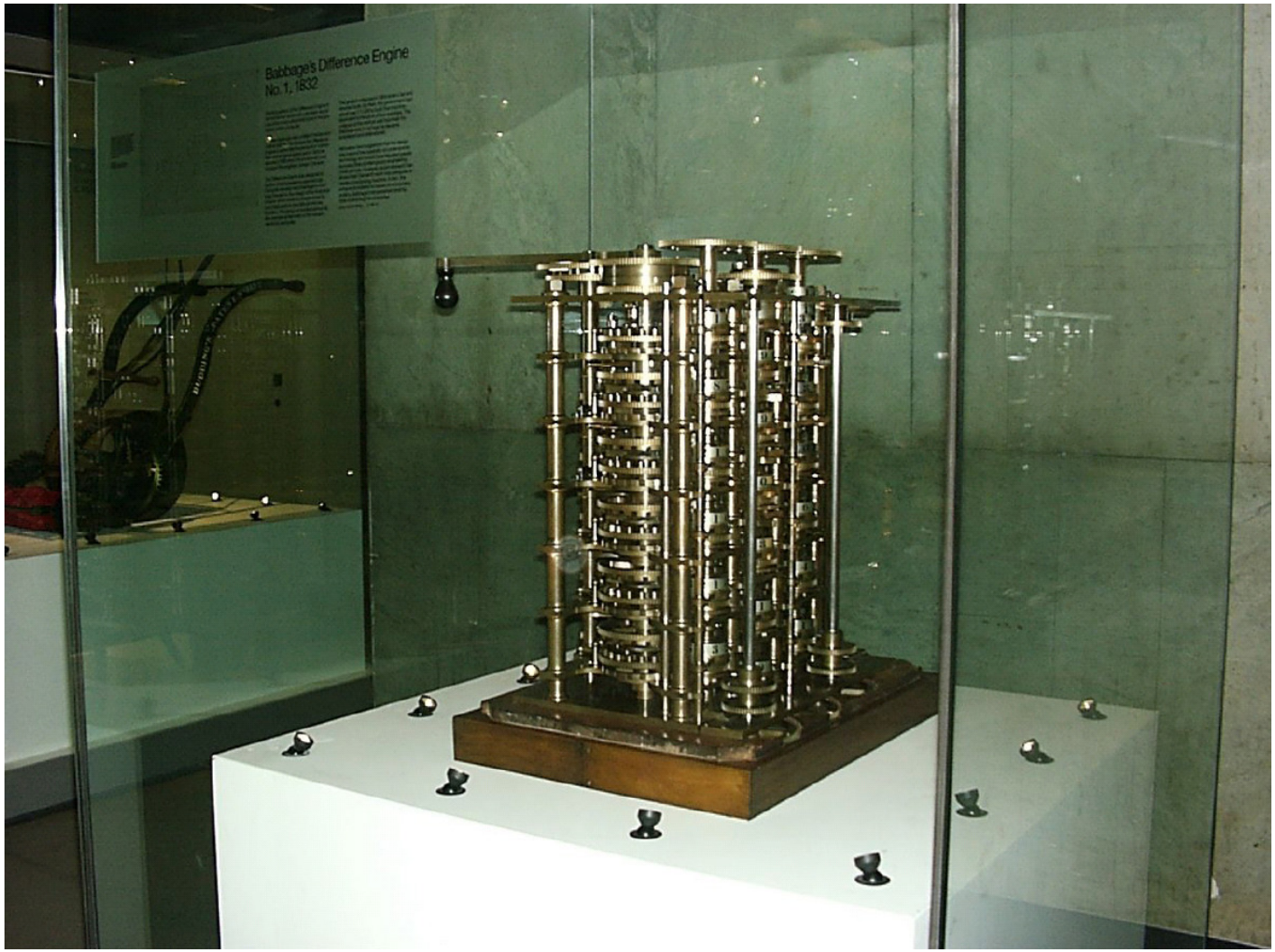


Figure 1.1: Reconstruction of Babbage's difference engine.

1848–1925 : G. Frege: Begriffsschrift

2-dimensional notation for PL1

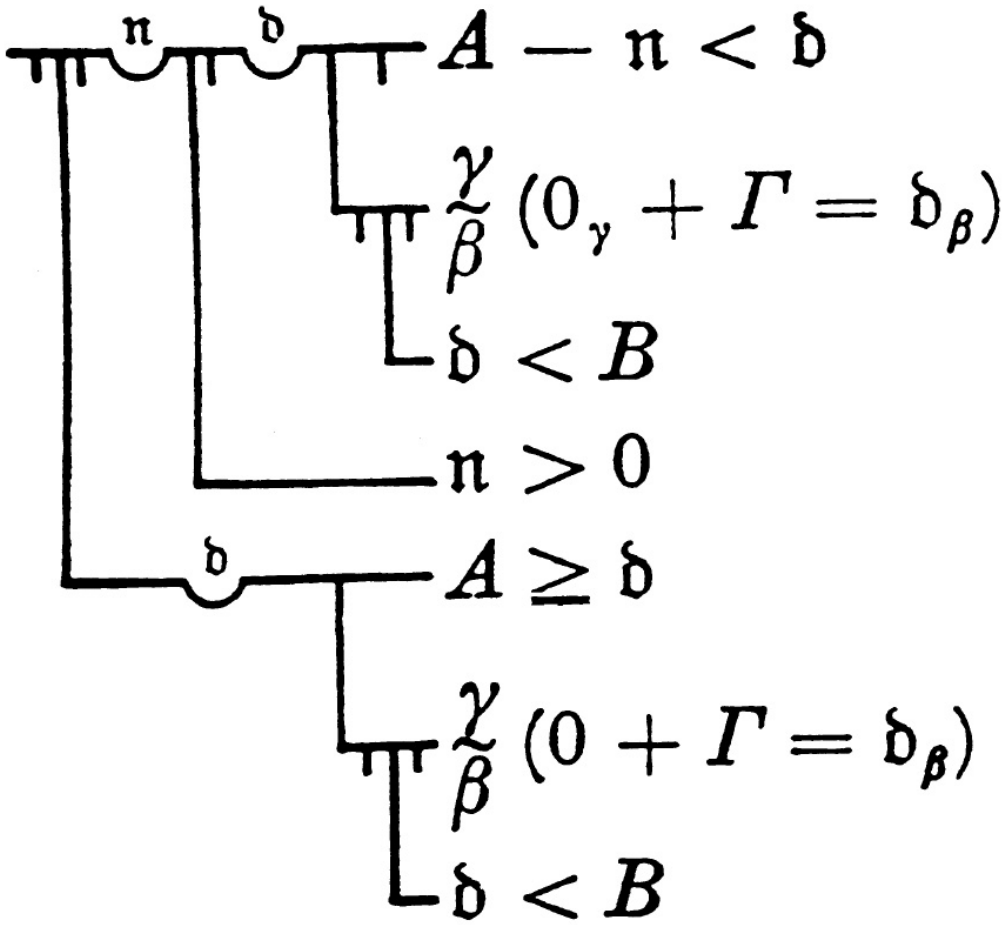


Figure 1.2: A formula from Frege's Begriffsschrift.

1862–1943: **D. Hilbert:**
Famous talk 1900 in Paris: 23 problems
23rd problem: The Entscheidungsproblem

1872–1970: **B. Russell:**
1910: Principia Mathematica
Logical positivism, *Vienna Circle* (1920–40)

1906–1978: **K. Gödel:**
Completeness theorem (1930)
Incompleteness theorem (1930/31)
Unprovability of theorems (1936)

1912–1954: **A. Turing:**
Turing-machine (1936)
Computability

1938/42: First operational programmable computer: Z 1
Z 3 by **K. Zuse** (Deutsches Museum)
with floating-point-arithmetic.
Plankalkül: First high-level programming language

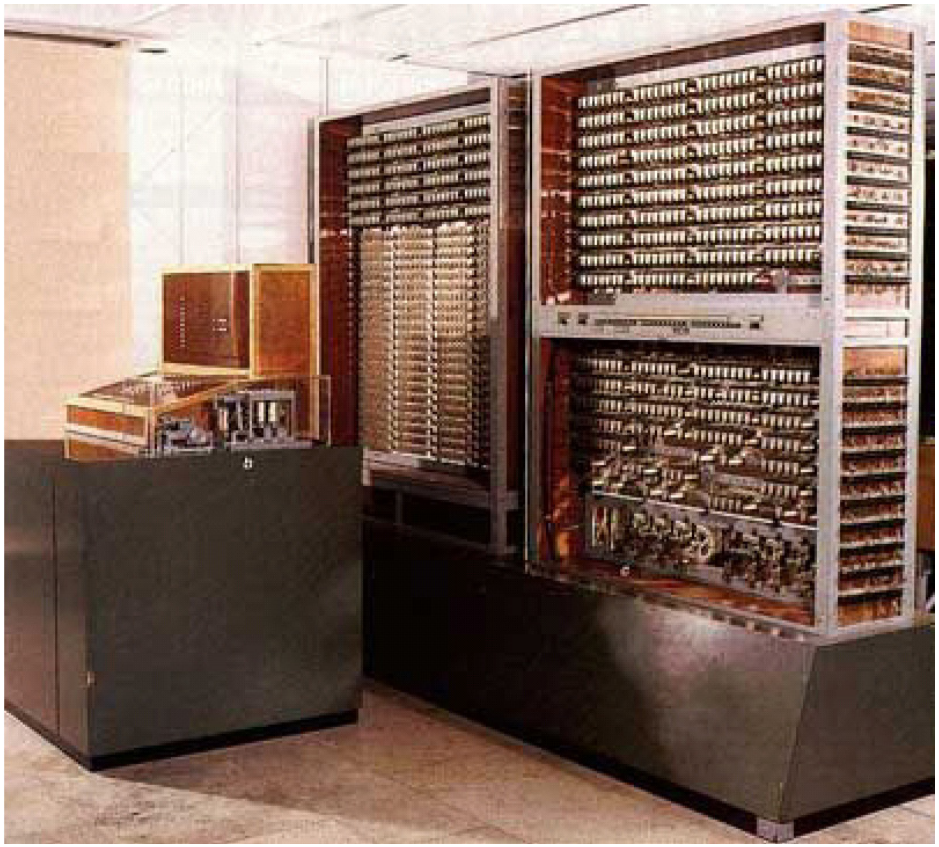


Figure 1.3: Reconstruction of Zuse's Z3.

1940 : First computer *"Heath Robinson"* built to decipher German messages (Turing)
1943 *"Collossus"* built from vacuum tubes

1940–45: **H. Aiken:** develops *MARK I, II, III*.
ENIAC
First general purpose electronic computer

1952 : IBM: *IBM 701*, first computer to yield profit (Rochester et alii)

1948: First stored program computer (*The Baby*)

Tom Kilburn (Manchester)

Manchester beats Cambridge by 3 months

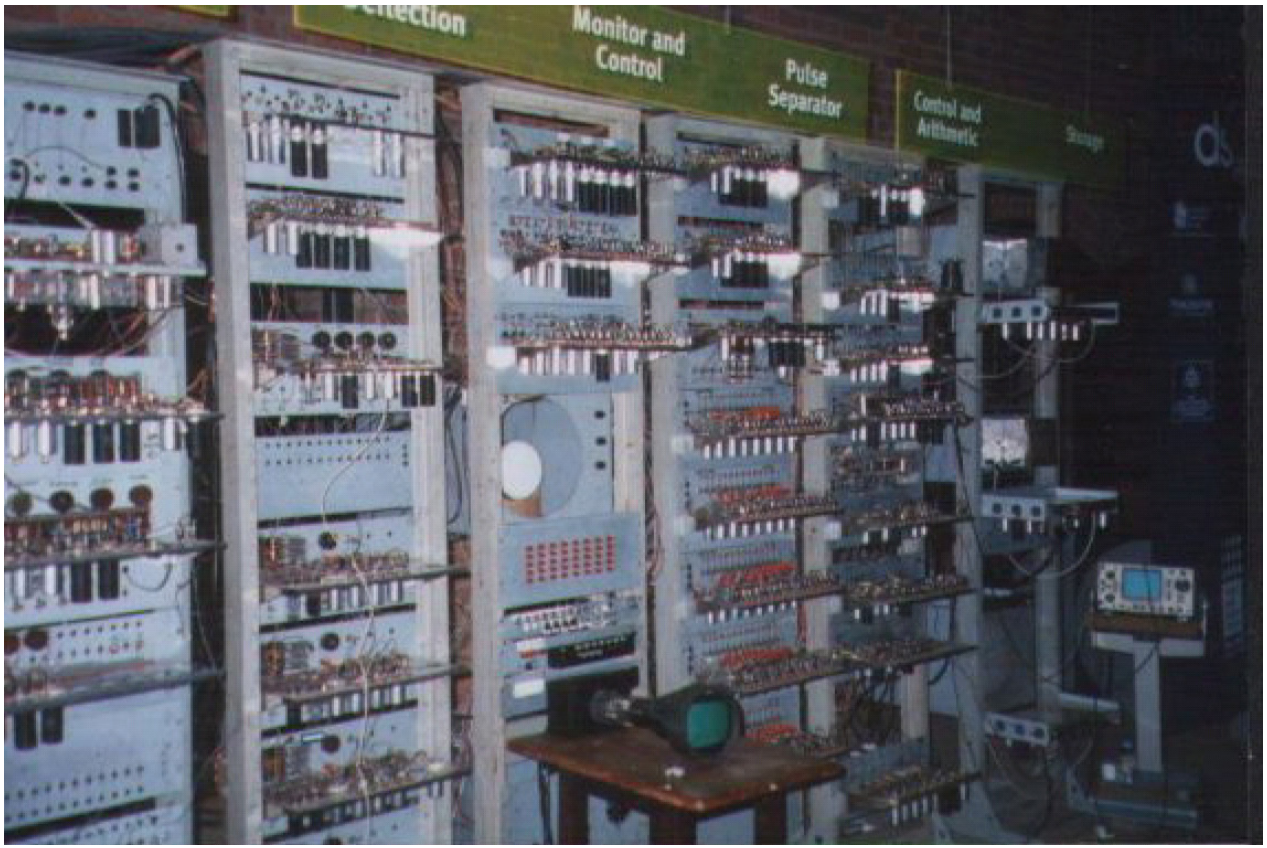


Figure 1.4: Reconstruction of Kilburn's baby.

First program run on The Baby in 1948:

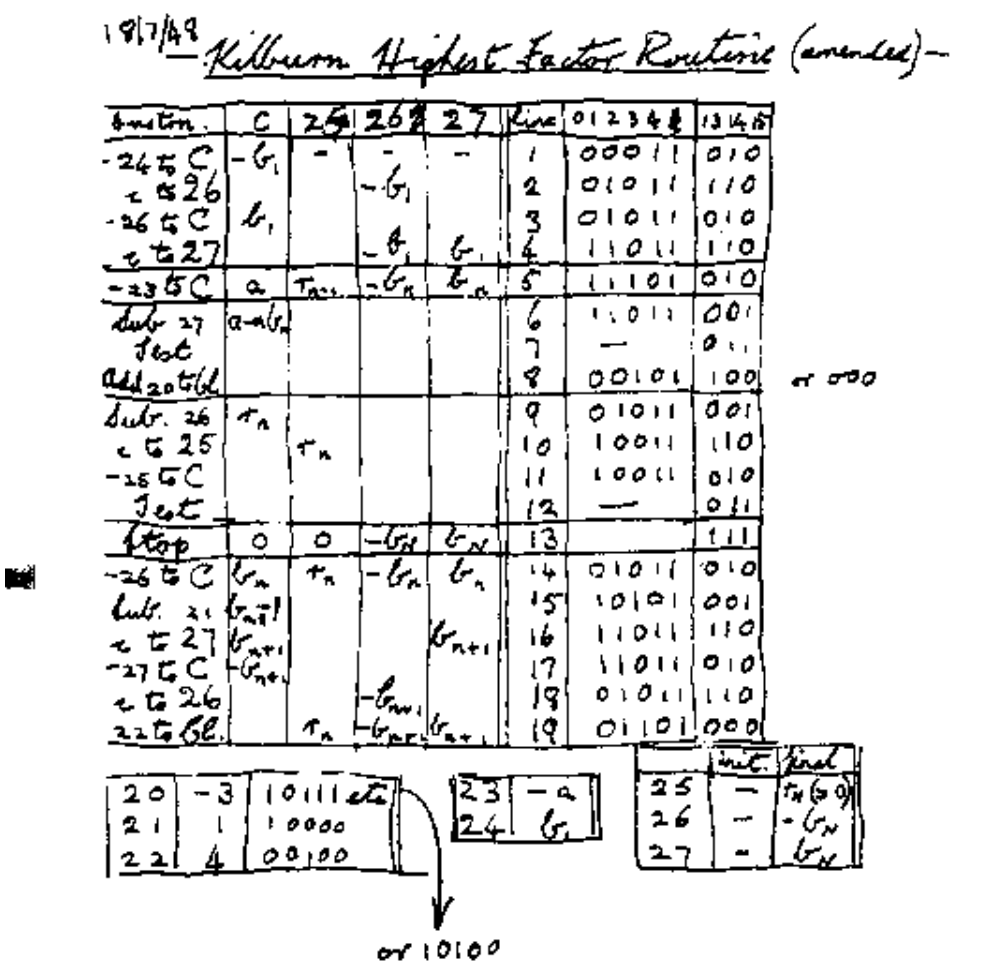


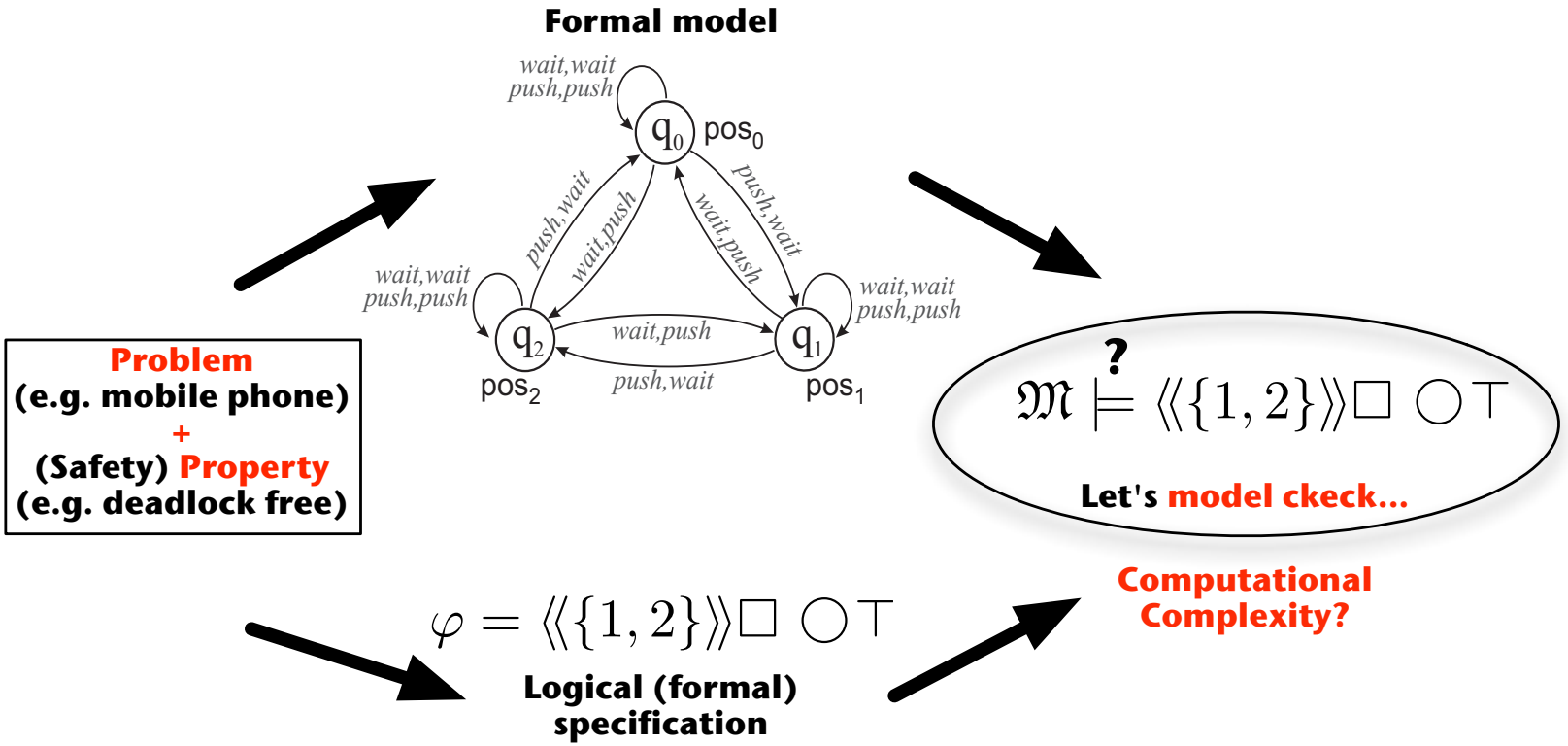
Figure 1.5: Reconstruction of first executed program on The Baby.

1.3 Verification

Model Checking Technique

Errors are expensive: Ariane 5 missile crash, . . .

Model checking provides means to **detect such erros**!



- **Model checking** refers to the problem to determine whether a given formula φ is satisfied in a state q of model $\mathcal{M}$.
- **Local model checking** is the decision problem that determines membership in the set

$$\mathsf{MC}(\mathcal{L}, \mathsf{MOD}, \models) =_{def}$$
$$\{(\mathcal{M}, q, \varphi) \in \mathsf{MOD} \times \mathcal{L} \mid \mathcal{M}, q \models \varphi\},$$

  where
  - $\mathcal{L}$ is a **logical language**,
  - MOD is a **class of (pointed) models** for $\mathcal{L}$ (i.e. a tuple consisting of a model and a state), and
  - $\models$ is a **semantic satisfaction relation** compatible with $\mathcal{L}$ and MOD.

Example 1.1 (Concurrency)

We assume the following three processes which run independently and parallel and share the integer variable x.

Inc	**while**	t	**do**	**if**	$x \leq 200$	**then**	$x := x + 1$	**fi**	**do**
Dec	**while**	t	**do**	**if**	$x \geq 0$	**then**	$x := x - 1$	**fi**	**do**
Reset	**while**	t	**do**	**if**	$x == 200$	**then**	$x := 0$	**fi**	**do**

Does this ensure that the value of x is always between (possibly including) 0 and 200?

Example 1.2 (Deadlock: Dining philosophers)

Five philosophers are sitting at a round table with a bowl of rice in the middle. They can just do three things: (1) thinking, (2) eating and (3) waiting to do (1) or (2). The problem is that they can eat only by using two chopsticks and eating rice out of the bowl.

But between two adjacent philosophers there is only one chopstick. Therefore, at any point in time, only two philosophers can eat concurrently.
If they all take the chopstick on their left, a deadlock occurs.

What is a fair synchronization so that there is no deadlock and no philosopher is starving?

How can this be modelled and the two properties formally verified?

Dining philosophers

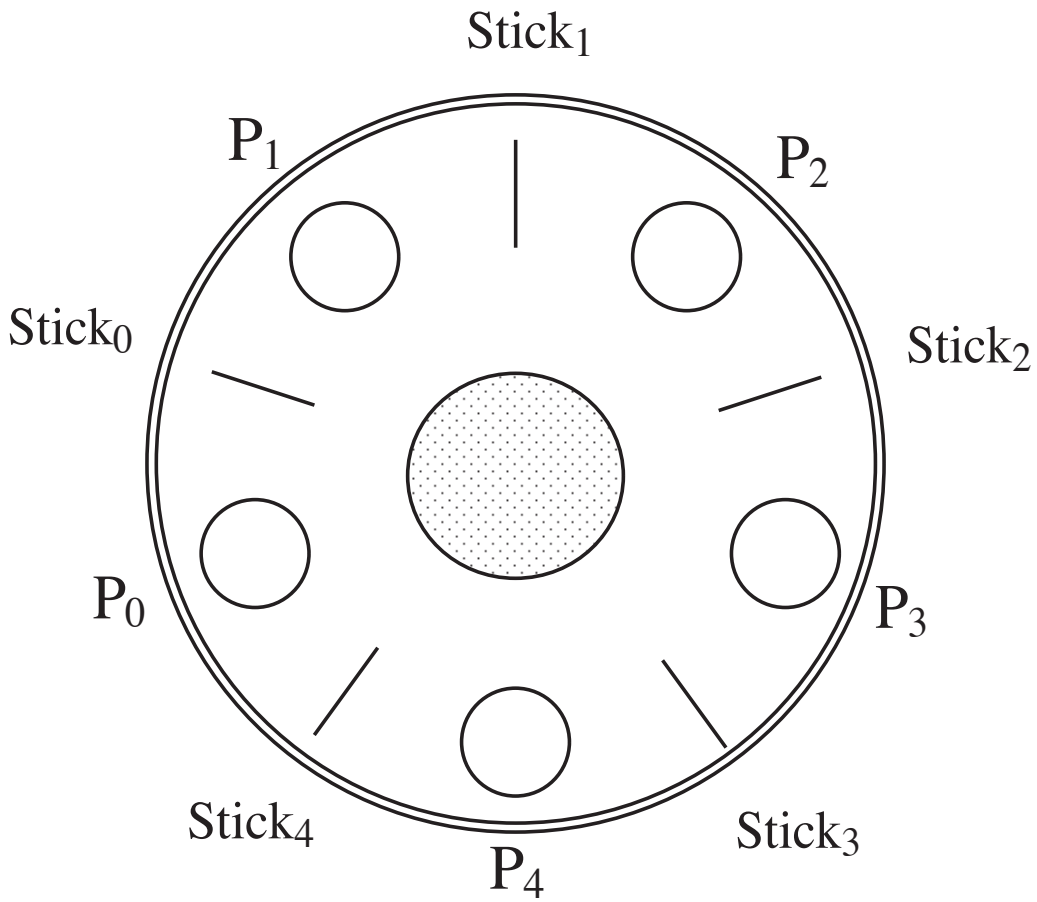


Figure 1.6: Dining philosophers.

Example 1.3 (What does program $\mathsf{P}(x)$ calculate?)

The input, x, ranges over integers.

```
y := 1;
z := 0;
while z ≠ x {
    z := z + 1;
    y := y · z
}
```

Example 1.4 (What do these programs calculate?)

```
int foo1(int n) {
local int k, int j;
k := 0;
j := 1;
while (k ≠ n){
    k := k + 1;
    j := 2 * j
} return(j) }
```

```
int foo2(int n) {
local int k, int j;
k := 0;
j := 1;
while (k ≠ n){
    k := k + 1;
    j := 2 + j
} return(j) }
```

1.4 References

- This lecture covers several areas, from classical logic, via classical verification techniques to the programming language **PROLOG**.
- There exist many good textbooks for all these areas.
- However, each has its own particular viewpoint and therefore introduces the notions in a different way. In particular concerning logic, many other, equivalent, approaches are possible.
- A theorem in our approach can be a definition in another and vice versa.
- **To solve the exercises, strictly stick to the exposition on the slides**, and what has been lectured until then.

Literature I

Christel Baier and Joost-Pieter Katoen.
Principles of model checking.
May 2008. Cambridge: MIT Press.

Rafael Bordini and Jürgen Dix.
Programming Multiagent Systems.
In G. Weiss, editor, *Multiagent systems*, pages 587–640, 2013. MIT-Press.

F. Dalpaiz and J. Dix and M. B. van Riemsdijk
Engineering Multi-Agent systems.
Second International Workshop (Paris 2014), revised and selected papers.
Springer LNAI series 8758, 2015.

Jürgen Dix and Michael Fisher.
Verifying Multi-Agent Systems.
In G. Weiss, editor, *Multiagent systems*, pages 641–693, 2013. MIT-Press.

Nils Bulling and Jürgen Dix and Wojciech Jamroga.
Model Checking Logics of Strategic Ability: Complexity.
In *Specification and Verification of Multi-Agent Systems*, pages 125–158, 2010. Springer.

Amir Pnueli.
The Temporal Logic of Programs.
In *Proceedings of the 18th IEEE Symp. on Foundations of Computer Science*, 1977.

A. Prasad Sistla and Edmund M. Clarke.
The complexity of Propositional Linear Temporal Logic.
In *Journal of the ACM*, volume32, no. 3, pages 733–749, 1985.

William Clocksin and Christopher S. Mellish.
Programming in PROLOG.
Springer Science & Business Media, 2003.

Michael Huth and Mark Ryan.
Logic in Computer Science: Modelling and reasoning about systems.
2000. Cambridge University Press.

D'Silva et al.
A Survey of Automated Techniques for Formal Software Verification.
In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, voulme 27, no. 7, pages 1165–1178, 2008.

2. Sentential Logic (SL)

Content of this chapter (1):

Logic: Logics can be used to describe the world and how it evolves. We start with propositional logic as the fundamental basis. All important notions (model, theory, validity, deducibility, derivability, calculus, model checking, counterexamples) are rigorously introduced and are the basis for first-order logic.

Examples: We illustrate the use of SL with three examples: The game of Sudoku, the Wumpus world and one of the weekly puzzles in the newspaper *Die Zeit*. Finding a solution for these problems is reduced to computing models of a certain theory.

Content of this chapter (2):

Calculi for SL: While it is nice to describe the world, we also want to draw conclusions about it. Therefore we have to derive new information and deduce statements, that are not explicitly given, but somehow contained in the description. We introduce a Hilbert-type calculus and the notion of proof in a purely syntactical way.

While Hilbert-type calculi are easily motivated, they cannot be efficiently implemented. Robinson's resolution calculus is much more suited and will be later extended to FOL.

2.1 Motivation

Folklore (1)

We all know the laws of Boole in algebra (**Boolean Algebra**)

$$
\begin{array}{rcl}
A \cap (B \cup C) & = & (A \cap B) \cup (A \cap C) \\
A \cup (B \cap C) & = & (A \cup B) \cap (A \cup C) \\
\overline{(A \cap B)} & = & \overline{A} \cup \overline{B} \\
\overline{(A \cup B)} & = & \overline{A} \cap \overline{B} \\
\overline{\overline{A}} & = & A \\
A \cup (B \cap A) & = & A \\
A \cap (B \cup A) & = & A \\
A \cup \overline{A} & = & U \\
A \cap \overline{A} & = & \emptyset \\
\overline{\emptyset} & = & U \\
\overline{U} & = & \emptyset
\end{array}
$$

Expressions formed using $\cup, \cap, \overline{\ }, (,), U, \emptyset$ are called **Boolean expressions** (U is the **universe**). $\rightsquigarrow$ **blackboard 2.1**

Folklore (2)

- Can we show, using these laws, that each Boolean expression (formed using $A, B, C, \ldots$ and $\cap, \cup, (,), \overline{\ }$) can be written as **an intersection of unions** of $A, \overline{A}, B, \overline{B}, C, \overline{C} \ldots$?
- Can we show, using these laws, that each Boolean expression (formed using $A, B, C, \ldots$ and $\cap, \cup, (,), \overline{\ }$) can be written as **a union of intersections** of $A, \overline{A}, B, \overline{B}, C, \overline{C} \ldots$?

$\rightsquigarrow$ **blackboard 2.2**

Folklore (3)

With the correspondence

$=$	corresponds to	$\leftrightarrow$
$\overline{}$	corresponds to	$\neg$
$\cap$	corresponds to	$\wedge$
$\cup$	corresponds to	$\vee$
$\emptyset$	corresponds to	false
U	corresponds to	true

we immediately get valid formulae in SL.

$\rightsquigarrow$ **blackboard 2.3**

Example 2.1 (Tweety and Friends)

We want to throw a party for Tweety, his friend Gentoo and Tux. But they have different circles of friends and dislike some. Tweety tells you that he would like to see either his friend the King or not to meet Gentoo's Adelie. But Gentoo proposes to invite Adelie or Humboldt or both. Tux, however, does not like Humboldt and the King too much, so he suggests to exclude at least one of them.

- Can we **represent** this using sentential logic?
- What do we *gain* by doing that?
- **Exactly how many** solutions are there?

Propositional Symbols

k means invite The King	$\neg k$ means exclude The King
a means invite Adelie	$\neg a$ means exclude Adelie
h means invite Humboldt	$\neg h$ means exclude Humboldt

Problem Formalized

Tweety: $k \vee \neg a$ means "invite The King or exclude Adelie", but not both: $\neg(k \wedge \neg a)$,

Gentoo: $a \vee h$ means "invite Adelie or Humboldt or both",

Tux: $\neg h \vee \neg k$ means "exclude Humboldt or The King or both".

Resulting Formula

Can we make the following formula true?

$$\varphi_{\text{party}}: \quad (k \vee \neg a) \wedge \neg(k \wedge \neg a) \wedge (a \vee h) \wedge (\neg h \vee \neg k)$$

There seems to be only one method of solution:
to check all possibilities.

$$\varphi_{\text{party}} : (k \vee \neg a) \wedge \neg(k \wedge \neg a) \wedge (a \vee h) \wedge (\neg h \vee \neg k)$$

k	a	h	$k \vee \neg a$	$\neg(k \wedge \neg a)$	$a \vee h$	$\neg h \vee \neg k$	φ_{party}
1	1	1	1	1	1	0	0
1	1	0	1	1	1	1	1
1	0	1	1	0	1	0	0
1	0	0	1	0	0	1	0
0	1	1	0	1	1	1	0
0	1	0	0	1	1	1	0
0	0	1	1	1	1	1	1
0	0	0	1	1	0	1	0

- Therefore there are exactly two solutions.
- We can also deduce, that **either Humboldt or both The King and Adelie can be invited**. Later we call the rows in the table models (or valuations).

Complexity

Truth tables have 2^n rows, where n is the number of propositional constants. In the worst case, all have to be checked (**or maybe not???????**).

Overview

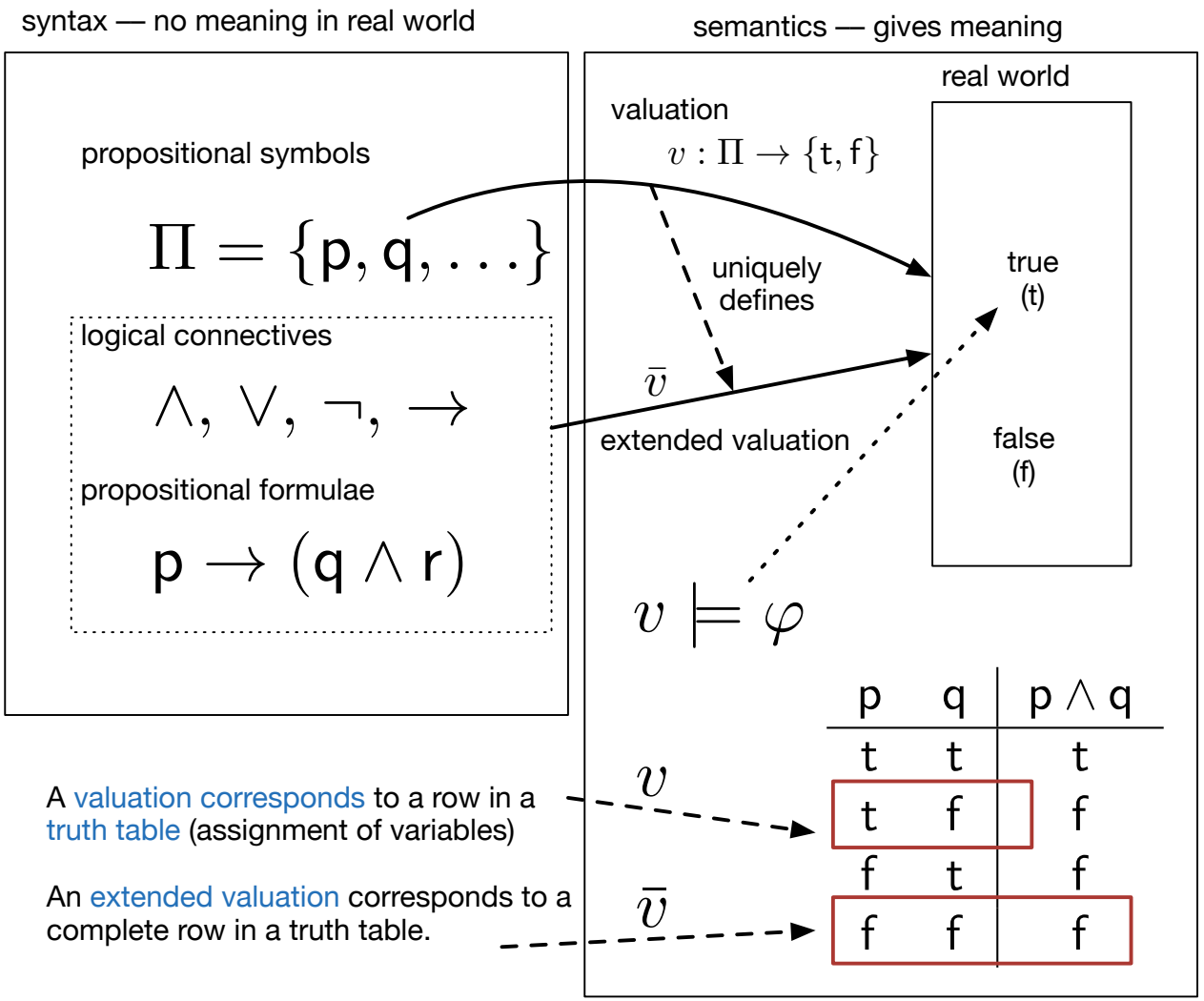


Figure 2.7: Syntax and Semantics for SL

2.2 Syntax and Semantics

Syntax: Language $\mathcal{L} = \mathcal{L}(\mathcal{P}rop)$

The sentential (propositional) language is built upon

propositional constants: $\Box$, $p_1, p_2, p_3, \ldots$ (countably many). We also use $a, b, c, \ldots, p, q, r, \ldots$. The symbol $\Box$ has a special meaning: it stands for **falsity**, the statement that is always false. This will become clear when we define the semantics of SL.

logical connectives: $\neg$ (unary), $\vee$ (binary), and

grouping symbols: $(,)$ for unique readability.

Often, for concrete applications, we consider only a finite, nonempty set of propositional constants and refer to it as $\mathcal{P}rop$ (e.g. $\mathcal{P}rop = \{a, p, q\}$). However, the symbol $\Box$ is always present, we do not include it in the set $\mathcal{P}rop$.

Logical connectives are used to construct complex formulae from the propositional constants.

Definition 2.2 (Sentential Language $\mathcal{L}(\mathcal{P}rop)$)

Given a set $\mathcal{P}rop$, the signature, the sentential (or propositional) language $\mathcal{L}(\mathcal{P}rop)$ over $\mathcal{P}rop$ determines the set $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ of $\mathcal{L}(\mathcal{P}rop)$ formulae defined by

$$\varphi ::= \Box \mid \mathsf{p} \mid (\neg\varphi) \mid (\varphi \vee \varphi)$$

where $\mathsf{p} \in \mathcal{P}rop$.

Note that this only makes sense for finite $\mathcal{P}rop$. However, by adding symbols 0 and 1, one can easily produce infinitely many propositional symbols p_{100101} using finitely many grammar rules.

$\rightsquigarrow$ **blackboard 2.4**

We assume that $\mathcal{P}rop$ is fixed and omit it if clear from context: so we write $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}}$ instead of $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ to denote the set of all SL-formulae over $\mathcal{P}rop$. We freely omit parentheses in formulae for better readability.

What about **other connectives**? They are defined as macros.

Definition 2.3 (SL connectives as macros)

We define the following syntactic constructs as macros:

$$\begin{aligned} \top &=_{def} (\neg\Box) \\ \varphi \wedge \psi &=_{def} (\neg((\neg\varphi) \vee (\neg\psi))) \\ \varphi \to \psi &=_{def} ((\neg\varphi) \vee \psi) \\ \varphi \leftrightarrow \psi &=_{def} ((\varphi \to \psi) \wedge (\psi \to \varphi)) \end{aligned}$$

$\rightsquigarrow$ **blackboard 2.5**

Semantics

What should it mean that a $\mathcal{L}(\mathcal{P}rop)$-formula φ is true? Intuitively, we want the following:

	$\top$		is always **t**
	$\square$		is always **f**
$\neg\varphi$ is **t**		iff	φ is **f**
$\varphi \vee \psi$ is **t**		iff	φ or ψ (or both) are **t**
$\varphi \wedge \psi$ is **t**		iff	both φ and ψ are **t**
$\varphi \rightarrow \psi$ is **t**		iff	either φ is **f** or ψ is **t**
$\varphi \leftrightarrow \psi$ is **t**		iff	φ and ψ are both **t**, or both **f**

- Transitions systems (see next chapter) consist of a set of states and actions (transitions) between these states.
- States are determined by what holds true in them.
- A state will be uniquely determined by the facts that are true in it.
- These facts are exactly the elements in $\mathcal{P}rop$: the **propositional constants**.
- Therefore, we need to fix the truth values of the propositional constants.

Definition 2.4 (Valuation, truth assignment)

A valuation (or truth assignment) for a language $\mathcal{L}(\mathcal{P}rop)$ is a mapping $v : \mathcal{P}rop \to \{\mathsf{t}, \mathsf{f}\}$ from the set of propositional constants into the set $\{\mathbf{t}, \mathbf{f}\}$.

A valuation fixes the values of individual propositional constants.

We are particularly interested in the truth of **complex formulae** like $(\mathsf{p} \vee \mathsf{q}) \wedge \mathsf{r}$.

Semantics

The process of mapping a set of $\mathcal{L}$-formulae into $\{\mathbf{t}, \mathbf{f}\}$ is called semantics.

It suffices to state the semantics for the basic connectives.

Definition 2.5 (Semantics $v \models \varphi$, $\bar{v}$)

Let v be a valuation. We define inductively the notion of a formula $\varphi \in \mathrm{Fml}^{\mathrm{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ being true or satisfied by v (notation: $v \models \varphi$):

$v \models \square$ does not hold,

$v \models \mathsf{p}$ *if, by definition,* $v(\mathsf{p}) = \mathbf{t}$ and $\mathsf{p} \in \mathcal{P}rop$,

$v \models \neg\varphi$ *if, by definition,* not $v \models \varphi$,

$v \models \varphi \vee \psi$ *if, by definition,* $v \models \varphi$ or $v \models \psi$

We denote a set of $\mathcal{L}(\mathcal{P}rop)$-formulae by T. Given a set $T \subseteq \mathrm{Fml}^{\mathrm{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ we write $v \models T$ *if, by definition,* $v \models \varphi$ for all $\varphi \in T$. We use $v \not\models \varphi$ instead of "not $v \models \varphi$".

Thus we have uniquely extended a valuation v to a mapping $\bar{v}$ from $\mathrm{Fml}^{\mathrm{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ into the set $\{\mathbf{t}, \mathbf{f}\}$.

$\rightsquigarrow$ **blackboard 2.6**

Truth Tables

Truth tables are a conceptually simple way of working with SL (invented by Peirce in 1893 and used by Wittgenstein in 1910'ies (and in TLP)).

p	q	$\neg p$	$p \vee q$	$p \wedge q$	$p \rightarrow q$	$p \leftrightarrow q$
t	**t**	**f**	**t**	**t**	**t**	**t**
f	**t**	**t**	**t**	**f**	**t**	**f**
t	**f**	**f**	**t**	**f**	**f**	**f**
f	**f**	**t**	**f**	**f**	**t**	**t**

Definition 2.6 (Model, Theory, Tautology (Valid))

1. If $v \models \varphi$, we also call v (or $\bar{v}$) a model of φ. We write $MOD(T)$ for all models of a theory T:

$$MOD(T) =_{def} \{v : v \models T\}$$

2. A theory is any set $T \subseteq \mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$.
   v satisfies T *if, by definition,* $v \models \varphi$ for all $\varphi \in T$.
3. A $\mathcal{L}$-formula φ is called $\mathcal{L}$-tautology (or simply called valid) if it is satisfied in all models: **for all models v it holds that $v \models \varphi$.**

We suppress the language $\mathcal{L}$ when obvious from context.

$\rightsquigarrow$ **blackboard 2.7**

Tweety revisited (1)

In Example 2.1 we can now state the following:

- The language we consider is $\mathcal{P}rop = \{\mathsf{a}, \mathsf{k}, \mathsf{h}\}$.
- The valuation v which makes $\mathsf{k}, \mathsf{a}, \mathsf{h}$ all true defines a structure which is not a model of the formula ϕ_{party}.
- The valuation which makes k, a true and h false, is a model of ϕ_{party}.
- ϕ_{party} can also be seen as the theory T_{party} consisting of the four formulae $\mathsf{k} \vee \neg\mathsf{a}$, $\neg(\mathsf{k} \wedge \neg\mathsf{a})$, $\mathsf{a} \vee \mathsf{h}$, and $\neg\mathsf{h} \vee \neg\mathsf{k}$.
- The formula ϕ_{party} is not a tautology, but $\mathsf{k} \to \mathsf{k}$ is one.

Definition 2.7 (Consequence Set $Cn(T)$, entailment)

A formula φ follows (semantically) from T (or is entailed from T) if for all models v of T (i.e. $v \models T$) also $v \models \varphi$ holds. We denote this by $T \models \varphi$.
We call

$$Cn_{\mathcal{L}}(T) =_{def} \{\varphi \in \mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)} : T \models \varphi\},$$

or simply $Cn(T)$, the semantic consequence operator.

We also say φ can be deduced from T.
$\rightsquigarrow$ **blackboard back to 2.7**

Duality of *MOD* and *Cn*

Both *Cn* and *MOD* are defined on *theories* (sets of formulae). But the definition of *Cn* can easily be extended to also deal with sets of models. For a set M consisting solely of models, we define

$Cn(M) =_{def} \{\varphi \in \mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)} : v \models \varphi \text{ for all } v \in M\}$.

MOD is obviously dual to *Cn*:

- $Cn(MOD(T)) = Cn(T)$,
- $MOD(Cn(T)) = MOD(T)$.

Tweety revisited (2)

Considering again Example 2.1 how does $Cn_{\mathcal{L}}(T_{\text{party}})$ look like?

- It is infinite.
- It contains all tautologies.
- It contains $(\mathsf{k} \wedge \mathsf{a} \wedge \neg \mathsf{h}) \vee \mathsf{h}$.
- Is $Cn_{\mathcal{L}}(T_{\text{party}}) = Cn_{\mathcal{L}}(\{(\mathsf{k} \wedge \mathsf{a} \wedge \neg \mathsf{h}) \vee \mathsf{h}\})$?
- Is $Cn_{\mathcal{L}}(T_{\text{party}}) = Cn_{\mathcal{L}}(\{(\mathsf{k} \wedge \mathsf{a} \wedge \neg \mathsf{h}) \vee (\neg \mathsf{k} \wedge \neg \mathsf{a} \wedge \mathsf{h})\})$?
- So there are different axiomatisations of $Cn_{\mathcal{L}}(T_{\text{party}})$. **How to decide which are equivalent?**

Duality

The duality principle is an important property in algebra and logic. It can be stated as follows.

Theorem 2.8 (Duality Principle)

We consider formulae ϕ, ψ over $\mathcal{P}rop$ built using only $\neg, \vee, \wedge, \Box, \top$. For such a formula φ, let $\mathsf{D}(\varphi)$ be the formula obtained by switching $\vee$ and $\wedge$, and $\Box$ and $\top$.

Then the following are equivalent:

- *ϕ is equivalent to ψ*
- *$D(\phi)$ is equivalent to $D(\psi)$.*

$\rightsquigarrow$ **blackboard 2.8**

Conjunctive/disjunctive Normal form

We consider formulae ϕ built over $\mathcal{P}rop$ from only $\neg, \vee, \wedge, \Box$.

Using the simple boolean rules, any formula ϕ can be written as a conjunction of disjunctions (conjunctive normal form)

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} \phi_{i,j}$$

The $\phi_{i,j}$ are just prop. constants or negated prop. constants from $\mathcal{P}rop$.

Definition 2.9 (Clauses, literals)

Constants and negated constants ($\neg$p) are called literals.
Disjunctions $\bigvee_{j=1}^{m_i} \phi_{i,j}$ built from literals $\phi_{i,j}$ are called clauses.

$\rightsquigarrow$ **blackboard 2.9**

Theorem 2.10 (Conjunctive/disjunctive normal form)

Any formula over $\mathcal{P}rop$ built using only $\neg, \vee, \wedge, \Box$ can be equivalently transformed into one with the same constants of the form

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} \phi_{i,j},$$

(the $\phi_{i,j}$ are, possibly negated, constants from $\mathcal{P}rop$, the empty disjunction is identified with $\Box$): conjunctive normal form.

Dually, any formula can be transformed into one of the form

$$\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} \phi_{i,j},$$

(the $\phi_{i,j}$ are, possibly negated, constants from $\mathcal{P}rop$, the empty conjunction is identified with $\top$): disjunctive normal form.

Normal forms

What are the normal forms of the following formulae?

- p,
- $\neg p$,
- $p \rightarrow p$,
- $\Box$,
- $\neg\neg(p \rightarrow \neg\Box)$.

Clauses

Normal form

Instead of working on arbitrary formulae, it is sometimes easier to work on finite sets of clauses.

This is without loss of generality (wlog): all formulae can be equivalently represented as a set (conjunction) of clauses.

This approach is much more suited for implementing a calculus (theorem proving) and we discuss it in detail in Subsection 2.5 from Slide 144 on.

Lemma 2.11 (Properties of $Cn(T)$)

The semantic consequence operator has the following properties:

1. T-expansion: $T \subseteq Cn(T)$,
2. Monotony: $T \subseteq T' \Rightarrow Cn(T) \subseteq Cn(T')$,
3. Closure: $Cn(Cn(T)) = Cn(T)$.

To entail (or deduce) something from a theory is important. But it is equally important to know that something is not entailed from a theory (i.e it does not follow from it): we emphasize this importance in the next lemma.

Lemma 2.12 ($\varphi \notin Cn$ (T), countermodel)

$\varphi \notin Cn(T)$ if and only if there is a model v with $v \models T$ and $v \models \neg\varphi$.

This model is often referred to as a countermodel for φ.

$\rightsquigarrow$ **blackboard 2.10**

Definition 2.13 (Completeness of a Theory T)

T is called complete if for each formula $\varphi \in \mathrm{Fml}^{\mathrm{SL}}_{\mathcal{L}(\mathcal{P}rop)}$: $T \models \varphi$ or $T \models \neg\varphi$ holds.

- Do not mix up this last condition with the property of a valuation (model) v: **each model is complete in the above sense**.
- A complete theory gives us a perfect description of the world (or state) we are in: we know everything!
- In most cases, however, our knowledge is incomplete.
- An incomplete theory leaves open many possibilities: many complete extensions of it, namely exactly the models of it.

Lemma 2.14 (Ex Falso Quodlibet)

The following are equivalent:

- T *has a model,*
- $Cn(T) \neq \mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$.

Russel story: Derive from "$7 = 8$*" that you are the pope.*

$\rightsquigarrow$ **blackboard 2.11**

Tweety revisited (3)

We discuss the following statements.

- Is the theory $Cn_{\mathcal{L}}(T_{\text{party}})$ complete, does it have a model?
- What about the theory $Cn_{\mathcal{L}}(\{A, A \to B, \neg B\})$?
- Does each theory with a model posess a maximal extension that still has a model?
- How many such extensions are there for T_{party}?
- Is there a theory T such that $Cn(T) = \emptyset$? What about $Cn(\emptyset)$?

Who killed Tuna, the cat?

Example 2.15 (Tuna the cat)

There are two people, Jack and Bill. There is also a cat, called Tuna, and a dog with no name, owned by Bill. Tuna has been killed by either Jack or Bill, but it is not known by whom precisely.
All we know is that animal lovers do not kill animals and that dog owners are animal lovers.
So who killed Tuna?

Who killed Tuna, the cat? (2)

How can we formalize this story in SL? We need to choose $\mathcal{P}rop$ appropriately.

- We need to express that Tuna is a cat: cat_{Tuna}.
- Either Jack or Bill killed Tuna: $\text{killer\_of\_Tuna}_{\text{Bill}}$, $\text{killer\_of\_Tuna}_{\text{Jack}}$.
- $\text{dog\_owner}_{\text{Bill}}$, $\text{dog\_owner}_{\text{Jack}}$.
- $\text{animal\_lover}_{\text{Bill}}$, $\text{animal\_lover}_{\text{Jack}}$.

What about $\text{dog\_owner}_{\text{Tuna}}$, cat_{Bill}?

Who killed Tuna, the cat? (3)

Our theory T consists of:

- cat_{Tuna}.
- $\text{killer\_of\_Tuna}_{\text{Bill}} \vee \text{killer\_of\_Tuna}_{\text{Jack}}$,
  $\neg(\text{killer\_of\_Tuna}_{\text{Bill}} \wedge \text{killer\_of\_Tuna}_{\text{Jack}})$
- $\text{dog\_owner}_{\text{Bill}}$.
- $\text{dog\_owner}_{\text{Bill}} \rightarrow \text{animal\_lover}_{\text{Bill}}$,
  $\text{dog\_owner}_{\text{Jack}} \rightarrow \text{animal\_lover}_{\text{Jack}}$. What about
  $\text{dog\_owner}_{\text{Tuna}} \rightarrow \text{animal\_lover}_{\text{Tuna}}$.
- $\text{animal\_lover}_{\text{Jack}} \rightarrow \neg\text{killer\_of\_Animals}_{\text{Jack}}$,
  $\text{animal\_lover}_{\text{Bill}} \rightarrow \neg\text{killer\_of\_Animals}_{\text{Bill}}$.
- $\text{animal}_{\text{Tuna}} \wedge (\neg\text{killer\_of\_Animals}_{\text{Bill}}) \rightarrow \neg\text{killer\_of\_Tuna}_{\text{Bill}}$,
  $\text{animal}_{\text{Tuna}} \wedge (\neg\text{killer\_of\_Animals}_{\text{Jack}}) \rightarrow \neg\text{killer\_of\_Tuna}_{\text{Jack}}$.

What is missing?

2.3 Some examples

Since some time, Sudoku puzzles are becoming quite famous.

			1	5		4	6	7
5	2	6		4				
4			9					
		4			5		2	6
6	9				7		1	
					1	8	3	9
	8							
	4	3	5	7	8	6		
			3				4	

Table 2: A simple Sudoku (S_1)

Can they be solved with sentential logic?

Idea: Given a Sudoku-Puzzle S, construct a language $\mathcal{P}rop_{\text{Sudoku}}$ and a theory $T_S \subseteq \text{Fml}^{\text{SL}}_{\mathcal{L}(}\mathcal{P}rop_{\text{Sudoku}})$ such that

$$MOD(T_S) = \text{ Solutions of the puzzle S}$$

Solution

In fact, we construct a theory T_{Sudoku} and for each (partial) instance of a 9×9 puzzle S a particular theory T_S such that

$$MOD(T_{\text{Sudoku}} \cup T_S) = \{\mathcal{S} : \mathcal{S} \text{ is a solution of S}\}$$

We introduce the following prop. constants:

1. $\text{eins}_{i,j}$, $1 \leq i, j \leq 9$,
2. $\text{zwei}_{i,j}$, $1 \leq i, j \leq 9$,
3. $\text{drei}_{i,j}$, $1 \leq i, j \leq 9$,
4. $\text{vier}_{i,j}$, $1 \leq i, j \leq 9$,
5. $\text{fuenf}_{i,j}$, $1 \leq i, j \leq 9$,
6. $\text{sechs}_{i,j}$, $1 \leq i, j \leq 9$,
7. $\text{sieben}_{i,j}$, $1 \leq i, j \leq 9$,
8. $\text{acht}_{i,j}$, $1 \leq i, j \leq 9$,
9. $\text{neun}_{i,j}$, $1 \leq i, j \leq 9$.

This completes the language $\mathcal{P}rop_{\text{Sudoku}}$.

How many symbols are these?

We distinguished between the puzzle S and a solution $\mathcal{S}$ of it.

What is a model (or valuation) in the sense of Definition 2.6 (Slide 70)?

			1	5		4	6	7
5	2	6		4				
4			9					
		4			5		2	6
6	9				7		1	
					1	8	3	9
	8							
	4	3	5	7	8	6		
			3				4	

Table 3: How to construct a model $\mathcal{S}$?

We have to give our symbols a meaning (**the semantics**), i.e. a valuation v.

$$\text{eins}_{i,j} \textbf{ means } \langle i,j \rangle \text{ contains a } 1$$
$$\text{zwei}_{i,j} \textbf{ means } \langle i,j \rangle \text{ contains a } 2$$
$$\vdots$$
$$\text{neun}_{i,j} \textbf{ means } \langle i,j \rangle \text{ contains a } 9$$

To be precise: given a 9×9 square that is completely filled out, we define our valuation v as follows (for all $1 \leq i,j \leq 9$).

$$v(\text{eins}_{i,j}) = \begin{cases} \text{true, if } 1 \text{ is at position } \langle i,j \rangle, \\ \text{false, else .} \end{cases}$$

$$v(\text{zwei}_{i,j}) = \begin{cases} \text{true, if } 2 \text{ is at position } \langle i,j \rangle, \\ \text{false, else .} \end{cases}$$

$$v(\text{drei}_{i,j}) = \begin{cases} \text{true, if } 3 \text{ is at position } \langle i,j \rangle, \\ \text{false, else .} \end{cases}$$

$$v(\text{vier}_{i,j}) = \begin{cases} \text{true, if } 4 \text{ is at position } \langle i,j \rangle, \\ \text{false, else .} \end{cases}$$

etc.

$$v(\text{neun}_{i,j}) = \begin{cases} \text{true, if } 9 \text{ is at position } \langle i,j \rangle, \\ \text{false, else .} \end{cases}$$

Therefore any 9×9 square can be seen as a model or valuation with respect to the language $\mathcal{L}_{\text{Sudoku}}$.

How does T_S look like?

$$
\begin{aligned}
T_S = \{ \quad & \mathsf{eins}_{1,4}, \mathsf{eins}_{5,8}, \mathsf{eins}_{6,6}, \\
& \mathsf{zwei}_{2,2}, \mathsf{zwei}_{4,8}, \\
& \mathsf{drei}_{6,8}, \mathsf{drei}_{8,3}, \mathsf{drei}_{9,4}, \\
& \mathsf{vier}_{1,7}, \mathsf{vier}_{2,5}, \mathsf{vier}_{3,1}, \mathsf{vier}_{4,3}, \mathsf{vier}_{8,2}, \mathsf{vier}_{9,8}, \\
& \vdots \\
& \mathsf{neun}_{3,4}, \mathsf{neun}_{5,2}, \mathsf{neun}_{6,9}, \\
\} &
\end{aligned}
$$

How should the theory T_{Sudoku} look like (s.t. models of $T_{\text{Sudoku}} \cup T_S$ correspond to solutions of the puzzle)?

First square: T_1

1. $\text{eins}_{1,1} \vee \ldots \vee \text{eins}_{3,3}$
2. $\text{zwei}_{1,1} \vee \ldots \vee \text{zwei}_{3,3}$
3. $\text{drei}_{1,1} \vee \ldots \vee \text{drei}_{3,3}$
4. $\text{vier}_{1,1} \vee \ldots \vee \text{vier}_{3,3}$
5. $\text{fuenf}_{1,1} \vee \ldots \vee \text{fuenf}_{3,3}$
6. $\text{sechs}_{1,1} \vee \ldots \vee \text{sechs}_{3,3}$
7. $\text{sieben}_{1,1} \vee \ldots \vee \text{sieben}_{3,3}$
8. $\text{acht}_{1,1} \vee \ldots \vee \text{acht}_{3,3}$
9. $\text{neun}_{1,1} \vee \ldots \vee \text{neun}_{3,3}$

The formulae on the last slide are saying, that

1. The number 1 must appear somewhere in the first square.
2. The number 2 must appear somewhere in the first square.
3. The number 3 must appear somewhere in the first square.
4. etc

Does that mean, that each number $1, \ldots, 9$ occurs exactly once in the first square?

No! We have to say, that each number occurs only once:

T_1':

1. $\neg(\text{eins}_{i,j} \wedge \text{zwei}_{i,j})$, $1 \leq i, j \leq 3$,
2. $\neg(\text{eins}_{i,j} \wedge \text{drei}_{i,j})$, $1 \leq i, j \leq 3$,
3. $\neg(\text{eins}_{i,j} \wedge \text{vier}_{i,j})$, $1 \leq i, j \leq 3$,
4. etc
5. $\neg(\text{zwei}_{i,j} \wedge \text{drei}_{i,j})$, $1 \leq i, j \leq 3$,
6. $\neg(\text{zwei}_{i,j} \wedge \text{vier}_{i,j})$, $1 \leq i, j \leq 3$,
7. $\neg(\text{zwei}_{i,j} \wedge \text{fuenf}_{i,j})$, $1 \leq i, j \leq 3$,
8. etc

How many formulae are these?

Second square: T_2

1. $\text{eins}_{1,4} \vee \ldots \vee \text{eins}_{3,6}$
2. $\text{zwei}_{1,4} \vee \ldots \vee \text{zwei}_{3,6}$
3. $\text{drei}_{1,4} \vee \ldots \vee \text{drei}_{3,6}$
4. $\text{vier}_{1,4} \vee \ldots \vee \text{vier}_{3,6}$
5. $\text{fuenf}_{1,4} \vee \ldots \vee \text{fuenf}_{3,6}$
6. $\text{sechs}_{1,4} \vee \ldots \vee \text{sechs}_{3,6}$
7. $\text{sieben}_{1,4} \vee \ldots \vee \text{sieben}_{3,6}$
8. $\text{acht}_{1,4} \vee \ldots \vee \text{acht}_{3,6}$
9. $\text{neun}_{1,4} \vee \ldots \vee \text{neun}_{3,6}$

And all the other formulae from the previous slides (adapted to this case): T_2'

The same has to be done for all 9 squares.

What is still missing:

Rows: Each row should contain exactly the numbers from 1 to 9 (no number twice).

Columns: Each column should contain exactly the numbers from 1 to 9 (no number twice).

First Row: $T_{\text{Row 1}}$

1. $\text{eins}_{1,1} \vee \text{eins}_{1,2} \vee \ldots \vee \text{eins}_{1,9}$
2. $\text{zwei}_{1,1} \vee \text{zwei}_{1,2} \vee \ldots \vee \text{zwei}_{1,9}$
3. $\text{drei}_{1,1} \vee \text{drei}_{1,2} \vee \ldots \vee \text{drei}_{1,9}$
4. $\text{vier}_{1,1} \vee \text{vier}_{1,2} \vee \ldots \vee \text{vier}_{1,9}$
5. $\text{fuenf}_{1,1} \vee \text{fuenf}_{1,2} \vee \ldots \vee \text{fuenf}_{1,9}$
6. $\text{sechs}_{1,1} \vee \text{sechs}_{1,2} \vee \ldots \vee \text{sechs}_{1,9}$
7. $\text{sieben}_{1,1} \vee \text{sieben}_{1,2} \vee \ldots \vee \text{sieben}_{1,9}$
8. $\text{acht}_{1,1} \vee \text{acht}_{1,2} \vee \ldots \vee \text{acht}_{1,9}$
9. $\text{neun}_{1,1} \vee \text{neun}_{1,2} \vee \ldots \vee \text{neun}_{1,9}$

Analogously for all other rows, eg.

Ninth Row: $T_{\text{Row 9}}$

1. $\text{eins}_{9,1} \vee \text{eins}_{9,2} \vee \ldots \vee \text{eins}_{9,9}$
2. $\text{zwei}_{9,1} \vee \text{zwei}_{9,2} \vee \ldots \vee \text{zwei}_{9,9}$
3. $\text{drei}_{9,1} \vee \text{drei}_{9,2} \vee \ldots \vee \text{drei}_{9,9}$
4. $\text{vier}_{9,1} \vee \text{vier}_{9,2} \vee \ldots \vee \text{vier}_{9,9}$
5. $\text{fuenf}_{9,1} \vee \text{fuenf}_{9,2} \vee \ldots \vee \text{fuenf}_{9,9}$
6. $\text{sechs}_{9,1} \vee \text{sechs}_{9,2} \vee \ldots \vee \text{sechs}_{9,9}$
7. $\text{sieben}_{9,1} \vee \text{sieben}_{9,2} \vee \ldots \vee \text{sieben}_{9,9}$
8. $\text{acht}_{9,1} \vee \text{acht}_{9,2} \vee \ldots \vee \text{acht}_{9,9}$
9. $\text{neun}_{9,1} \vee \text{neun}_{9,2} \vee \ldots \vee \text{neun}_{9,9}$

Is that sufficient? What if a row contains several 1's?

First Column: $T_{\text{Column 1}}$

1. $\text{eins}_{1,1} \vee \text{eins}_{2,1} \vee \ldots \vee \text{eins}_{9,1}$
2. $\text{zwei}_{1,1} \vee \text{zwei}_{2,1} \vee \ldots \vee \text{zwei}_{9,1}$
3. $\text{drei}_{1,1} \vee \text{drei}_{2,1} \vee \ldots \vee \text{drei}_{9,1}$
4. $\text{vier}_{1,1} \vee \text{vier}_{2,1} \vee \ldots \vee \text{vier}_{9,1}$
5. $\text{fuenf}_{1,1} \vee \text{fuenf}_{2,1} \vee \ldots \vee \text{fuenf}_{9,1}$
6. $\text{sechs}_{1,1} \vee \text{sechs}_{2,1} \vee \ldots \vee \text{sechs}_{9,1}$
7. $\text{sieben}_{1,1} \vee \text{sieben}_{2,1} \vee \ldots \vee \text{sieben}_{9,1}$
8. $\text{acht}_{1,1} \vee \text{acht}_{2,1} \vee \ldots \vee \text{acht}_{9,1}$
9. $\text{neun}_{1,1} \vee \text{neun}_{2,1} \vee \ldots \vee \text{neun}_{9,1}$

Analogously for all other columns.

Is that sufficient? What if a column contains several 1's?

All put together:

$$\begin{aligned} T_{\text{Sudoku}} = \; & T_1 \cup T_1' \cup \ldots \cup T_9 \cup T_9' \\ & T_{\text{Row 1}} \cup \ldots \cup T_{\text{Row 9}} \\ & T_{\text{Column 1}} \cup \ldots \cup T_{\text{Column 9}} \end{aligned}$$

Here is a more difficult one.

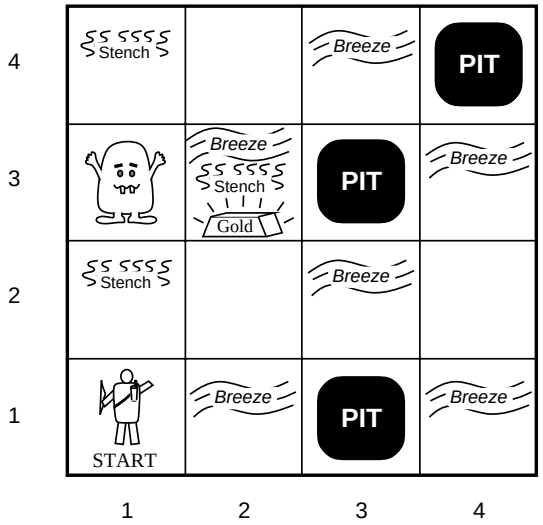
2		3		9				6
	1							4
	9				5			
		7						
6			9				7	
8	4							
9	3		1		7	5		
7		2			9	4		
1			2				6	

Table 4: A difficult Sudoku $S_{\text{difficult}}$

Example 2.16 (Wumpus)

A wumpus is moving around in a grid. A knight in shining armour moves around and is looking for gold. The wumpus is trying to kill him when they are on the same cell. The knight is also dying when he enters a pit. Fortunately, in the cells adjacent to pits there is a cold breeze. And in adjacent cells to the wumpus, there is an incredible stench.

How should the knight behave?



1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 **OK**	2,2	3,2	4,2
1,1 **A** **OK**	2,1 **OK**	3,1	4,1

(a)

A = *Agent*
B = *Breeze*
G = *Glitter, Gold*
OK = *Safe square*
P = *Pit*
S = *Stench*
V = *Visited*
W = *Wumpus*

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 **OK**	2,2 **P?**	3,2	4,2
1,1 **V** **OK**	2,1 **A** **B** **OK**	3,1 **P?**	4,1

(b)

1,4	2,4	3,4	4,4
1,3 **W!**	2,3	3,3	4,3
1,2 **A** **S** **OK**	2,2 **OK**	3,2	4,2
1,1 **V** **OK**	2,1 **B** **V** **OK**	3,1 **P!**	4,1

(a)

A = *Agent*
B = *Breeze*
G = *Glitter, Gold*
OK = *Safe square*
P = *Pit*
S = *Stench*
V = *Visited*
W = *Wumpus*

1,4	2,4 **P?**	3,4	4,4
1,3 **W!**	2,3 **A** **S** **G** **B**	3,3 **P?**	4,3
1,2 **S** **V** **OK**	2,2 **V** **OK**	3,2	4,2
1,1 **V** **OK**	2,1 **B** **V** **OK**	3,1 **P!**	4,1

(b)

Language definition:

$\mathsf{s}_{i,j}$	stench on field $\langle i, j \rangle$
$\mathsf{b}_{i,j}$	breeze on field $\langle i, j \rangle$
$\mathsf{pit}_{i,j}$	$\langle i, j \rangle$ is a pit
$\mathsf{gl}_{i,j}$	$\langle i, j \rangle$ glitters
$\mathsf{w}_{i,j}$	$\langle i, j \rangle$ contains Wumpus

General knowledge:

$$\neg \mathsf{s}_{1,1} \longrightarrow (\neg \mathsf{w}_{1,1} \wedge \neg \mathsf{w}_{1,2} \wedge \neg \mathsf{w}_{2,1})$$
$$\neg \mathsf{s}_{2,1} \longrightarrow (\neg \mathsf{w}_{1,1} \wedge \neg \mathsf{w}_{2,1} \wedge \neg \mathsf{w}_{2,2} \wedge \neg \mathsf{w}_{3,1})$$
$$\neg \mathsf{s}_{1,2} \longrightarrow (\neg \mathsf{w}_{1,1} \wedge \neg \mathsf{w}_{1,2} \wedge \neg \mathsf{w}_{2,2} \wedge \neg \mathsf{w}_{1,3})$$

$$\mathsf{s}_{1,2} \longrightarrow (\mathsf{w}_{1,3} \vee \mathsf{w}_{1,2} \vee \mathsf{w}_{2,2} \vee \mathsf{w}_{1,1})$$

Knowledge after the 3rd move:

$$\neg s_{1,1} \wedge \neg s_{2,1} \wedge s_{1,2} \wedge \neg b_{1,1} \wedge b_{2,1} \wedge \neg b_{1,2}$$

Question:

Can we derive that the wumpus is located at $\langle 1, 3 \rangle$?

Answer:

Yes. With any correct and complete calculus.
Because it is semantically entailed.

We now formalize a ”Logelei“ (in order to solve it with a **theorem prover**).

Example 2.17 (”Logelei“ from ”Die Zeit“ (1))

Alfred ist als neuer Korrespondent in Wongowongo. Er soll über die Präsidentschaftswahlen berichten, weiß aber noch nichts über die beiden Kandidaten, weswegen er sich unter die Leute begibt, um Infos zu sammeln. Er befragt eine Gruppe von Passanten, von denen drei Anhänger der Entweder-oder-Partei sind und drei Anhänger der Konsequenten.

”Logelei“ from ”Die Zeit“ (2)

Auf seinem Notizzettel notiert er stichwortartig die Antworten.
A: »Nachname Songo: Stadt Rongo«,
B: »Entweder-oder-Partei: älter«,
C: »Vorname Dongo: bei Umfrage hinten«,
A: »Konsequenten: Vorname Mongo«,
B: »Stamm Bongo: Nachname Gongo«,
C: »Vorname Dongo: jünger«,
D: »Stamm Bongo: bei Umfrage vorn«,
E: »Vorname Mongo: bei Umfrage hinten«,
F: »Konsequenten: Stamm Nongo«,
D: »Stadt Longo: jünger«,
E: »Stamm Nongo: jünger«.
F: »Konsequenten: Nachname Gongo«.

”Logelei“ from ”Die Zeit“ (3)

Jetzt grübelt Alfred. Er weiß, dass die Anhänger der Entweder-oder-Partei (A, B und C) immer eine richtige und eine falsche Aussage machen, während die Anhänger der Konsequenten (D, E und F) entweder nur wahre Aussagen oder nur falsche Aussagen machen.
Welche Informationen hat Alfred über die beiden Kandidaten?

(By Zweistein)

Towards a solution

- Selection of the language $\mathcal{L}$ ($\mathcal{P}rop$).
- Analysis and formalization of the problem.
- Transformation to the input format of a prover.
- Output of a solution, i.e. a **model**.

Definition of the constants

$\text{sur}_{x,\text{Songo}} \equiv$	x's surname is Songo
$\text{sur}_{x,\text{Gongo}} \equiv$	x's surname is Gongo
$\text{first}_{x,\text{Dongo}} \equiv$	x's first name is Dongo
$\text{first}_{x,\text{Mongo}} \equiv$	x's first name is Mongo
$\text{tribe}_{x,\text{Bongo}} \equiv$	x belongs to the Bongos
$\text{tribe}_{x,\text{Nongo}} \equiv$	x belongs to the Nongos
$\text{city}_{x,\text{Rongo}} \equiv$	x comes from Rongo
$\text{city}_{x,\text{Longo}} \equiv$	x comes from Longo
$\text{senior}_x \equiv$	x is the senior candidate
$\text{junior}_x \equiv$	x is the junior candidate
$\text{worse}_x \equiv$	x's poll is worse
$\text{better}_x \equiv$	x's poll is better

Here x is a candidate, i.e. $x \in \{a, b\}$. So we have 24 constants in total.

The correspondent Alfred noted 12 statements about the candidates (each interviewee gave 2 statements, ϕ, ϕ') which we enumerate as follows

$$\phi_A, \phi'_A, \phi_B, \phi'_B, \ldots, \phi_F, \phi'_F,$$

All necessary symbols are now defined, and we can formalize the given statements.

Formalization of the statements

$$
\begin{aligned}
\phi_A &\leftrightarrow (\mathsf{sur}_{a,\mathsf{Songo}} \wedge \mathsf{city}_{a,\mathsf{Rongo}}) \vee \\
&\quad\ (\mathsf{sur}_{b,\mathsf{Songo}} \wedge \mathsf{city}_{b,\mathsf{Rongo}}) \\
\phi'_A &\leftrightarrow \mathsf{first}_{b,\mathsf{Mongo}} \\
\phi_B &\leftrightarrow \mathsf{senior}_a \\
\phi'_B &\leftrightarrow (\mathsf{tribe}_{a,\mathsf{Bongo}} \wedge \mathsf{sur}_{a,\mathsf{Gongo}}) \vee \\
&\quad\ (\mathsf{tribe}_{b,\mathsf{Bongo}} \wedge \mathsf{sur}_{b,\mathsf{Gongo}}) \\
&\ \ \vdots
\end{aligned}
$$

Furthermore, **explicit** conditions between the statements are given, e.g.

$$(\phi_A \wedge \neg\phi'_A) \vee (\neg\phi_A \wedge \phi'_A)$$

and

$$(\phi_D \wedge \phi'_D) \vee (\neg\phi_D \wedge \neg\phi'_D).$$

Analogously, for the other statements.

Is this enough information to solve the puzzle?

E.g., can the following formula be satisfied?

$$\mathsf{sur}_{a,\mathsf{Songo}} \wedge \mathsf{sur}_{a,\mathsf{Gongo}}$$

We also need **implicit** conditions (**axioms**) which are required to solve this problem.
It is necessary to state that each candidate has only one name, comes from one city, etc.

We need the following background knowledge...

$$\text{sur}_{x,\text{Songo}} \leftrightarrow \neg\text{sur}_{x,\text{Gongo}}$$
$$\text{first}_{x,\text{Dongo}} \leftrightarrow \neg\text{first}_{x,\text{Mongo}}$$
$$\vdots$$
$$\text{worse}_x \leftrightarrow \neg\text{better}_x$$

Can we abstain from these axioms by changing our representation of the puzzle?

What is still missing?

Can we prove that when a's poll is worse, then s's poll is better?

We need to state the relationships between these attributes:

$$\text{worse}_x \leftrightarrow \text{better}_y$$

$$\text{senior}_x \leftrightarrow \text{junior}_y$$

Finally, we have modeled all "sensible" information. Does this yield a unique model?

No! There are **6 models** in total, but this is all right. It just means there is no unique solution.

What if a unique model is desirable?

Often, there are additional assumptions hidden "between the lines". Think, for example, of deductions by Sherlock Holmes (or Miss Marple, Spock, Monk etc).

For example, it might be sensible to assume that both candidates come from **different** cities:

$$\text{city}_{x,\text{Rongo}} \leftrightarrow \text{city}_{y,\text{Longo}}$$

Indeed, with this additional axiom there is an unique model.

But, be careful...

... this additional information may not be justified by the nature of the task!

2.4 Hilbert Calculus

Deriving formulae purely algorithmically

- We have a clear undestanding of what it means that "a formula follows or can be deduced from a theory T".
- Can we **automize** that?
- Can we find a procedure to systematically derive new formulae?
- In such a way that all formulae that do indeed follow from T will be eventually derived?
- Let us take inspiration of what mathematicians have done since centuries!

Definition 2.18 (Hilbert-Type Calculi)

A Hilbert-Type calculus over a language $\mathcal{L}$ is a pair $\langle \mathrm{Ax}, \mathrm{Inf} \rangle$ where

Ax: is a subset of the set of $\mathcal{L}$-formulae: they are called axioms,

Inf: is a set of pairs written in the form

$$\frac{\phi_1, \phi_2, \ldots, \phi_n}{\psi}$$

where $\phi_1, \phi_2, \ldots, \phi_n, \psi$ are from $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$: they are called inference rules.

Intuitively, we assume the axioms to be true and use the inference rules to derive new formulae.

Definition 2.19 (Calculus for Propositional Logic SL)

We define $\mathrm{Hilbert}_{\mathcal{L}}^{SL} = \langle \{\neg\varphi \vee \varphi, \neg\Box\}, \mathtt{Inf} \rangle$, a Hilbert-Type calculus for SL. The only axiom schema is $\neg\varphi \vee \varphi$, the only axiom is $\neg\Box$.
The set $\mathtt{Inf}$ of inference rules consists of the following four schemata

Expansion: $\frac{\varphi}{\psi \vee \varphi}$,

Associativity: $\frac{\varphi \vee (\psi \vee \chi)}{(\varphi \vee \psi) \vee \chi}$,

Shortening: $\frac{\varphi \vee \varphi}{\varphi}$,

Cut: $\frac{\varphi \vee \psi, \neg\varphi \vee \chi}{\psi \vee \chi}$.

(φ, ψ, χ stand for arbitrarily complex formulae (not just constants). They represent schemata, rather than particular formulae in the language.)

Definition 2.20 (Proof)

A **proof** of a formula φ from a theory $T \subseteq \mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$ is a finite **sequence** $\varphi_1, \ldots, \varphi_n$ of formulae such that $\varphi_n = \varphi$ and for all i with $1 \leq i \leq n$ one of the following conditions holds:

- φ_i is **instance of the axiom schema**, or of the form $\neg\Box$,
- $\varphi_i \in T$,
- there is φ_l with $l < i$ and φ_i is obtained from φ_l by **expansion**, **associativity**, or **shortening**,
- there is φ_l, φ_k with $l, k < i$ and φ_i is obtained from φ_l, φ_k by **cut**.

We write: $T \vdash_{\mathsf{SL}} \varphi$ (φ can be **derived (or proved)** from T).

We have now introduced two important notions:

Syntactic derivability $\vdash_{SL}$: the notion that certain formulae can be derived (or proved) from other formulae using a certain calculus,

Semantic validity $\models$: the notion that certain formulae follow (semantically) (or can be deduced (are entailed)) from other formulae based on the semantic notion of a model.

Definition 2.21 (Correct-, Completeness for a calculus)

Given an arbitrary **calculus** (which defines a notion $\vdash$) and a **semantics** based on certain models (which defines a relation $\models$), we say that

Correctness: The calculus is **correct** with respect to the semantics, if the following holds:

$$T \vdash \phi \text{ implies } T \models \phi.$$

Completeness: The calculus is **complete** with respect to the semantics, if the following holds:

$$T \models \phi \text{ implies } T \vdash \phi.$$

Lemma 2.22 (Corrrectness of $\mathrm{Hilbert}_{\mathcal{L}}^{SL}$)

Let T be a (possibly infinite) theory and φ a formula over $\mathcal{P}rop$.

Then $T \vdash_{SL} \varphi$ implies that $T \models \varphi$.

I.e. each provable formula is also entailed!

Proof.

By induction on the structure of φ.
We have to show that

1. **all instances of the axiom schema are valid**, $\neg\Box$ is valid, and
2. **for each inference rule the conjunction of the premises entails its conclusion**.

□

We show a few simple facts that we need later to prove completeness.

Lemma 2.23 (Some derivations)

1. $\vdash_{\mathsf{SL}} \neg\Box$.
2. $\phi \vee \psi \vdash_{\mathsf{SL}} \psi \vee \phi$.
3. $\varphi \vdash_{\mathsf{SL}} \varphi \vee \psi$.
4. $\phi \vee \psi \vdash_{\mathsf{SL}} \neg\neg\phi \vee \psi$.
5. $\neg\neg\phi \vee \psi \vdash_{\mathsf{SL}} \phi \vee \psi$.
6. $\{\neg\phi \vee \chi, \neg\psi \vee \chi\} \vdash_{\mathsf{SL}} \neg(\phi \vee \psi) \vee \chi$.

The first three are shown on $\rightsquigarrow$ **blackboard 2.12**. (4) and (5) are $\rightsquigarrow$ **exercise**.

Lemma 2.24 (Derived inference rules)

The following rules can be added to the calculus without affecting the set of derivable formulae. They are also called derived rules.

1. **(Modus Ponens)** $\frac{\chi, \chi \to \varphi}{\varphi}$,
2. **(Commutativity)** $\frac{\psi \vee \varphi}{\varphi \vee \psi}$,
3. *Let* $i_1, i_2, \ldots, i_m \in \{1, 2, \ldots, n\}$. *Then*

   (Gen. Expansion) $\frac{\varphi_{i_1} \vee \ldots \vee \varphi_{i_m}}{\varphi_1 \vee \ldots \vee \varphi_n}$

Theorem 2.25 (Weak completeness)

Let $\varphi_1, \ldots, \varphi_n$ and φ formulae over $\mathcal{P}rop$.Then:

$\{\varphi_1, \ldots, \varphi_n\} \models \varphi$ *implies that* $\{\varphi_1, \ldots, \varphi_n\} \vdash_{SL} \varphi$.

Definition 2.26 ((In-) Consistency of a theory T)

A theory T is called **consistent**, if there is a formula that can not be proved from it.

A theory T is called **inconsistent**, if it is not consistent. I.e. from an inconsistent theory, **all** formulae can be proved.

Consistency is a property of a **calculus**. It is often mixed up with **existence of a model** (which it is often equivalent to).

Attention: Completeness

After we have proved the completeness theorem, we know that **inconsistent theories are exactly those that do not posess any models** (see Slide 83). But we do not know this yet! It has to be proved.

The key to prove completeness of our calculus is based on the following

Theorem 2.27 (Deduction Theorem)

Let T be a theory and ϕ, ψ be formulae from $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$. Then the following holds

$$T \vdash_{\mathsf{SL}} (\phi \to \psi) \text{ if and only if } T \cup \{\phi\} \vdash_{\mathsf{SL}} \psi$$

Proof.

One direction (left to right) is trivial: just an application of Modus Ponens (which we have shown to be a derived rule on Slide 132). The other direction is by **induction on the length of a proof for ψ**.

A proof of ψ from $T \cup \{\phi\}$ might have used ϕ at several places. The idea is to replace in each step of the proof of ψ a formula η by $\phi \to \eta$. This then transforms the old proof into one of $(\phi \to \psi)$ in T. □

An important consequence of the deduction theorem is the following

Lemma 2.28 (Consistency and Derivability)

Let T be a theory and ϕ a formula from $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(Prop)}$ (both could be inconsistent). Then the following holds

1. $T \vdash_{\mathsf{SL}} \phi$ *if and only if* $T \cup \{\neg\phi\}$ *is inconsistent.*
2. $T \nvdash_{\mathsf{SL}} \phi$ *if and only if* $T \cup \{\neg\phi\}$ *is consistent.*

Proof of Lemma 2.28.

The second statement follows trivially from the first (contraposition).

Let $T \vdash_{\mathsf{SL}} \phi$. Adding $\neg\phi$ does not influence the proof of ϕ, so $T \cup \{\neg\phi\} \vdash_{\mathsf{SL}} \phi$. But, trivially, $T \cup \{\neg\phi\} \vdash_{\mathsf{SL}} \neg\phi$, so $T \cup \{\neg\phi\}$ is inconsistent. Conversely, if $T \cup \{\neg\phi\}$ is inconsistent, then $T \cup \{\neg\phi\} \vdash_{\mathsf{SL}} \phi$ (because any formula can be derived). By the deduction theorem, $T \vdash_{\mathsf{SL}} \neg\phi \to \phi$, thus $T \vdash_{\mathsf{SL}} \phi$. □

Theorem 2.29 (Correct-, Completeness for $\mathrm{Hilbert}_{\mathcal{L}}^{SL}$)

A formula follows semantically from a theory T if and only if it can be derived:

$$T \models \varphi \text{ if and only if } T \vdash_{\mathsf{SL}} \varphi$$

Proof of the Correctness part of Theorem 2.29.

Correctness follows by **induction on the length of proofs**: the axiom is valid and all four inference rules have the property, that from their premises, the conclusions follow. We have already discussed this in Lemma 2.22.

□

Proof of the Completeness part of Theorem 2.29.

To prove completeness, it is enough to show that **each consistent theory is satisfiable** (because if $T \models \varphi$ and not $T \vdash_{\mathsf{SL}} \varphi$, then $T \cup \{\neg\varphi\}$ is **consistent** (why?), thus satisfiable: a **contradiction**). Given a consistent theory T, how to construct a model for T? We claim that the **models** of T are exactly the **maximal consistent** extensions of T. Let $\varphi_0, \varphi_1, \ldots$ an enumeration of $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$. We construct the sequence T_i as follows: $T_0 := T$,

$$T_{i+1} := \begin{cases} T_i & \text{if } T_i \vdash_{\mathsf{SL}} \varphi_i, \\ T_i \cup \{\neg\varphi_i\} & \text{else.} \end{cases}$$

Then $\bigcup_{i=0}^{\infty} T_i$ is a maximal consistent extension of T:If it were inconsistent, then there were a proof of $\Box$; this proof uses only finitely many formulae; but then there is a n_0 such that T_{n_0} contains all of them, so T_{n_0} were inconsistent; but all T_i are consistent by Lemma 2.28! It is maximal because $\varphi_0, \varphi_1, \ldots$ is an enumeration of $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$. $\Box$

The following important result is equivalent to the completeness theorem:

Corollary 2.30 (Compactness for $\mathrm{Hilbert}_{\mathcal{L}}^{SL}$)

A formula **follows from a theory** T *if and only if it* **follows from a finite subset of** T*:*

$$Cn(T) = \bigcup \{Cn(T') : T' \subseteq T, T' \text{finite}\}.$$

Proof.

Derivability in a calculus means that **there is a proof**, a finite object. Thus, by completeness, if a formula follows from a (perhaps infinite) theory T, there is a finite proof of it which **involves only finitely many formulae** from T. Thus the formula follows from these finitely many formulae. □

Satisfiability of a theory

The following equally important result is again a corollary to the completeness theorem

Corollary 2.31 (Compactness for theories T)

Let T be a theory from $\mathsf{Fml}^{\mathsf{SL}}_{\mathcal{L}(\mathcal{P}rop)}$. Then the following are equivalent:

- *T is satisfiable,*
- *each finite subset of T is satisfiable.*

2.5 Resolution Calculus

Conjunctive Normal Form

- We have already seen that each formula can be written in conjunctive normal form.
- Thus each formula can be identified with a set of clauses.
- We note that it is possible, that a clause is empty: it is represented by $\square$.
- We also note that it is possible, that a conjunction is empty. The empty conjunction is obviously dual to the empty disjunction: it is represented by $\neg\square$ (or by the macro $\top$ introduced in Definition 2.3 on Slide 64).

Language for the Resolution Calculus

How can we determine whether a disjunction $\bigvee_{j=1}^{m_i} \phi_{i,j}$ is satisfiable or not?

Definition 2.32 (Clauses, Language $\mathsf{Fml}_{\mathcal{L}}^{clausal}$)

A propositional formula of the form $\bigvee_{j=1}^{m_i} \phi_{i,j}$, where all $\phi_{i,j}$ are constants or negated constants, is called a **clause**.

We also allow the **empty clause**, represented by $\square$. Dually, $\neg\square$ can be interpreted as the **empty conjunction**.

Given a signature $\mathcal{P}rop$, determining a language $\mathcal{L}(\mathcal{P}rop)$ or just $\mathcal{L}$, we denote by $\mathsf{Fml}_{\mathcal{L}(\mathcal{P}rop)}^{clausal}$, **or just** $\mathsf{Fml}_{\mathcal{L}}^{clausal}$ the set of formulae consisting of just **clauses** over $\mathcal{P}rop$.

Definition 2.33 (Set-notation of clauses)

A clause $A \vee \neg B \vee C \vee \ldots \vee \neg E$ can also be represented as a set

$$\{A, \neg B, C, \ldots, \neg E\}.$$

Thus the **set-theoretic union** of such sets corresponds again to a clause: $\{A, \neg B\} \cup \{A, \neg C\}$ represents $A \vee \neg B \vee \neg C$. The empty set $\emptyset$ corresponds to the empty disjunction and is represented by $\square$.

But we know this already from Slides 52–54.
Note that we identify **double occurrences** implicitly by using the set-notation: $\{A, A, B, C\} = \{A, B, C\}$.

We define an inference rule on $\mathsf{Fml}_{\mathcal{L}}^{clausal}$:

Definition 2.34 (SL resolution)

Let C_1, C_2 be clauses and let X by any constant from $\mathcal{P}rop$. The following inference rule allows to derive the clause $C_1 \vee C_2$ from $C_1 \vee \mathsf{X}$ and $C_2 \vee \neg\mathsf{X}$:

$$\textbf{(Res)}\ \frac{C_1 \vee \mathsf{X},\ C_2 \vee \neg\mathsf{X}}{C_1 \vee C_2}$$

If $C_1 = C_2 = \emptyset$, then $C_1 \vee C_2 = \square$.

Compare with the **Cut** rule in Definition 2.19.

If we use the set-notation for clauses, we can formulate the inference rule as follows:

Definition 2.35 (SL resolution (Set notation))

Derive the clause $C_1 \cup C_2$ from $C_1 \cup \{\mathsf{X}\}$ and $C_2 \cup \{\neg\mathsf{X}\}$:

$$\textbf{(Res)}\ \frac{C_1 \cup \{\mathsf{X}\},\ C_2 \cup \{\neg\mathsf{X}\}}{C_1 \cup C_2}$$

In the rule C_1 or C_2 can be both empty, in which case $\square$ is derived from X and $\neg$X. (Note, that we identify the empty set $\emptyset$ with $\square$.)

Definition 2.36 (Resolution Calculus for SL)

We define the **resolution calculus** $\text{Robinson}^{SL}_{\mathcal{L}^{clausal}}$ **as the pair** $\langle \emptyset, \{\text{Res}\} \rangle$ operating on the set $\text{Fml}^{clausal}_{\mathcal{L}}$ of clauses.
We denote the **corresponding derivation relation** by $\vdash_{\text{Res}}$ (in contrast to $\vdash_{\text{SL}}$ induced by the Hilbert calculus from Definition 2.19).

- So there are no axioms at all and only one inference rule.
- The notion of a **proof** in this system is obvious: it is literally the same as given in Definition 2.19.

Question:

Is this calculus correct and complete?

Answer:

It is correct, but not complete!

But "$T \models \phi$" is equivalent to

$$\text{"}T \cup \{\neg\phi\} \text{ is unsatisfiable"}$$

or rather to

$$T \cup \{\neg\phi\} \vdash_{\mathsf{SL}} \square.$$

Now in such a case, the **resolution calculus is powerful enough** to derive $\square$.

Resolution calculus is not complete

Example 2.37 (Complete vs refutation complete)

We consider $\mathcal{P}rop = \{a, b\}$. Obviously $a \models_{SL} a \vee b$ and $a \vdash_{SL} a \vee b$. But a $\nvdash_{RES}$ a $\vee$ b.

The reason is that with a alone, **there is no possibility to derive** a $\vee$ b with the resolution rule (the premises are not fulfilled). Whereas when $\neg$(a $\vee$ b) is added, i.e. both $\neg$a and $\neg$b, then the rules can be applied and lead to the derivation of $\square$.

- If $T \vdash_{\mathsf{SL}} \varphi$ does not imply $T \vdash_{\mathsf{Res}} \varphi$.
- If $T \vdash_{\mathsf{Res}} \varphi$ then also $T \vdash_{\mathsf{SL}} \varphi$ (for $\varphi \in \mathsf{Fml}_{\mathcal{L}}^{clausal}$).
- But the following holds:

$$T \cup \{\neg\phi\} \vdash_{\mathsf{SL}} \square \textit{ if and only if } T \cup \{\neg\phi\} \vdash_{\mathsf{Res}} \square$$

We say that resolution is refutation complete.

Theorem 2.38 (Completeness of resolution refutation)

If M is an unsatisfiable set of clauses then the empty clause $\square$ can be derived in $\mathrm{Robinson}^{SL}_{\mathcal{L}^{clausal}}$.

How to use the resolution calculus? (1)

Suppose we want to prove that $T \models \phi$. Using the Hilbert calculus, we could try to prove ϕ directly. This is not possible for the resolution calculus, for two different reasons:

- it **operates on clauses**, not on arbitrary formulae,
- there is **no completeness result** in a form similar to Theorem 2.29.

How to use the resolution calculus? (2)

But we have refutation completeness (see Theorem 2.38) of the resolution calculus, which allows us to do the following:

1. We transform $T \cup \{\neg\phi\}$ into a set of clauses.
2. Then we apply the resolution calculus and try to derive $\square$. I.e. instead of $T \vdash_{\text{res}} \phi$ we try to show $T \cup \{\neg\phi\} \vdash_{\text{res}} \square$.
3. If we succeed, we have shown $T \models \phi$.
4. If we can show that the empty clause is not derivable at all, then $T \not\models \phi$.

How to use the resolution calculus? (3)

How to show that the empty clause is **not** derivable from a theory T?

- One could try to formally prove that no such derivation is possible in the resolution calculus.
- This could be done by a clever induction on the structure of all possible derivations.
- But this is often very complicated and far from trivial.
- Just applying the calculus and not being able to derive the empty clause is not enough (there might be other ways).

The best way is to argue semantically: to show that there is a model of $T \cup \{\neg\phi\}$ by giving a concrete valuation.

3. Verification I: LT properties

Content of this chapter (1):

In this chapter we explore ways to use SL for the verification of reactive systems. First we need to model concurrent systems using SL. This allows us to analyse a broad class of hardware and software systems.

Transition Systems: They are the main underlying abstraction to describe concurrent systems. We start with **asynchronous systems** and then deal with **synchronization**: interleaving, and handshaking. An important classic example that we discuss is the dining philosophers problem.

LT properties: Which properties of the concurrent systems that we introduced do we want to verify? It turns out that many interesting conditions are linear temporal properties. Among them are fairness, safety as well as starvation and deadlock-freeness.

Content of this chapter (2):

LTL: This is an extension of SL to formulate linear-temporal properties within a logic. Given a model and a **LTL** formula, checking whether the formula is satisfied in the model is called **LTL model checking**. We describe and discuss this method in detail.

CTL, timed CTL: These are nontrivial extensions of **LTL** and allow us to express much more interesting properties. However, these are too advanced and will not be dealt with in this course.

3.1 Motivation

The need for verification

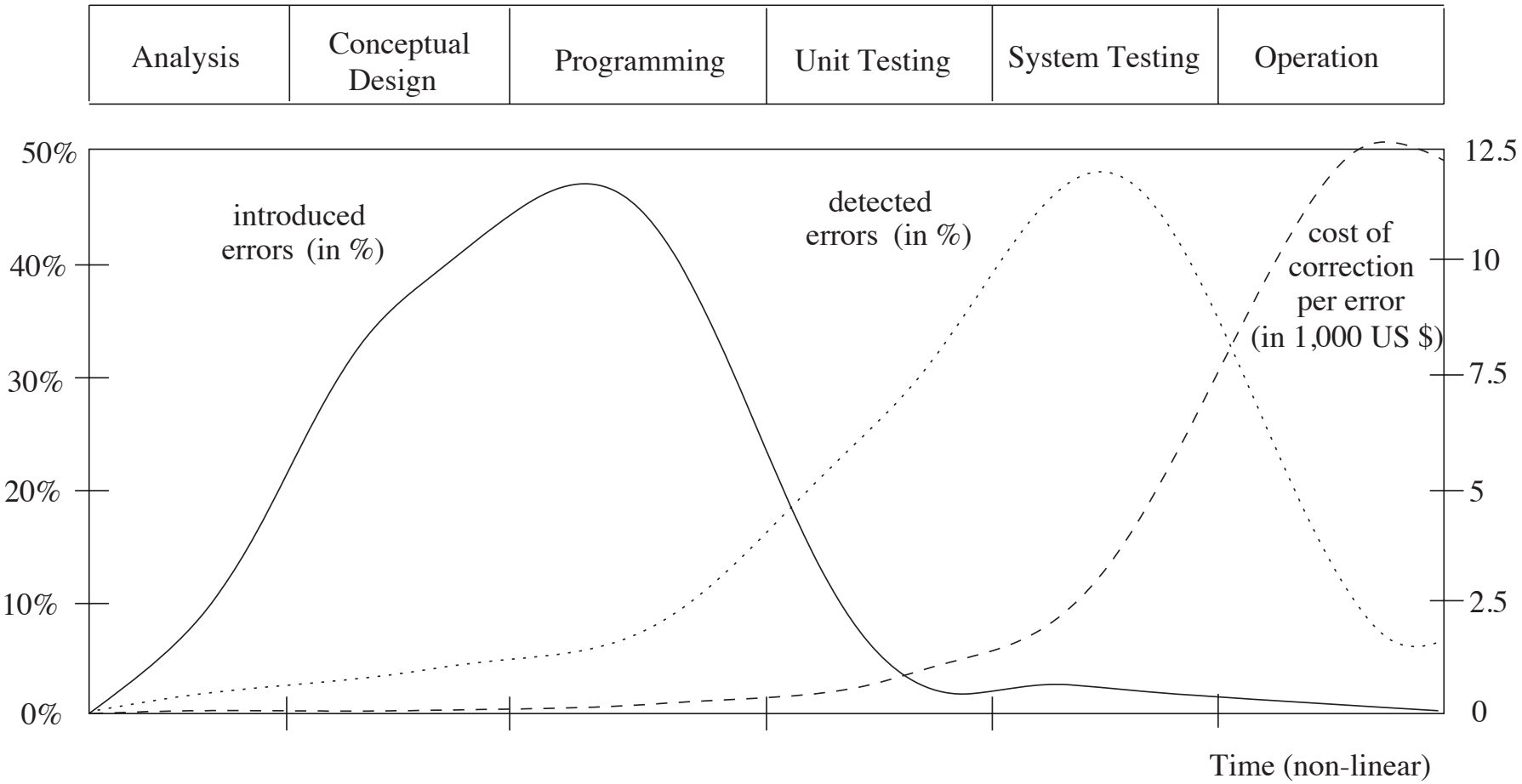


Figure 3.9: Errors in software development

The need for verification

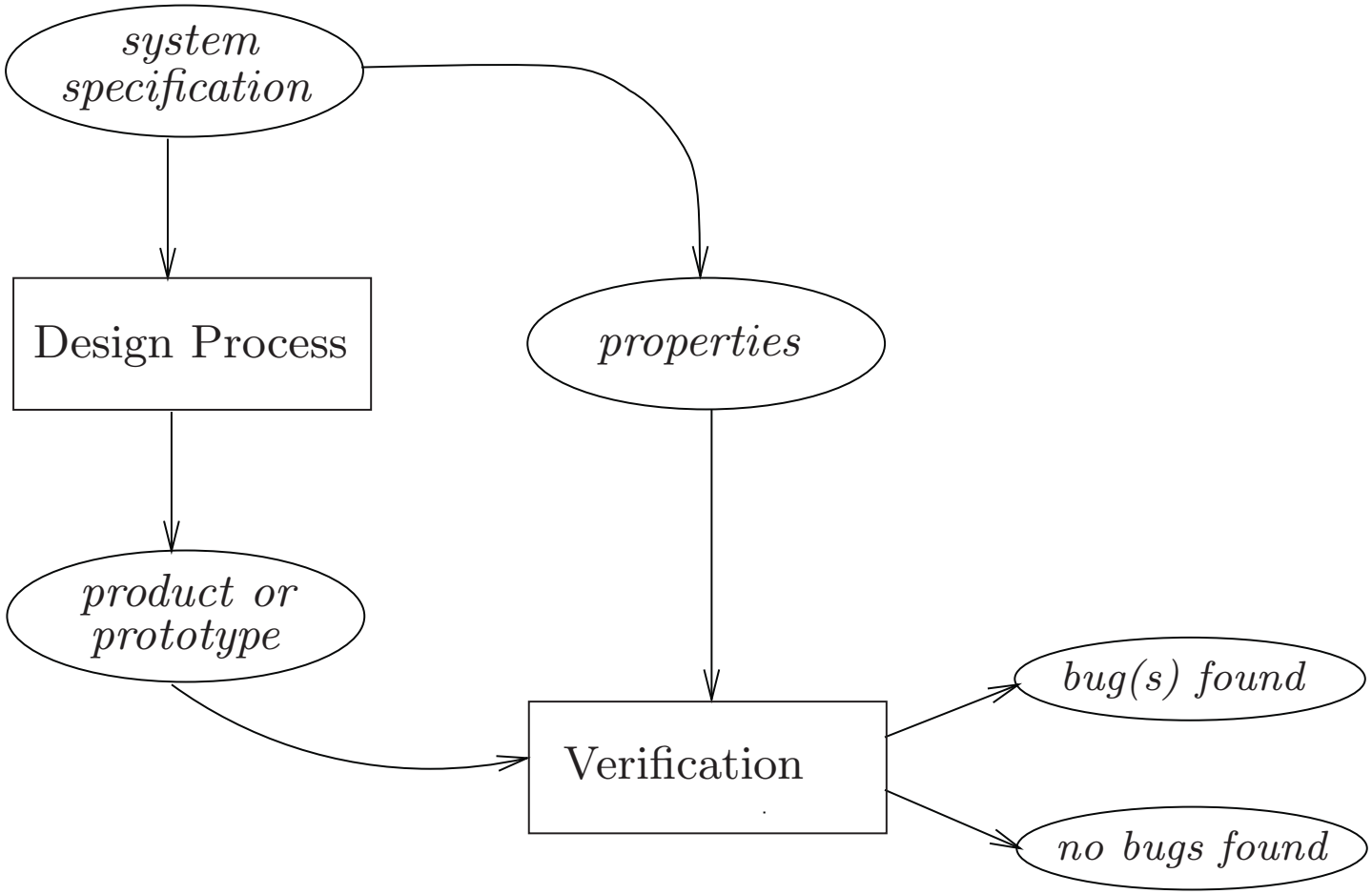


Figure 3.10: Overall approach

Inference Tasks: The Three Questions

The semantical perspective allows us to think about the following inference tasks:

Inference Tasks

Model Checking: Given φ and a model $\mathcal{M}$, does the formula correctly describe this model?

Satisfiability Checking: Given φ, does there exist a model in which the formula is true?

Validity Checking: Given φ, is it true in all models?

Model Checking

- This is the simplest of the three tasks.
- Nevertheless it is useful, e.g. for hardware verification.

Example 3.1

Think of a model $\mathcal{M}$ as a **mathematical picture** of a chip. A logical description φ might define some security issues. If $\mathcal{M}$ fulfills φ, then this means that the chip will be secure wrt. these issues.

Model checking for simple SL

Given a model v and a description φ.
Is $v \models \varphi$ true?

Example 3.2

Given a model v in which a is true and b is false.
Is $\varphi := (\mathsf{a} \vee \neg \mathsf{b}) \rightarrow \mathsf{b}$ true?

Model checking

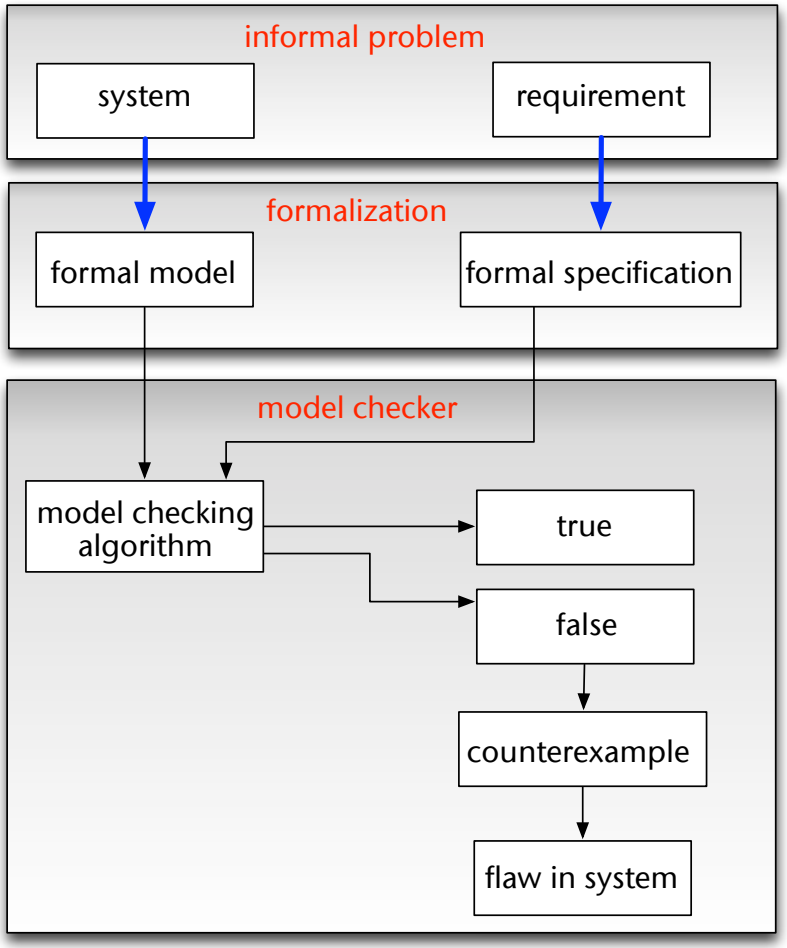


Figure 3.11: Model Checking

Satisfiability Checking

- One can interpret the description as a constraint.
- Is there anything that matches this description?
- We have to create model after model until we find one that satisfies φ.
- In the worst case we have to generate all models.

A good example is our sudoku problem.

Satisfiability Checking for SL

Given a description φ. Is there a model v s.t. $v \models \varphi$ is true?

Validity Checking

- The intuitive idea is that we write down a set of all our fundamental and indisputable axioms.
- Then, with this theory, we derive new formulae.

Example 3.3

Mathematics: We are usually given a set of axioms.
E.g. Euclid's axioms for geometry. We want to prove whether a certain statement follows from this set, or can be derived from it.

Validity Checking for SL

Given a description φ. Does $v \models \varphi$ hold for all models v?

3.2 Basic transition systems

- What is an appropriate abstraction to model hardware and software systems?

  $\rightsquigarrow$ transition systems.
- **Easy**: processes **run** completely **autonomously**.
- Difficult: processes communicate with each other.
- Main notion is a state: this is exactly what we called a model in Chapter 2.

Definition 3.4 (Transition system)

A **transition system** *TS* is a tuple $\langle \mathrm{S}, \mathrm{Act}, \longrightarrow, I, \mathcal{P}rop, \pi \rangle$ where

- S is a set of **states**, denoted by $s_1, s_2, \ldots$,
- Act is a set of **actions**, denoted by $\alpha, \beta, \gamma, \ldots$,
- $\longrightarrow \subseteq \mathrm{S} \times \mathrm{Act} \times \mathrm{S}$ is a **transition relation**,
- $I \subseteq \mathrm{S}$ is the set of **initial states**,
- $\mathcal{P}rop$ is a set of **atomic propositions**, and
- $\pi : \mathrm{S} \to 2^{\mathcal{P}rop}$ is a **labelling (or valuation)**.

A *TS* is **finite** *if, by definition,* S, Act and $\mathcal{P}rop$ are all finite.

This is in accordance with $\mathcal{L}_{SL}$. States are what we called **models**. A labelling corresponds to **a set of valuations**. The new feature is that we can **move between states by means of actions**: instead of $\langle s, \alpha, s' \rangle$ we write $s \xrightarrow{\alpha} s'$.

Example 3.5 (Beverage Vending Machine)

A machine delivers soda or beer after a coin has been inserted. A possible *TS* is:

- $\mathcal{P}rop = \{\text{pay}, \text{soda}, \text{beer}, \text{select}\}$,
- $\mathrm{S} = \{s_{\text{pay}}, s_{\text{beer}}, s_{\text{soda}}, s_{\text{select}}\}$, $I = \{s_{\text{pay}}\}$,
- $\text{Act} = \{\alpha_{\text{insert-coin}}, \alpha_{\text{get-soda}}, \alpha_{\text{get-beer}}, \tau\}$,
- the transitions: $s_{\text{pay}} \xrightarrow{\alpha_{\text{insert-coin}}} s_{\text{select}}$, $s_{\text{beer}} \xrightarrow{\alpha_{\text{get-beer}}} s_{\text{pay}}$, $s_{\text{soda}} \xrightarrow{\alpha_{\text{get-soda}}} s_{\text{pay}}$, $s_{\text{select}} \xrightarrow{\tau} s_{\text{beer}}$, $s_{\text{select}} \xrightarrow{\tau} s_{\text{soda}}$.

Beverage Vending Machine (cont.)

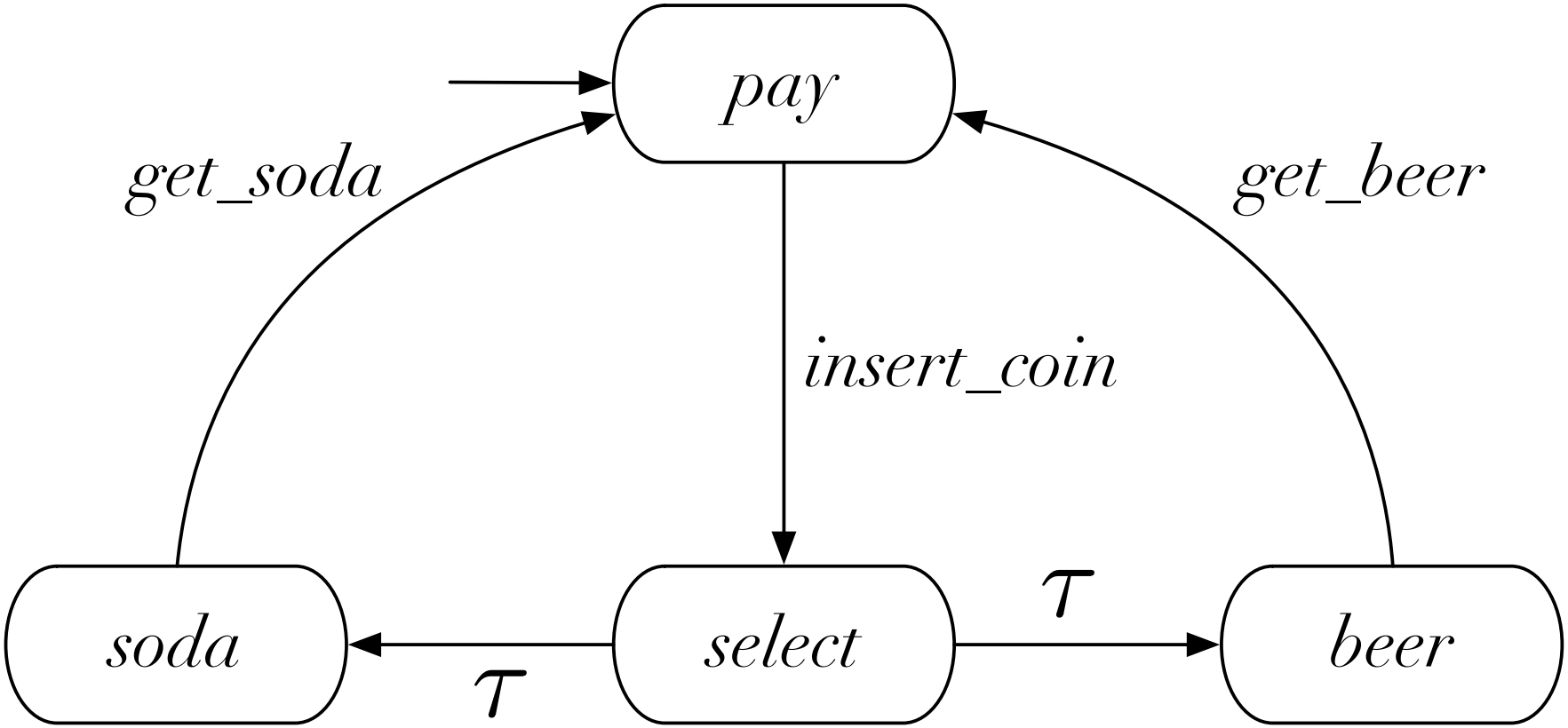


Figure 3.12: *TS* of a beverage vending machine

Simple Beverage Vending Machine (cont.)

- Note that the machine **nondeterministically** chooses beer or soda after a coin has been inserted. Nondeterminism can be seen as **implementation freedom**.
- Often we choose $\mathcal{P}rop = \mathrm{S}$, so that the labelling function is very simple: $\pi(s) = \{s\}$.
- When some properties do not refer to particular constants, we can also choose $\mathcal{P}rop \subsetneq \mathrm{S}$ and $\pi(s) = \{s\} \cap \mathcal{P}rop$.
- We can also choose $\mathcal{P}rop$ and S independently. **"The machine only delivers a drink after a coin has been inserted".** We choose $\mathcal{P}rop = \{\mathsf{paid}, \mathsf{drink}\}$ with labelling function: $\pi(s_{\mathrm{pay}}) =_{def} \emptyset, \pi(s_{\mathrm{soda}}) =_{def} \pi(s_{\mathrm{beer}}) =_{def} \{\mathsf{paid}, \mathsf{drink}\}, \pi(s_{\mathrm{select}}) =_{def} \{\mathsf{paid}\}$.
- Sometimes we want to abstract away from particular actions (internal actions of the machine not of any interest), like τ_1, τ_2. In that case, we use the symbol τ for all such actions.

Simple Hardware Circuit

We consider the hardware circuit diagram below. Next to it is the corresponding transition system *TS*.

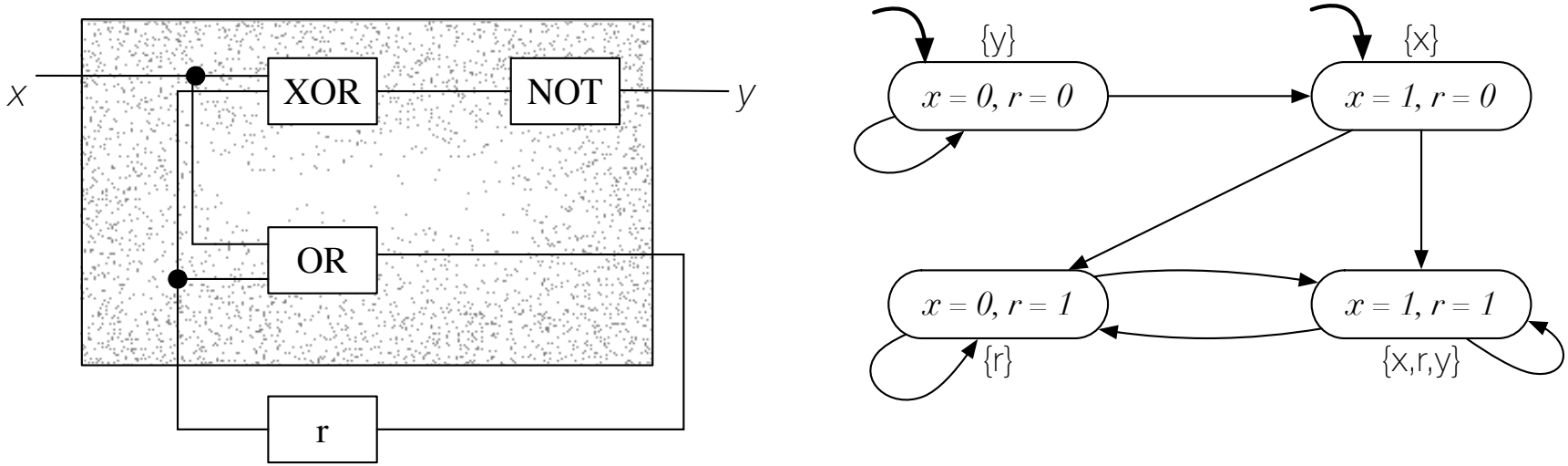


Figure 3.13: *TS* of a hardware circuit

Example 3.6 (Simple Hardware Circuit)

Here we are dealing with boolean variables: input x, output y and register r. They can be easily treated within $\mathcal{P}rop$.

- A state is determined by the contents of x and register r: $s_{x=0,r=0}$, $s_{x=0,r=1}$, $s_{x=1,r=0}$, $s_{x=1,r=1}$.
- $\mathcal{P}rop =_{def} \{\mathsf{x}, \mathsf{r}, \mathsf{y}\}$.
- The labelling is given by $\pi(s_{x=0,r=0}) = \{\mathsf{y}\}$, $\pi(s_{x=0,r=1}) = \{\mathsf{r}\}$, $\pi(s_{x=1,r=0}) = \{\mathsf{x}\}$, $\pi(s_{x=1,r=1}) = \{\mathsf{x}, \mathsf{r}, \mathsf{y}\}$.
- One could also use $\mathcal{P}rop =_{def} \{\mathsf{x}, \mathsf{y}\}$ and modify π accordingly.

Successor/predeccessor, terminal states

Successor (resp. predeccessor) states of a given state s in a transition system *TS* are important. They are determined by all **outgoing** (resp. **ingoing**) transitions.

- $\mathrm{Post}(s) =_{def} \bigcup_{\alpha \in \mathsf{Act}} \mathrm{Succ}(s,\alpha)$, where
  - $\mathrm{Succ}(s,\alpha) =_{def} \{s' : s \xrightarrow{\alpha} s'\}$.
- $\mathrm{Pre}(s) =_{def} \bigcup_{\alpha \in \mathsf{Act}} \mathrm{Anc}(s,\alpha)$, where
  - $\mathrm{Anc}(s,\alpha) =_{def} \{s' : s' \xrightarrow{\alpha} s\}$.

A terminal state s is a state without any outgoing transitions: $\mathrm{Post}(s) = \emptyset$.

Determinism/indeterminism

We already discussed the **indeterminism of transition systems** on Slide 174. It is similar to the **indeterministic choice in a proof-calculus**: there are many paths that are proofs.

- Sometimes the **observable behaviour** is deterministic. How can we formalize that?
- Level of actions versus level of states.

Definition 3.7 (Deterministic transition system)

A transition system *TS* is called

- action-deterministic, *if, by definition,* $|I| \leq 1$ and for all $s \in \mathrm{S}$ and $\alpha \in \mathrm{Act}$: $|\mathrm{Succ}(s, \alpha)| \leq 1$.
- $\mathcal{P}rop$-deterministic, *if, by definition,* $|I| \leq 1$ and for all $s \in \mathrm{S}$ and $T \subseteq \mathcal{P}rop$: $|\mathrm{Post}(s) \cap \{s' \in \mathrm{S} : \pi(s') = T\}| \leq 1$.

Executions

The informal working of a transition system is clear. How can we formally describe it and work with it?

- An execution of a *TS* is a (usually infinite) sequence of states and actions starting in an initial state and built using the transition relation of *TS*.
- We require the sequence to be maximal, i.e. it either ends in a terminal state or it is infinite.
- Sometimes we also consider initial fragments of an execution.
- Reachable states are those states of *TS* that can be reached with initial fragments of executions.

Executions/Reachable states

Definition 3.8 (Executions and reachable states)

Given a transition system *TS*, an execution (or run) is an alternating sequence of states and actions, written $s_0\alpha_1 s_1\alpha_2 \ldots \alpha_n s_n \ldots$ or

$$s_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} s_n \cdots$$

satisfying:

- $s_0 \in I$,
- $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all i,
- either (1) the sequence is finite and ends in a terminal state, or (2) the sequence is infinite.

A state $s \in \mathrm{S}$ is called reachable, if there is a finite initial fragment of an execution that ends in the state s.

Example 3.9 (Extended beverage vending machine)

The extended machine counts the number of bottles available and returns the coin if it is empty. It also refills automatically if there are no bottles left.

- We need more complex transitions: conditional transitions,
  e.g. select $\xrightarrow{\text{nsoda}=0\wedge\text{nbeer}=0:\ \text{ret-coin}}$ start,
  select $\xrightarrow{\text{nsoda}\geq 0:\ \text{sget}}$ start, select $\xrightarrow{\text{nbeer}\geq 0:\ \text{bget}}$ start,
  start $\xrightarrow{\textbf{t}:\ \text{coin}}$ select, start $\xrightarrow{\textbf{t}:\ \text{refill}}$ start.
- sget and bget decrement the numbers nsoda and nbeer by one, refill sets it to the maximum value max.
- This leads to the notion of a **program graph**.
- Such a graph can be unfolded to a transition system TS.
- So we end up with model checking transition systems.

The extended beverage vending machine has been introduced in Example 3.9 on Slide 181. It counts the number of bottles and returns the coin if it is empty. Setting max to 2 we get the un-folded transition system depicted on the right.

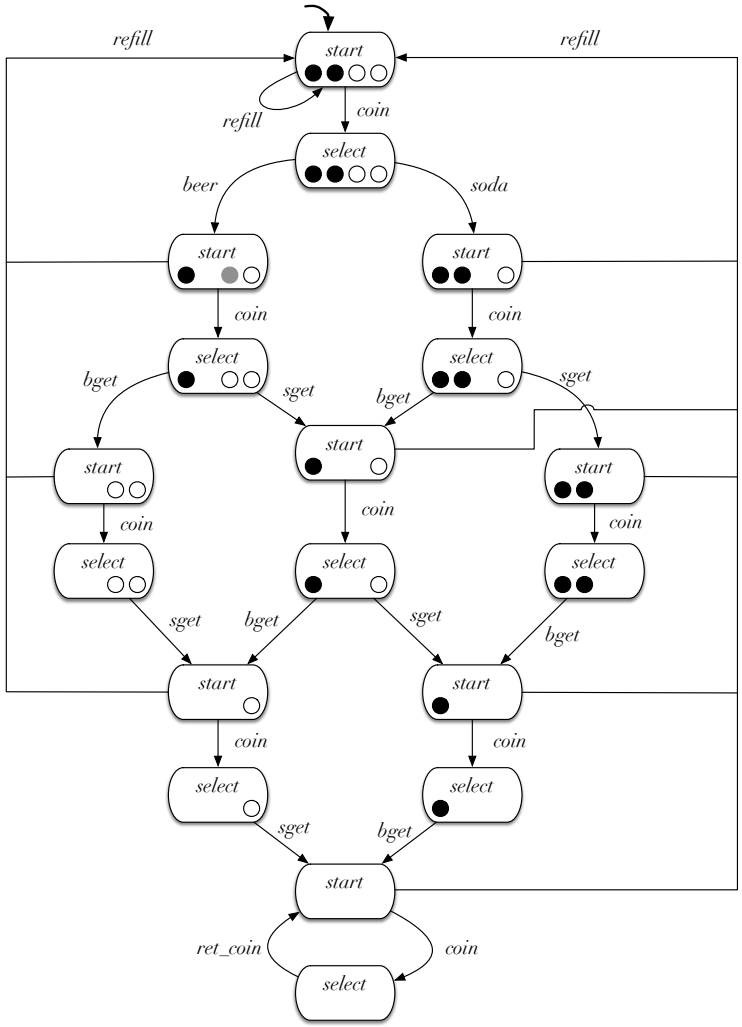


Figure 3.14: *TS* of the extended beverage vending machine

Program graph

- Instead of a *TS*, a **program graph** *PG* is the right notion, as it deals with **conditional transitions** and allows variables Var.
- The role of $\mathcal{P}rop$ and the labelling function π is taken over by an evaluation function Eval, that assigns values to the typed variables in Var (see Definition 3.10).
- The program graph is a directed graph, the nodes of which are called locations: they take the role of the **states** in a *TS*.

Program graph

Definition 3.10 (Program graph)

A program graph PG over a set of typed variables Var is a directed graph where the edges are labelled with conditions and actions: it is a tuple $\langle \text{Loc}, \text{Act}, \text{Effect}, \hookrightarrow, \text{Loc}_0, \mathcal{P}rop, g_0 \rangle$ where

- Loc is a set of **locations**, denoted by $l_1, l_2, \ldots$,
- Act is a set of **actions**, denoted by $\alpha, \beta, \gamma, \ldots$,
- Eval(Var) is the set of all assignments ϱ of values to variables in Var (compatible with their type).
- Effect : $\text{Act} \times \text{Eval}(\text{Var}) \to \text{Eval}(\text{Var})$ takes an assignment ϱ, applies action α and computes the resulting assignment ϱ'.
- $\hookrightarrow \subseteq \text{Loc} \times \text{Cond}(\text{Var}) \times \text{Act} \times \text{Loc}$ is a **conditional transition relation**,
- $\text{Loc}_0 \subseteq \text{Loc}$ is the set of **initial locations**,
- $g_0 \in \text{Cond}(\text{Var})$ is the initial condition.

Boolean conditions over Var

- Cond(Var) is the set of all boolean conditions over Var. E.g.:

$$(-3 \leq x \lneq 5) \wedge (y = \text{blue}) \vee (z = \square)$$

- How can we define these conditions formally?
- Assume we consider variables $x_1, \ldots, x_n$ of particular types ranging over domains $\mathrm{dom}_1, \ldots, \mathrm{dom}_n$. For $1 \leq i \leq n$ let $D_i \subseteq \mathrm{dom}_i$. We consider the set $\mathcal{P}rop_{\mathsf{Var}}$ of sentential constants of the form "$\mathsf{p}_{x \in D_i}$" (for arbitrary i, and variable x of type i and D_i). $\mathsf{p}_{x \in D_i}$ is true whenever the actual value of x is indeed in D_i, otherwise it is false.

  Then Cond(Var) is the set of all SL formulae over $\mathcal{P}rop_{\mathsf{Var}}$.

PG of extended beverage vending machine

Var: $\mathrm{nsoda}, \mathrm{nbeer}$, with domain $\{0, 1, \ldots, \mathrm{max}\}$;

Loc: start, select;

Act: bget, sget, coin, ret-coin, refill;

Effect: $\langle \mathsf{coin}, \eta \rangle \mapsto \eta$, $\langle \mathsf{ret\text{-}coin}, \eta \rangle \mapsto \eta$,
$\langle \mathsf{sget}, \eta \rangle \mapsto \eta[\mathrm{nsoda} - 1/\mathrm{nsoda}]$,
$\langle \mathsf{bget}, \eta \rangle \mapsto \eta[\mathrm{nbeer} - 1/\mathrm{nbeer}]$,
$\langle \mathsf{refill}, \eta \rangle \mapsto \eta'$, where $\eta'(\mathrm{nsoda}) := \eta'(\mathrm{nbeer}) := \mathrm{max}$.

$\hookrightarrow$: obvious from Figure 3.14,

Loc_0: start,

g_0: $\mathrm{nsoda} = \mathrm{max} \ \wedge \ \mathrm{nbeer} = \mathrm{max}$.

Synchronous product

- Often there exists a **central clock** that allows two transition systems to be synchronized. e.g. **synchronous hardware circuits**.
- This leads to the synchronous product $TS_1 \otimes TS_2$ of two transition systems: both systems have to perform **all steps** in a synchronous fashion.

Synchronous product (cont.)

Definition 3.11 (Synchronous product: $TS_1 \otimes TS_2$)

Given two transition systems TS_1, TS_2 ($\langle \mathrm{S}_i, \mathrm{Act}, \longrightarrow_i, I_i, \mathcal{P}rop_i, \pi_i \rangle$, $i = 1, 2$) with a common set of actions Act, we define the synchronous product $TS_1 \otimes TS_2$ by

$$\langle \mathrm{S}_1 \times \mathrm{S}_2, \mathrm{Act}, \longrightarrow, I_1 \times I_2, \mathcal{P}rop_1 \cup \mathcal{P}rop_2, \pi \rangle$$

where the labelling π and $\longrightarrow$ are defined by

- $\pi(\langle s_1, s_2 \rangle) =_{def} \pi_1(s_1) \cup \pi_2(s_2)$,
- $\langle s_1, s_2 \rangle \xrightarrow{\alpha \star \beta} \langle s_1', s_2' \rangle$ *if, by definition,* $s_1 \xrightarrow{\alpha}_1 s_1'$ and $s_2 \xrightarrow{\beta}_2 s_2'$.

$\star$ is a mapping from Act $\times$ Act into Act that assigns to each pair $\langle \alpha, \beta \rangle$ an action $\alpha \star \beta$.

$\star$ is usually assumed to be commutative and associative. Often action names are irrelevant (e.g. for hardware circuits), so they do not play any role in such cases.

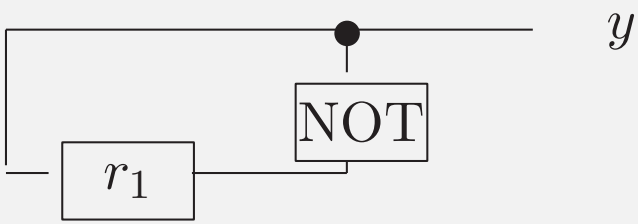
Synchronous product

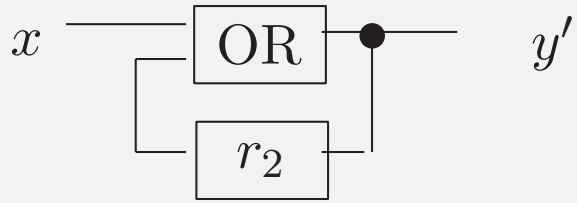
- There is no autonomy, as in the **interleaving operator** to be introduced below in Definition 3.14 on Slide 198.
- Normally, when asynchronous systems are represented by transition systems, one does not make any assumptions about **how long the processes take** for execution. Only that these are finite time intervals.

Synchronous product

Example 3.12 (Synchronous product of two circuits)

We consider the following circuits. The first one has no input variables (output is y defined by r_1 and register transition $\neg r_1$), the second one has input x, and output y' defined by $x \vee r_2$ and register transition $x \vee r_2$.

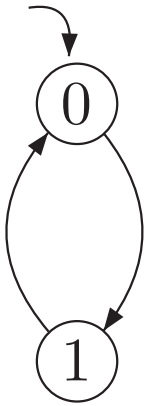




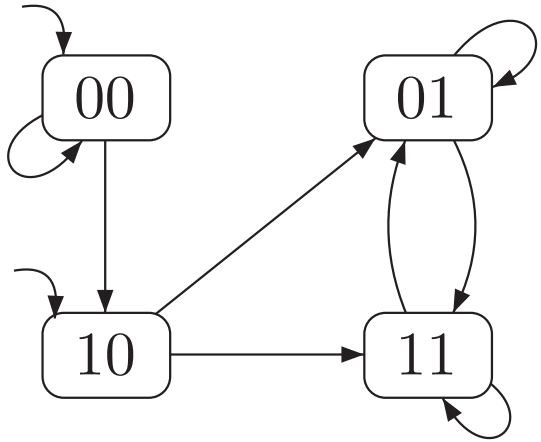
Synchronous product

The corresponding transition systems are:

TS_1 :



TS_2 :



Synchronous product

The synchronous product of Example 3.12 is:

$TS_1 \otimes TS_2$:

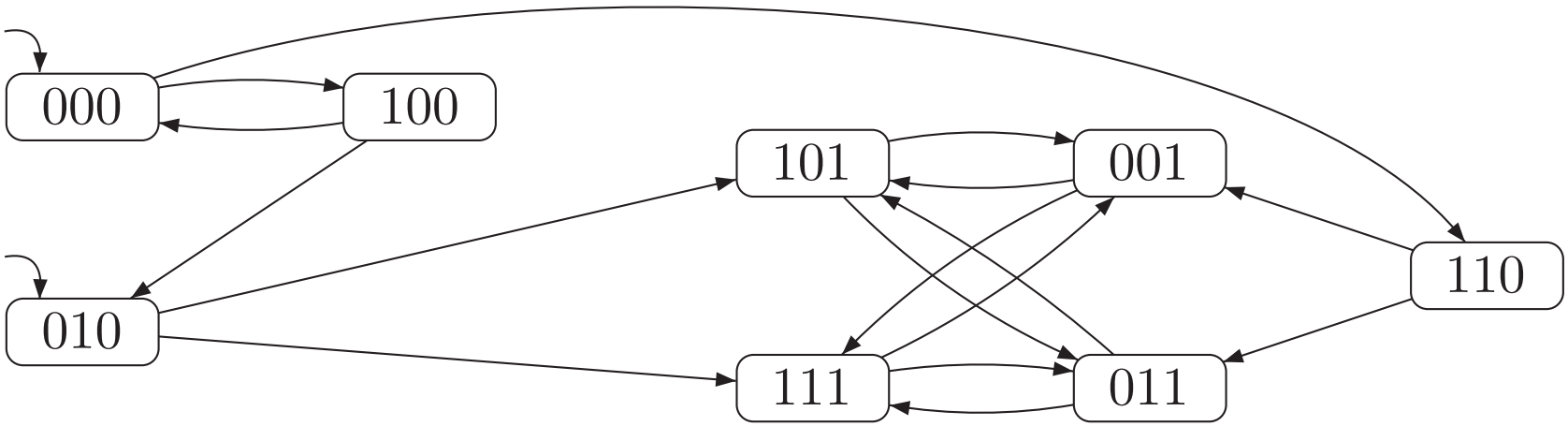


Figure 3.15: Synchronous product of two circuits

3.3 Interleaving and handshaking

Interleaving independent processses

How can we merge two completely independent transition systems? (Think of finite state automata!)

Consider traffic lights that can simply switch between $green$ and red.

Interleaving independent processses (cont.)

Example 3.13 (Two independent traffic lights)

Two independent traffic lights on two roads.

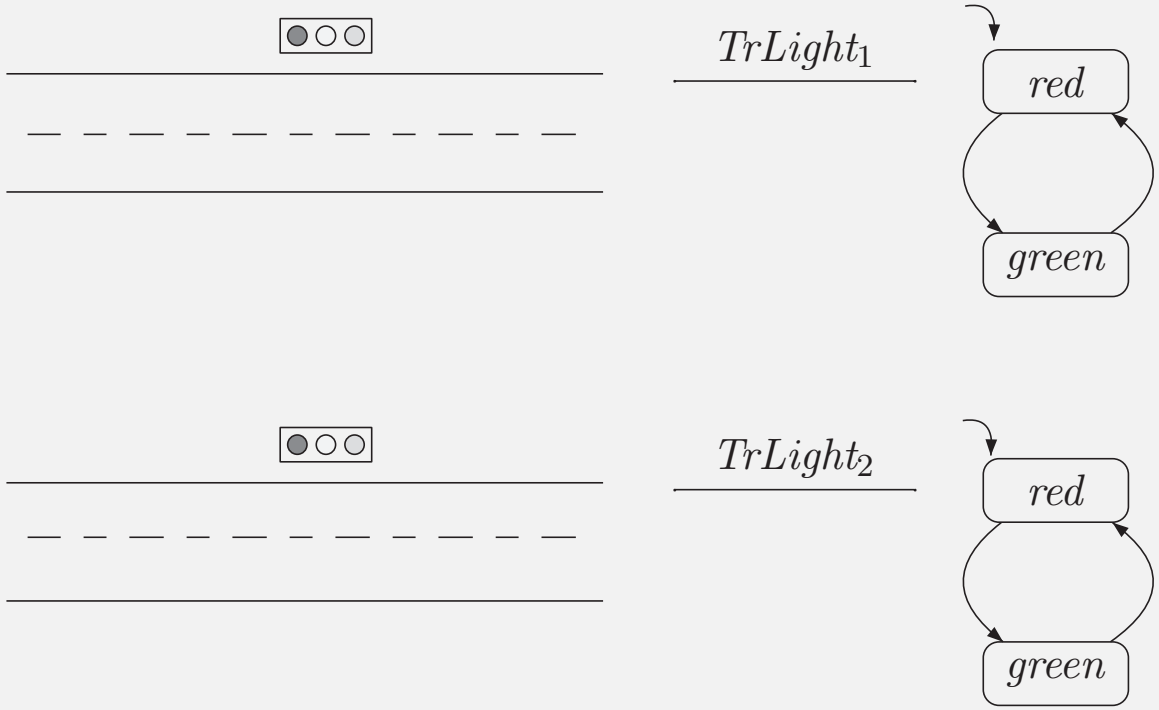


Figure 3.16: *TS* of two independent traffic lights

Interleaving independent processses (cont.)

The result should be:

$$TrLight_1 \;|||\; TrLight_2$$

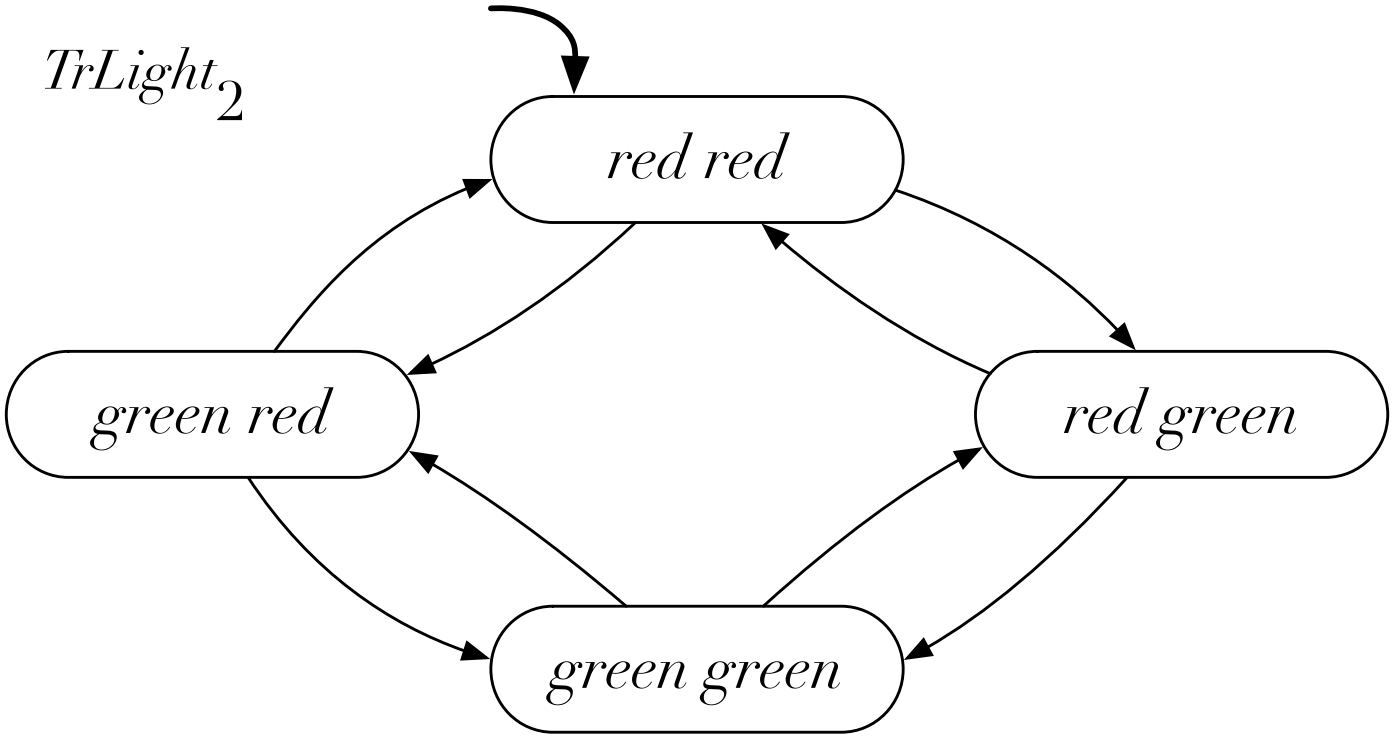


Figure 3.17: *TS* of two traffic lights

Interleaving independent processses

The result of interleaving the following processes is also obvious.

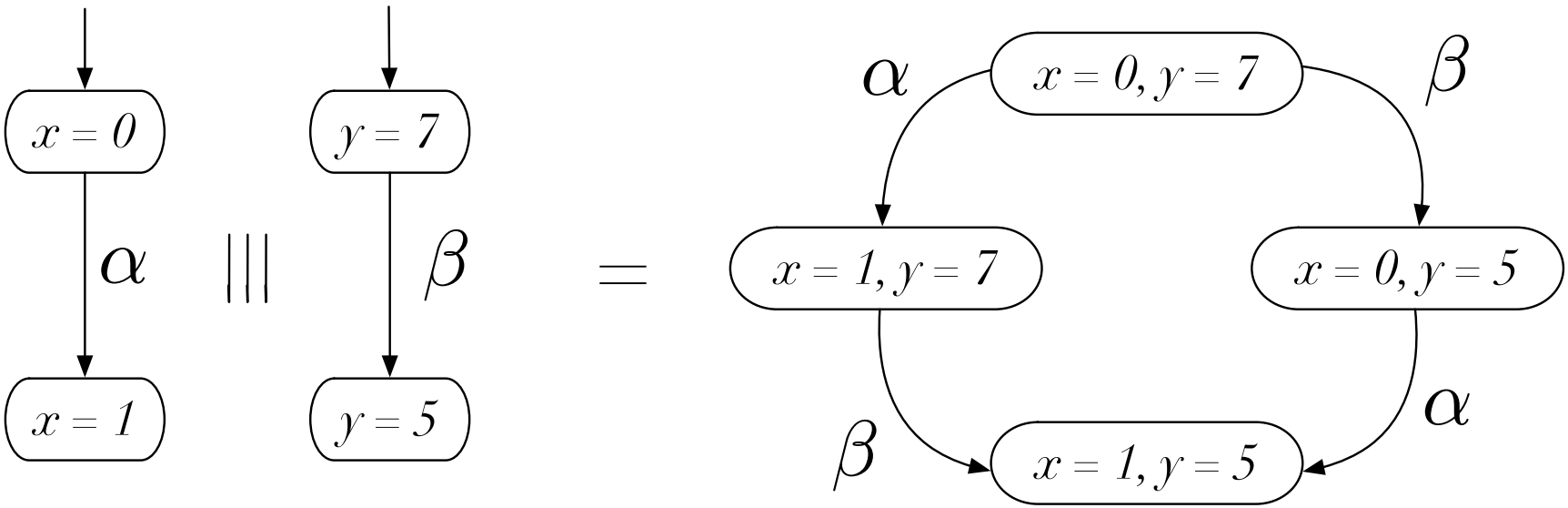


Figure 3.18: *TS* of two independent processes

Definition 3.14 (Interleaving: $TS_1 \,|||\, TS_2$)

Given two transition systems TS_1, TS_2 ($\langle \mathrm{S}_i, \mathrm{Act}_i, \longrightarrow_i, I_i, \mathcal{P}rop_i, \pi_i \rangle$, $i = 1, 2$) we define the **interleaved transition system $\mathrm{TS}_1 \,|||\, \mathrm{TS}_2$** by

$$\langle \mathrm{S}_1 \times \mathrm{S}_2, \mathrm{Act}_1 \cup \mathrm{Act}_2, \longrightarrow, I_1 \times I_2, \mathcal{P}rop_1 \cup \mathcal{P}rop_2, \pi \rangle$$

where the labelling π and $\longrightarrow$ are defined by

- $\pi(\langle s_1, s_2 \rangle) =_{def} \pi_1(s_1) \cup \pi_2(s_2)$,
- $\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2' \rangle$ *if, by definition,* $s_1 \xrightarrow{\alpha}_1 s_1'$ **or** $s_2 \xrightarrow{\alpha}_2 s_2'$.

In contrast to the synchronous product (Definition 3.11 on Slide 188) there is no clock: executions do not depend on time.

Shared variables

The examples on Slides 196 and 197 are simple, because there are no shared variables.
What happens, if we do the same construction in the following example:

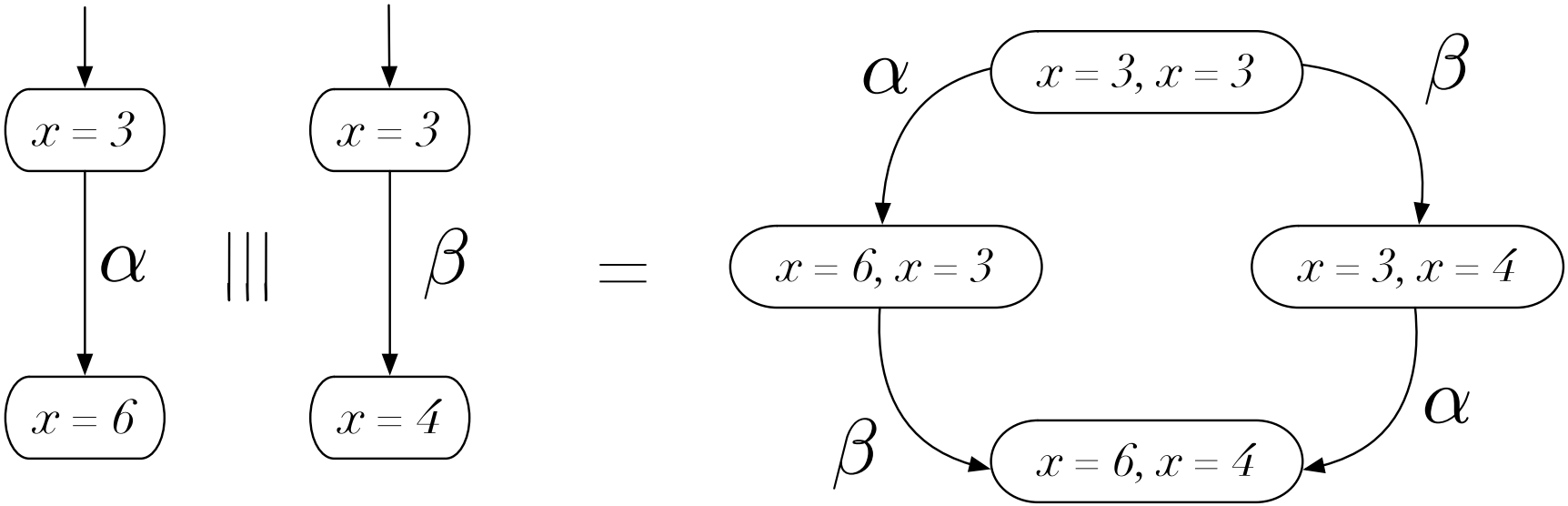


Figure 3.19: *TS* of processes with shared variable

Semaphores

Example 3.15 (Mutex via a semaphore)

Two processes P_1 and P_2 have access to a binary variable y, a **semaphore**. Value 0 of y means the semaphore **is possessed** by one process, value 1 means it is free.

- Each process P_i is in one of three states: noncrit_i, wait_i, crit_i.
- There is a transition from noncrit_i to wait_i and from wait_i to crit_i but the last one only when y is 1 (then the transition fires and the value is set to 0).
- There is a transition from crit_i to noncrit_i and y is set to 1.

How can we model this as a TS?

Semaphores (cont.)

We first model it as program graphs.

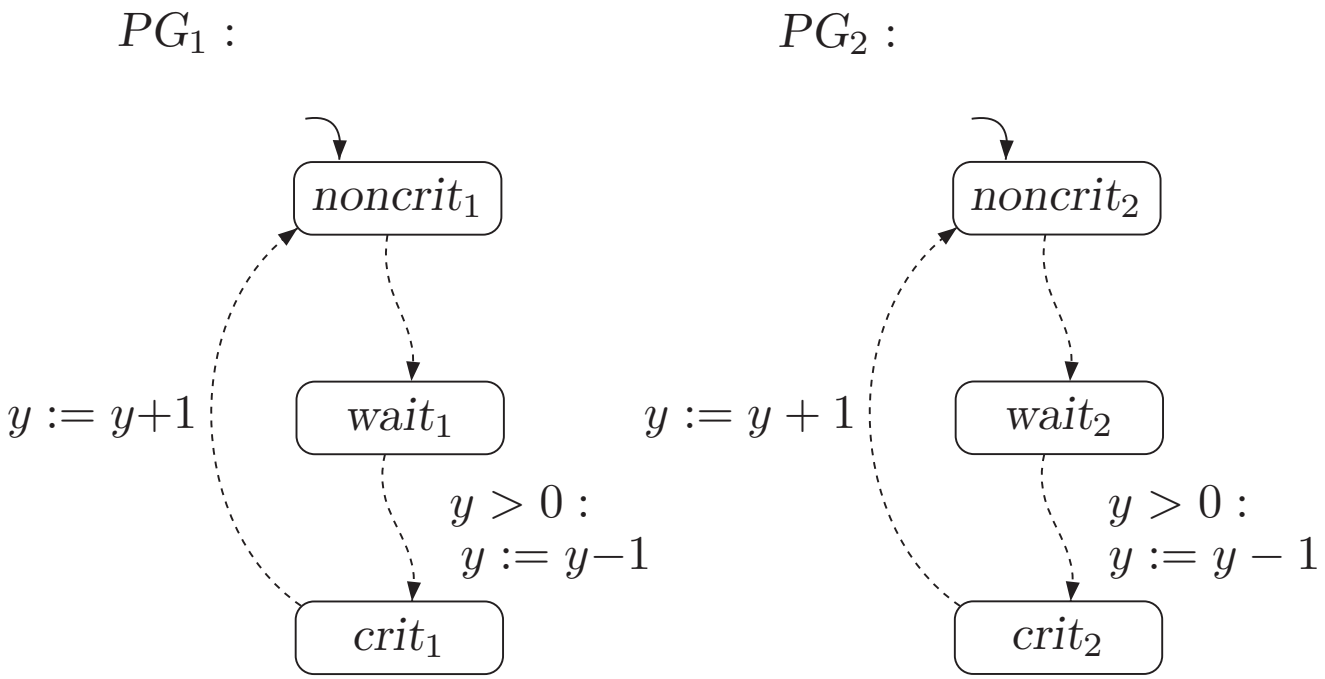


Figure 3.20: Semaphores and program graphs

Interleaved program graph

A construction similar to Definition 3.14 yields an **interleaved program graph**

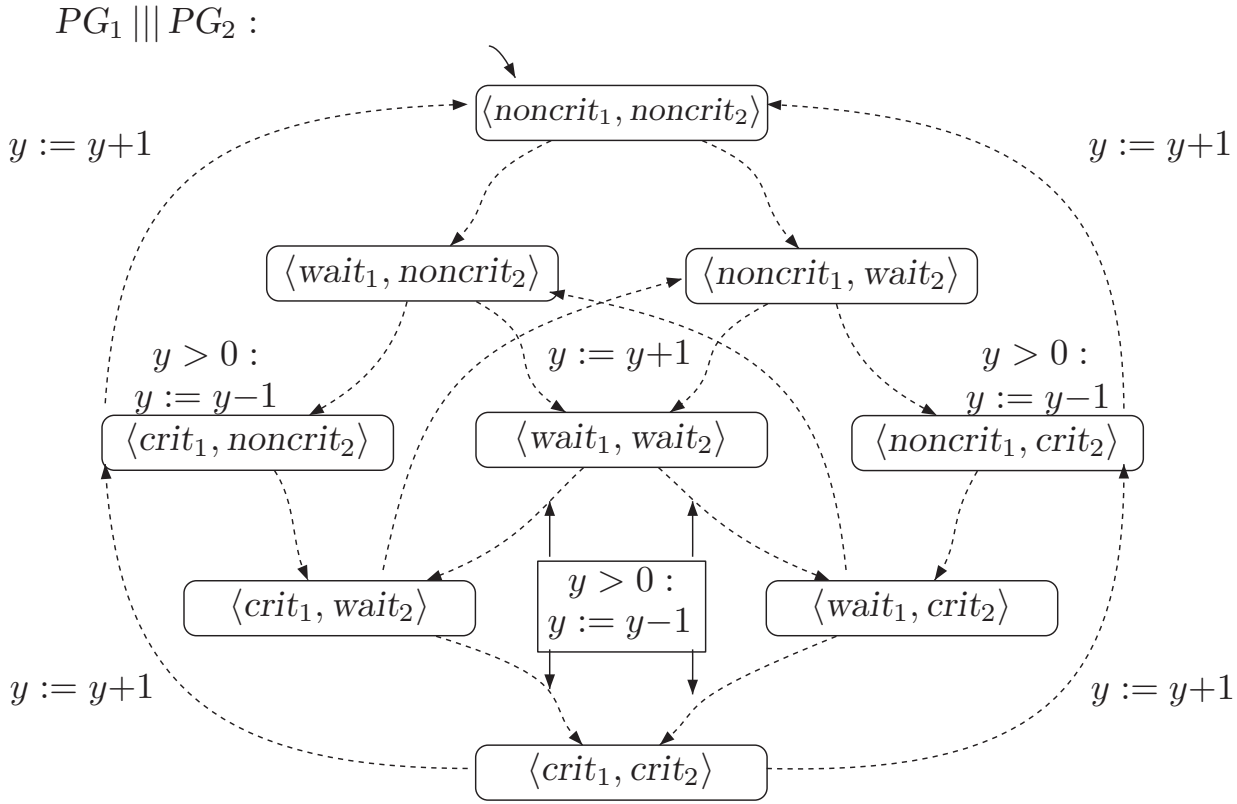


Figure 3.21: Interleaving program graphs

Unfolding program graph into *TS*

Here is the transition system that we obtain from Figure 3.21

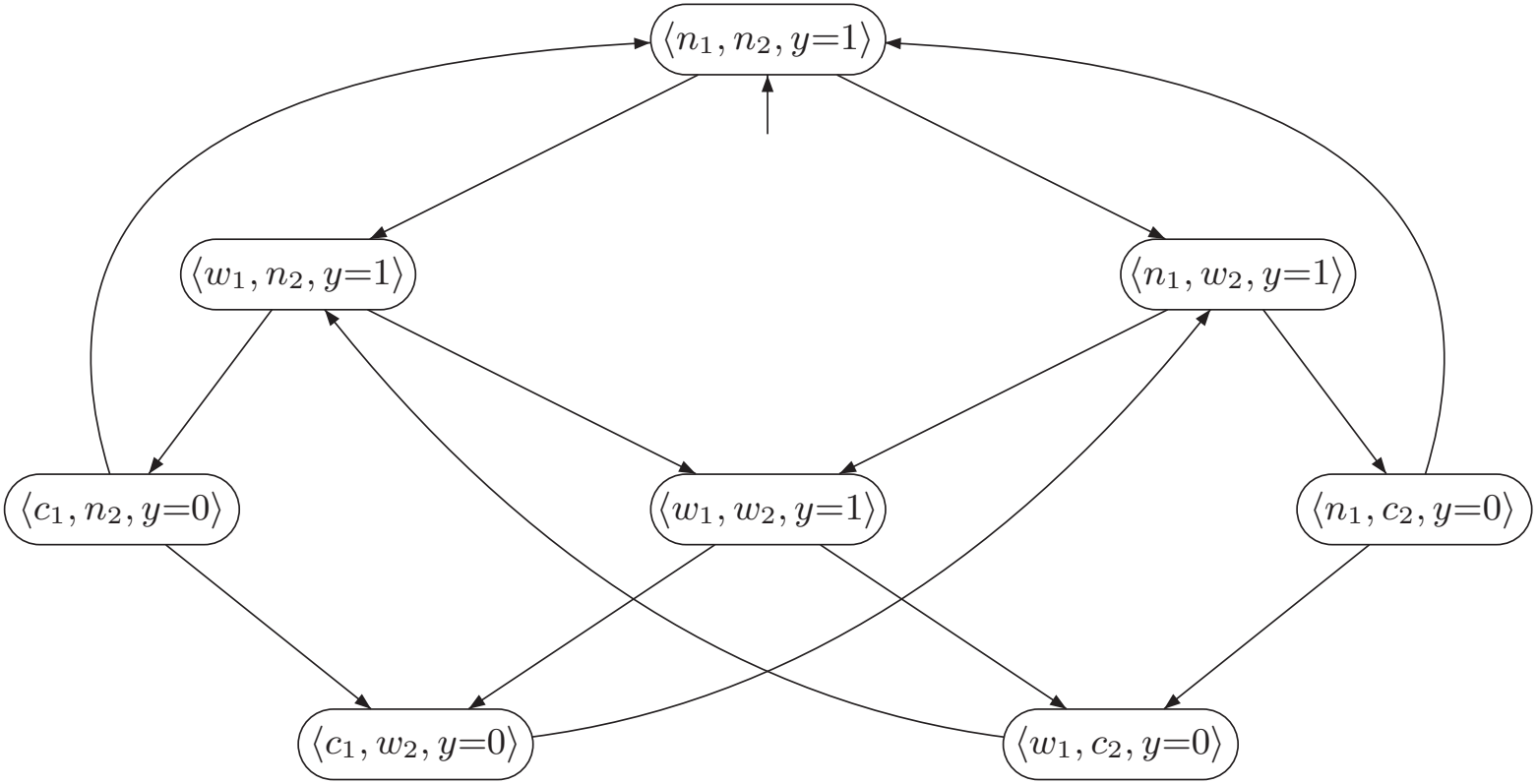


Figure 3.22: The final transition system

Semaphores (cont.)

- The transition system **does not contain anymore the critical state** (because it is not reachable). The system satisfies the mutual exclusion property.
- But it is possible that both processes are at the same time in the waiting state: this is a **deadlock** situation that should be avoided (by an appropriate scheduling algorithm to implement the nondeterministic choice).

What have we achieved so far?

- Interleaving of transition systems works well for **independent** systems.
- It is not appropriate when synchronization of certain parts is essential.
- Instead of a *TS*, the **program graph** *PG* (introduced in Definition 3.10 on Slide 184) is the right notion, as it deals with **conditional transitions** and allows **shared** variables Var.
- We have just seen that shared variable communication (e.g. semaphores) can be used to model synchronization.

What have we achieved so far? (cont.)

- An interleaving of program graphs can be defined similarly to Definition 3.14: see Slide 202.
- This **interleaved graph** can again be unfolded to a transition system *TS*: see Slide 203.
- We end up again with model checking transition systems.

We are now introducing another method for modelling **synchronization**: handshaking, a refined version of interleaving on the level of transition systems (not program graphs).

From interleaving to handshaking

- Concurrent processes that need to interact can do so in a synchronous fashion via handshaking.
- Sometimes one process has to wait for the other to finish: they have to synchronize.
- We introduce a set of distinguished handshake actions H.
- We define a transition system $TS_1 \|_H TS_2$ that takes into account the handshakes in H.
- For an empty set H we get back our notion of an interleaved transition system $TS_1 \,|||\, TS_2$.

Example 3.16 (Traffic junction)

We consider a traffic junction with two traffic lights. Both switch from green to red and green etc., but in a synchronized way: when one is green the other one should be red (to avoid accidents).

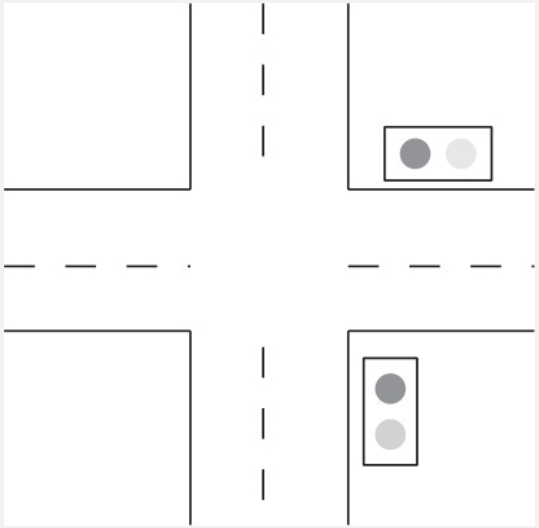


Figure 3.23: Traffic junction

Traffic junction

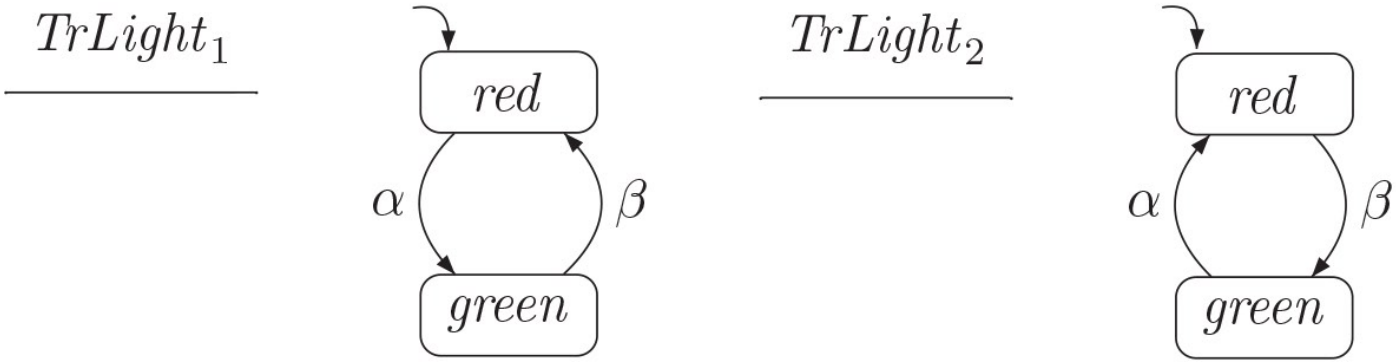
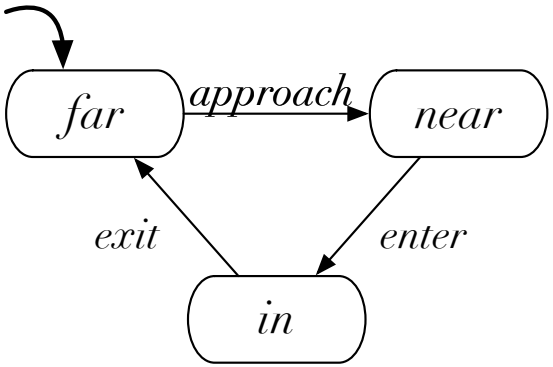


Figure 3.24: Traffic junction

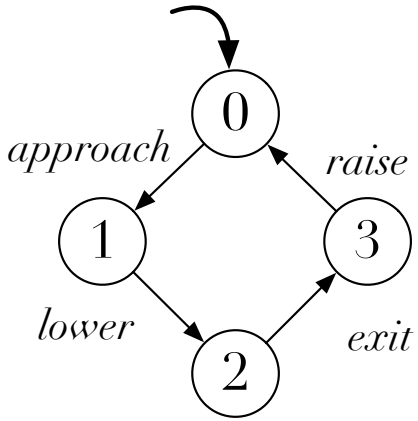
The interleaved system $TrLight_1 \;|||\; TrLight_1$ does not work anymore!

Example 3.17 (Railroad crossing)

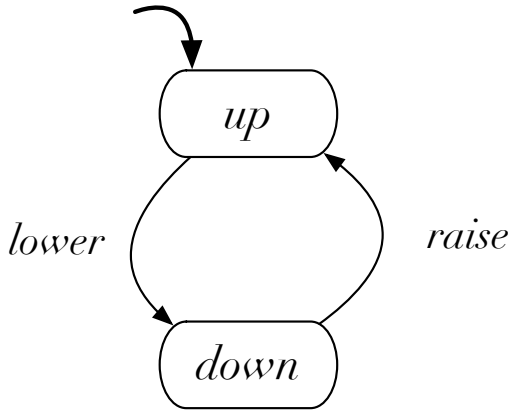
A railroad crossing consists of a train, a gate and a controller. An approaching train sends a signal so that the gates have to be closed. The gates open again only when another signal, indicating that the train has passed, has been sent.



Train



Controller



Gate

Figure 3.25: Components of Railroad crossing

Mutual exclusion in general

Suppose we have two processes P_1, P_2 with **shared variables**.

- When they are operating on these shared variables, they are in a **critical** phase. Otherwise they are in an **uncritical** phase.
- We assume the processes **alternate between critical and noncritical** phases (infinitely often).
- The problem is to avoid concurrency when **both are in critical phases**.

Mutual exclusion in general (cont.)

- Mutual exclusion algorithms are scheduling algorithms to ensure that no critical actions are executed in parallel.
- One of the most prominent solutions is Peterson's mutex algorithm.
- As Peterson's algorithm is based on the **interleaving of program graphs** (which we have not formally introduced but illustrated on Slides 201—203), we introduce a variant based on $\|_H$ between transition systems.

Definition 3.18 (Handshaking: $TS_1 \|_H TS_2$)

For transition systems TS_1, TS_2 ($\langle S_i, Act_i, \longrightarrow_i, I_i, \mathcal{P}rop_i, \pi_i \rangle$), $H \subseteq Act_1 \cap Act_2$ and $\tau \notin H$, we define the transition system with handshaking $TS_1 \|_H TS_2$ by

$$\langle S_1 \times S_2, Act_1 \cup Act_2, \longrightarrow, I_1 \times I_2, \mathcal{P}rop_1 \cup \mathcal{P}rop_2, \pi \rangle$$

where the labelling π and $\xrightarrow{\alpha}$ are defined by

- $\pi(\langle s_1, s_2 \rangle) =_{def} \pi_1(s_1) \cup \pi_2(s_2)$,
- for $\alpha \notin H$, $\xrightarrow{\alpha}$ is defined as in Definition 3.14,
- for $\alpha \in H$, $\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2' \rangle$ *if, by definition,* $s_1 \xrightarrow{\alpha}_1 s_1'$, and $s_2 \xrightarrow{\alpha}_2 s_2'$.

When $H = Act_1 \cap Act_2$, we simply write $TS_1 \| TS_2$ instead of $TS_1 \|_H TS_2$. When $H = \emptyset$ then $TS_1 \|_\emptyset TS_2 = TS_1 \,|||\, TS_2$.

Example 3.19 (Mutual exclusion via an arbiter)

Suppose we have two processes P_1 and P_2. Both alternate infinitely often between critical and noncritical phases. The interleaved system $TS_1 \,|||\, TS_2$ has a critical state that has to be avoided (shared resources). **How can a third process, the arbiter, be used to resolve the conflicts?**

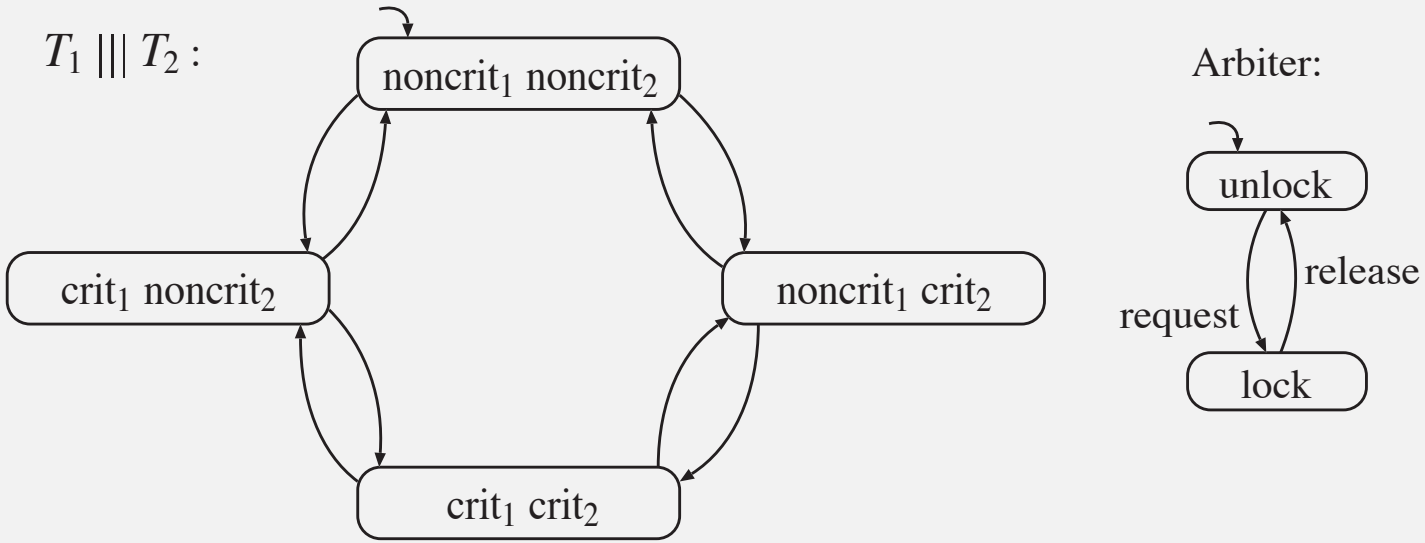


Figure 3.26: Transition systems and arbiter

Mutual exclusion via an arbiter (cont.)

Asssume P_1 and P_2 have each actions request and release. We set $H = \{\text{request}, \text{release}\} = \text{Act}_1 \cap \text{Act}_2$ and get the system:

$$(T_1 \;|||\; T_2) \;||\; \text{Arbiter} :$$

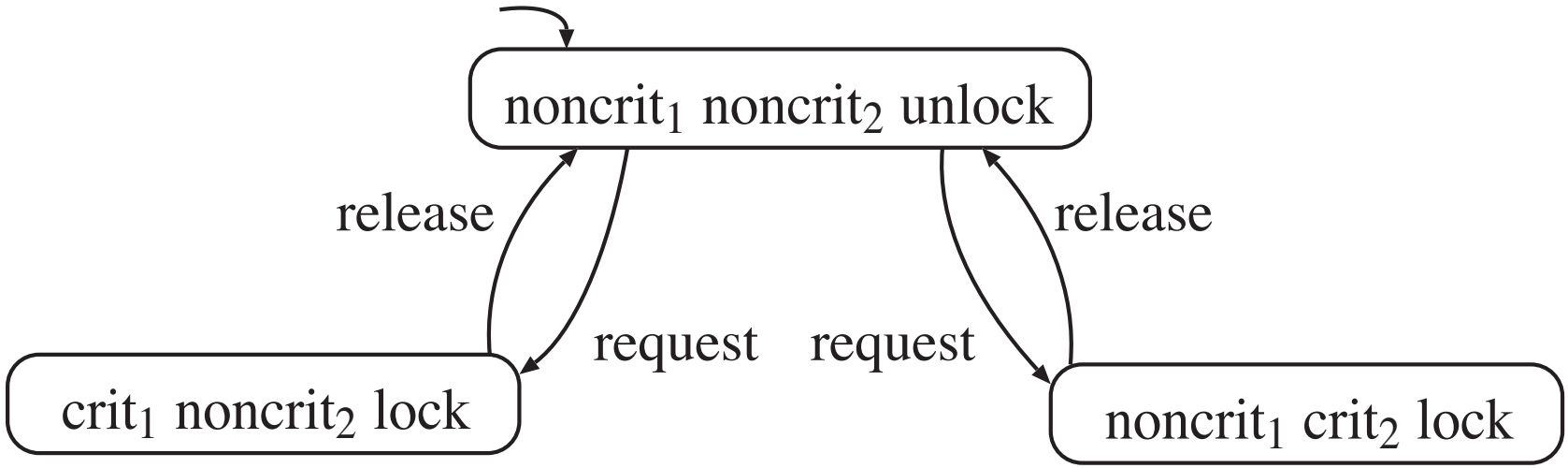


Figure 3.27: Mutual exclusion via arbiter

Traffic junction revisited

- We reconsider Example 3.16 from Slide 208.
- This time we build $TS_1 \|_H TS_2$ with $H = \{\alpha, \beta\}$ and it all works!
- However, there is a small problem. The state $\langle red, red \rangle$ is a **deadlock**.
- So even if two transition systems do not have deadlocks (an assumption to be made on Slide 229) their parallel composition might have.

Railroad crossing revisited

- We reconsider Example 3.17 from Slide 210.
- This time we build

$$Train \|_{\{approach,exit\}} Controller \|_{\{lower,raise\}} Gate,$$

or, according to our convention, simply

$$Train \| Controller \| Gate.$$

Railroad crossing: almost working

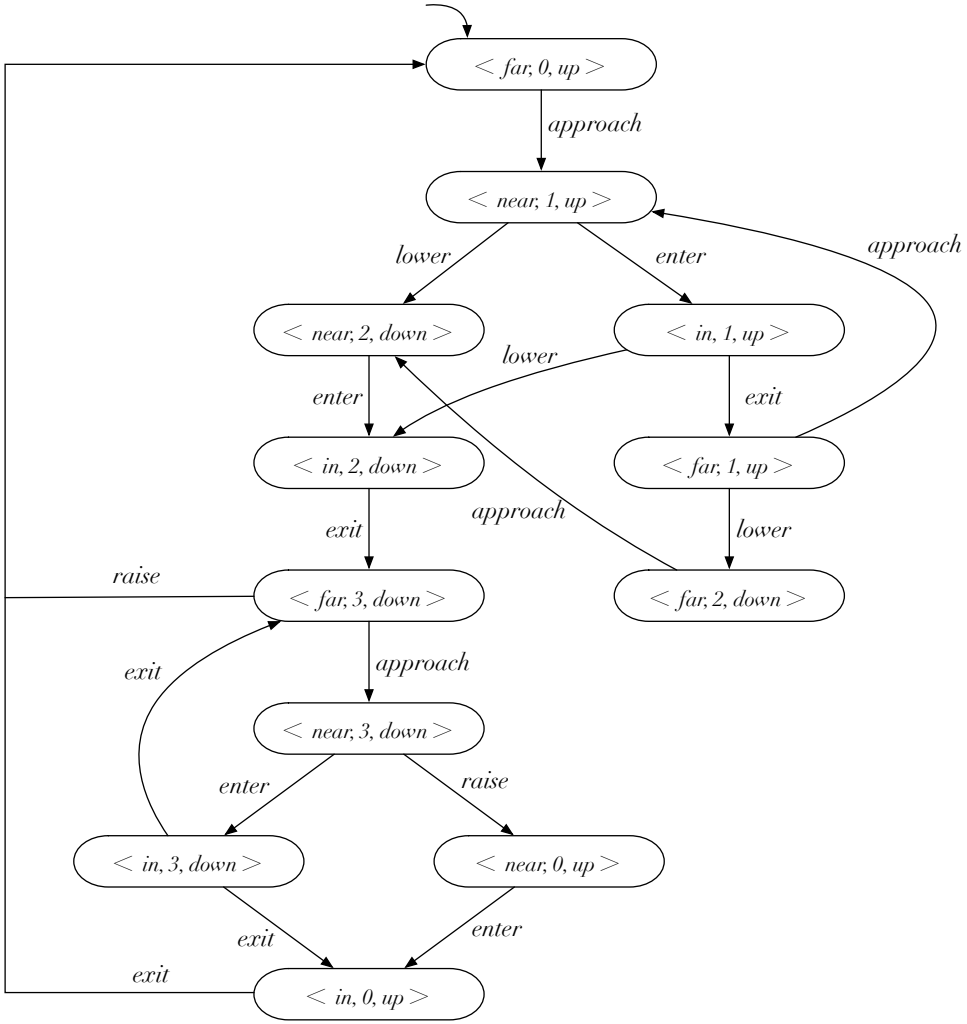


Figure 3.28: *TS* of Railroad crossing

Dining Philosophers revisited

We reconsider Example 1.2 from Slide 41. An obvious representation is for each philosopher to have **four actions available** (request left stick, request right stick, release left stick, release right stick) — we model "thinking" and "waiting" in the states. And for each stick also two requests and two releases (from the adjacent philosophers). This results in the **overall system**

$$Phil_0 || Stick_4 || Phil_4 || Stick_3 || Phil_3 || Stick_2 || \ldots || Phil_4 || Stick_0$$

Dining Philosophers revisited (cont)

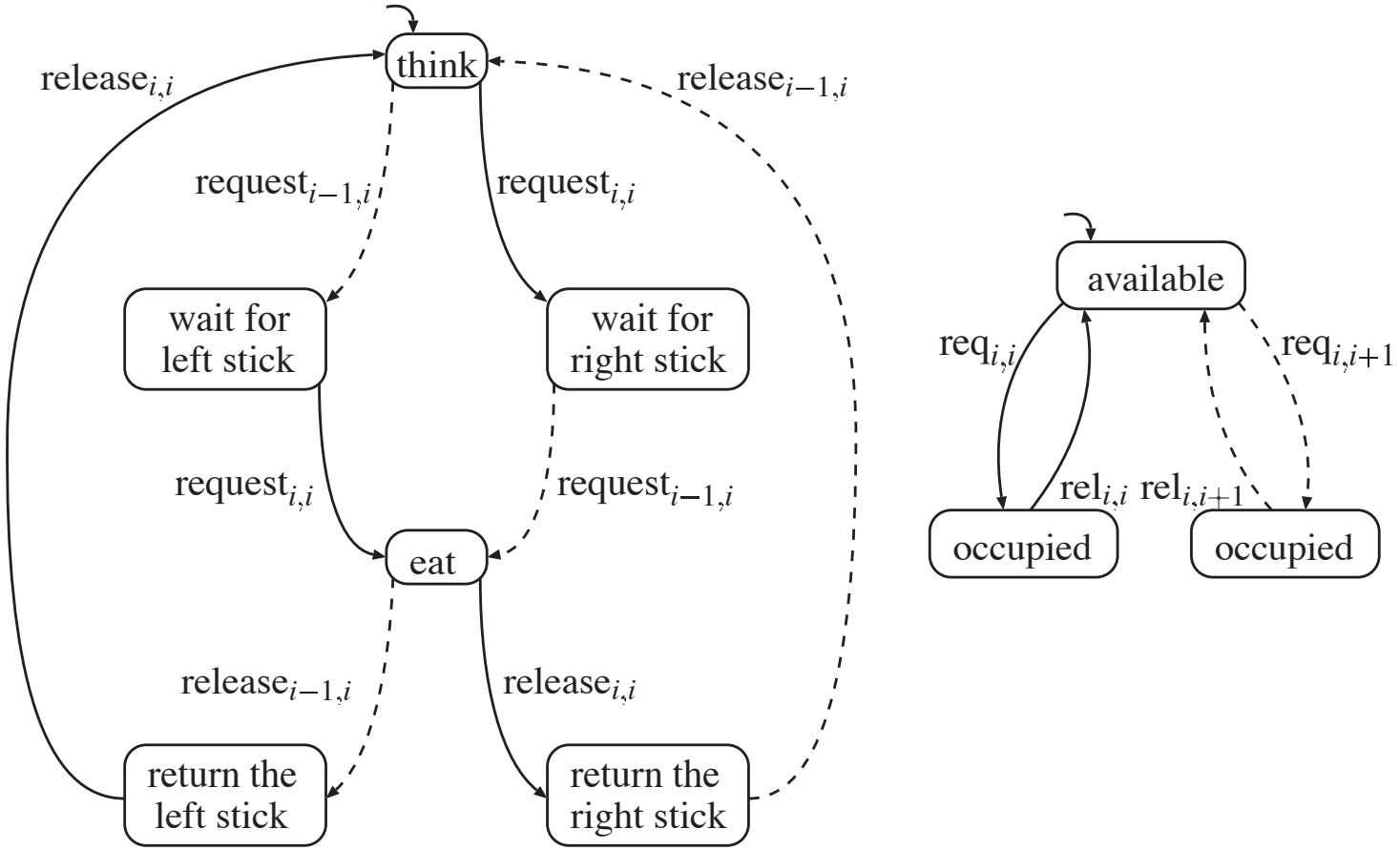


Figure 3.29: Dining Philosophers

Dining Philosophers revisited (cont.)

- What is wrong with this model?
- deadlock, starvation.
- How can it be improved?
- Idea: make sticks available only for one philosopher at a time.
- And make sure, that one half of the sticks start in a different state than the other half.

Dining Philosophers revisited (cont)

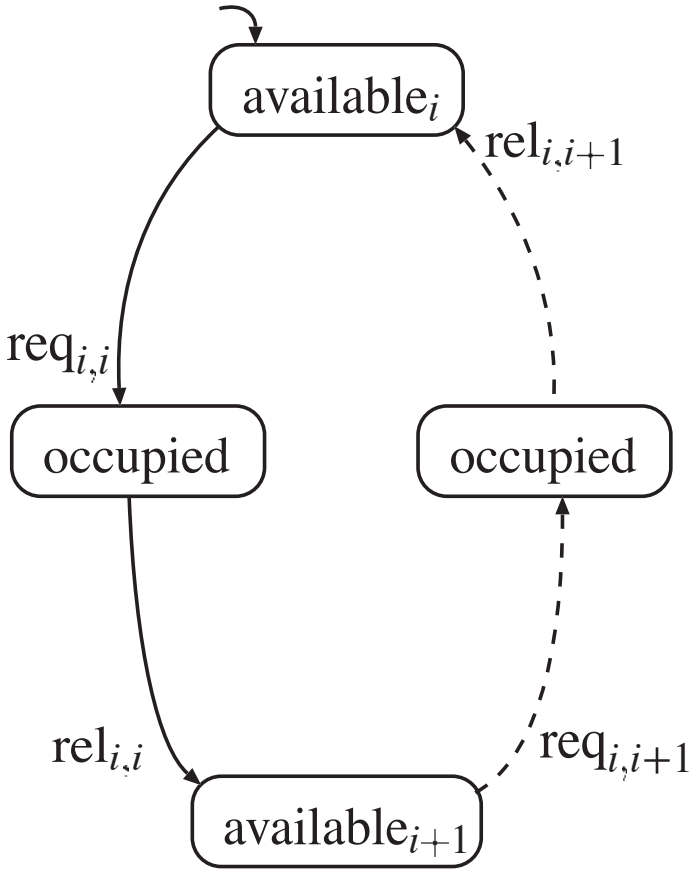


Figure 3.30: Dining Philosophers refined

3.4 State-space explosion

Input size

- Remember: input size of an integer n is $\log n$, not n.
- n is **exponential** in $\log n$.
- Similarly, the program graph is a compact representation of the underlying transition system, which is obtained by **unfolding**.
- When asking **"how complex is it to check a property of a *TS*"**, the representation of the *TS* matters.
- Size of program graph versus size of TS.

Size of unfolded transition system

What is the size of a transition system obtained from unfolding a program graph with variables $x \in \mathrm{Var}$?

- Size of **underlying domains** is important.
- $\mathrm{dom}(x)$ infinite: unfolded transition system is **infinite** and undecidability issues arise.
- $\mathrm{dom}(x)$ finite: number of states is $|\mathrm{Loc}| \ \prod_{x \in \mathrm{Var}} \ |\mathrm{dom}(x)|$.
- $\#\,\mathrm{states}$ is exponential in $\#\,\mathrm{variables}$.

Influence of $|\mathcal{P}rop|$ and $|S|$

Clearly also the **# constants** in a state plays a role. The same is true for the size of the **state space of the interleaved system**.

- $\mathcal{P}rop$ can be large, due to the allowed **conditions of the variables** in the program graph. In practice only a small number of such conditions are interesting.
- What about the labelling function π?
- Representing π explicitly is expensive. Often, this info can be derived from the state.
- State space of $TS_1 \,|||\, \ldots \,|||\, TS_n$ is cartesian product: $\Pi_{i=1}^{n} S_i$. Size is exponential in # components.
- In general, the size of the TS obtained from a program graph is exponential in the size of the program graph.

3.5 LT properties in general

LT properties of transition systems

Transition systems are abstractions from systems in the real world. We would like to

- reason about particular computations of a *TS*;
- reason about all possible executions in a *TS*,
- formulate interesting properties such a *TS* should satisfy.

Main notion: a path in a *TS*. It is a sequence of states starting in an initial state. It corresponds to an execution of the *TS* and is closely related to the notion of run introduced in Definition 3.8 on Slide 180.

Transitions systems without terminal nodes

From now on, we assume wlog that all transition systems do not have any terminal nodes. In case a *TS* has terminal nodes, we could simply add for each such node a new action and a new state (with an arrow pointing to itself) and extend the *TS* appropriately.

Definition 3.20 (Path λ (of a *TS*))

A path $\lambda : \mathbb{N}_0 \to \mathrm{S}$ in a *TS* is a sequence of states starting with an initial state such that this sequence corresponds to a **run of the *TS***.

Paths versus traces

Often, we are not so much interested in the states as such, only **what is true** in them, i.e. which propositions from $\mathcal{P}rop$ are true: $\pi(s_i)$.

Definition 3.21 (Trace of λ of a *TS*)

The **trace** of a path $\lambda : \mathbb{N}_0 \to \mathrm{S}$ in a *TS* with $\mathcal{P}rop$ is the following infinite sequence

$$\pi(\lambda(0))\pi(\lambda(1))\pi(\lambda(2))\ldots\pi(\lambda(i))\ldots$$

($\pi(\lambda(i))$ is the set of all propositions that are true in state $\lambda(i)$). We call this also an **ω-word** over $2^{\mathcal{P}rop}$.

The set of all traces of a *TS* is the set of the traces of all paths from *TS*: it is denoted by *Traces*(*TS*).

Runs or paths?

We want to define what it means that
two transition systems behave the same.

1. We take the set of **runs** of a *TS* as the defining behaviour.
2. Often the actions do not play any role, only the states do. Then we could take the set of **paths** of a *TS* as the defining behaviour.

Both possibilities rely on the **internal** behaviour, that we might not be able to determine.

Runs or paths? (cont.)

Often one cannot distinguish between certain states, we only know what is true in them (and that **depends on the language** $\mathcal{P}rop$).

3 Therefore we choose from now on the observable behaviour to describe a *TS*: the set of traces as the defining behaviour of a TS.

Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt.

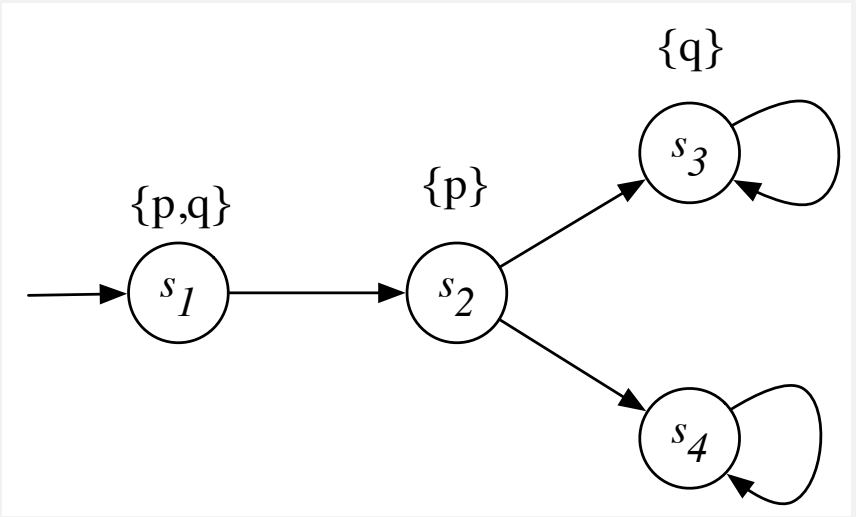
L. Wittgenstein, TLP, Satz 5.6

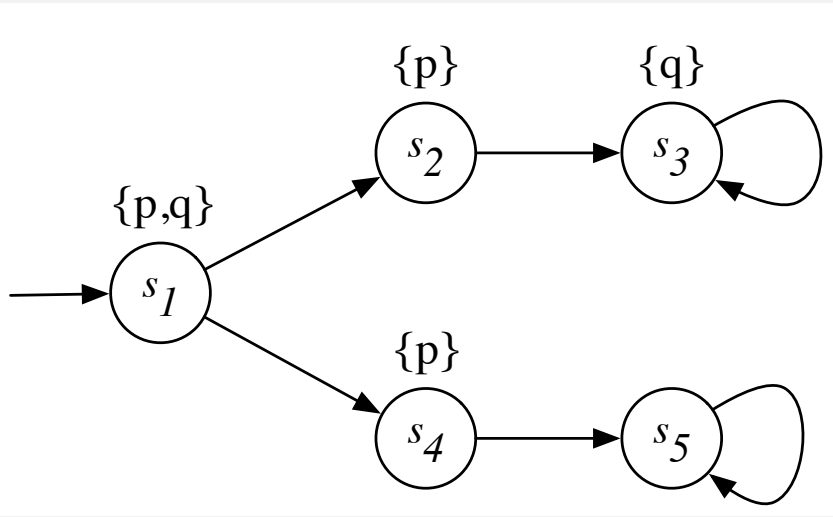
Definition 3.22 (Equivalence of transition systems)

Let TS_1 and TS_2 be two transition systems over $\mathcal{P}rop$ and let $A \subseteq \mathcal{P}rop$. We say that

1. TS_1 and TS_2 are A-equivalent, *if, by definition,* $\text{Traces}(TS_1)\,|_A = \text{Traces}(TS_2)\,|_A$,
2. TS_1 is a correct implementation of TS_2, *if, by definition,* $\text{Traces}(TS_1) \subseteq \text{Traces}(TS_2)$.

Example 3.23 (Two transition systems TS_1, TS_2)





What are the paths, traces of them?

1. $s_1 s_2 s_3^\omega$, $s_1 s_2 s_4^\omega$. $\{p,q\}\{p\}\{q\}^\omega$, $\{p,q\}\{p\}\{\}^\omega$
2. $s_1 s_2 s_3^\omega$, $s_1 s_4 s_5^\omega$. $\{p,q\}\{p\}\{q\}^\omega$, $\{p,q\}\{p\}\{\}^\omega$.
3. Both transition systems have the same set of traces. **Is there any difference between them?**

Properties of transition systems

- "Whenever p holds, a state with q is reachable."
- Obviously this is true in TS_1 but not in TS_2.
- Later: this cannot be expressed in LTL.
- Any property that is solely based on the set of traces of a TS can not distinguish between TS_1 and TS_2.
- But still many useful properties can be defined: linear time (LT) properties.
- The former two transition systems **cannot be distinguished** by any LT-property.

Properties of transition systems

Definition 3.24 (LT properties)

Given a set $\mathcal{P}rop$, a LT property over $\mathcal{P}rop$ is any subset of $(2^{\mathcal{P}rop})^\omega$.
A transition system *TS* satisfies such a property, if all its traces are contained in it.

- A LT property is a, possibly infinite, set of infinite words.

Attention

$\{p,q\}\{p\}\{q\}^\omega$ really means the infinite word $\{p,q\}\{p\}\{q\}\ldots\{q\}\ldots$, not $\{p\}\{p\}\{q\}\ldots\{q\}\ldots$, or $\{q\}\{p\}\{q\}\ldots\{q\}\ldots$ as in regular expressions.

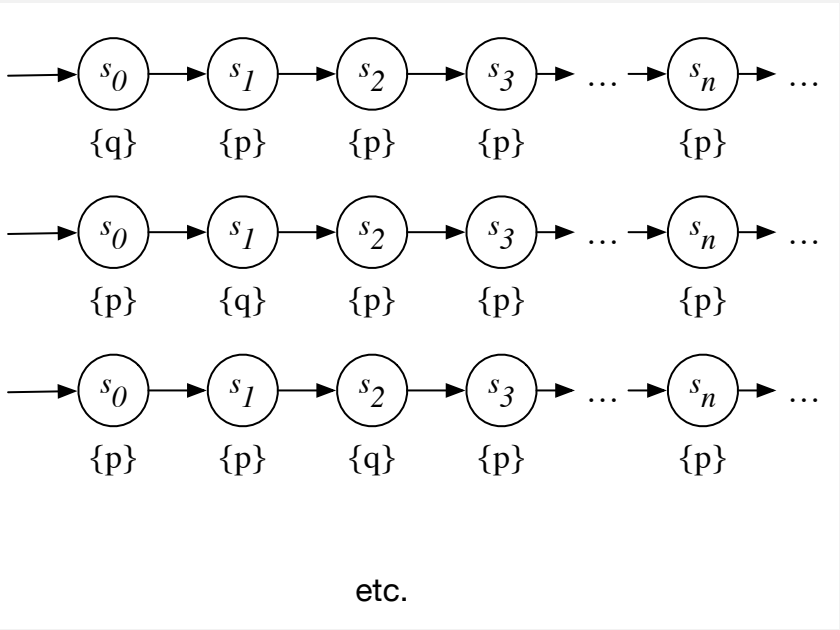
Properties of transition systems

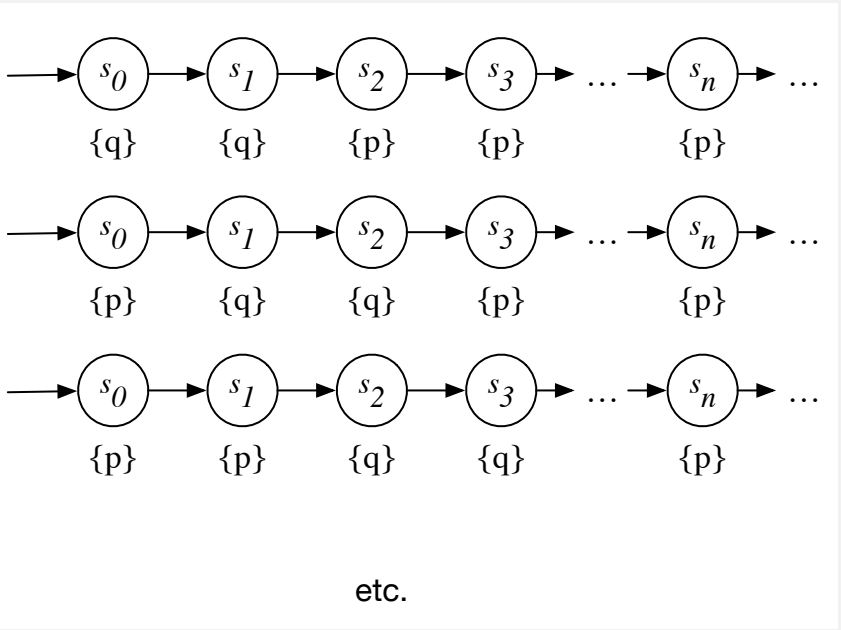
We also use $\{\{p,q\},\{p\}\}\{p\}\{q\}^\omega$ to denote the set of words that either start with $\{p,q\}$ or with $\{p\}$.

Example 3.25

1. When p then q two steps farther ahead.
2. It is never the case that p and q. This is important for mutual exclusion algorithms: one wants to ensure that never two processes are at the same time in their critical section.
3. When p then eventually q. When a message has been sent, it will eventually be received.

Example 3.26 (Sets of paths)





The first infinite set of paths M_1 can be denoted more succinctly by $\{p\}^*\{q\}\{p\}^\omega$ and the second, M_2, by $\{p\}^*\{q\}\{q\}\{p\}^\omega$ (we are using the Kleene * as in regular expressions). These expressions are called ω regular expressions.

Which properties does M_1 satisfy?

- At some time q is true and then always p holds.
- q is true exactly once.
- p is always true with one single exception, when q is true.
- **q is true exactly once (and in that state p is not true), and before and after that, only p holds true.** This will be expressed as a LTL formula shortly.

Can M_1 be represented as a (finite) transition system at all?

Theorem 3.27 (Traces and LT-properties)

Let TS_1 and TS_2 be two transition systems over $\mathcal{P}rop$. Then the following are equivalent:

- $Traces(TS_1) \subseteq Traces(TS_2)$
- *for all LT properties M: if TS_2 satisfies M, so does TS_1.*

Corollary 3.28

TS_1 and TS_2 satisfy the same LT properties if and only if $Traces(TS_1) = Traces(TS_2)$.

More interesting properties

For the formal specification and verification of concurrent and distributed systems, the following **useful concepts can be formally, and concisely, specified** as LT properties (and later also using temporal logics):

- safety properties,
- liveness properties,
- fairness properties.

Safety Properties

Many safety properties are quite simple, they are just conditions on the states and called invariants. They have the form ($A_i \subset 2^{\mathcal{P}rop}$)

$$M = \{A_0 A_1 \ldots A_i \ldots : \text{ for all } i: \ A_i \models \varphi\}$$

for a **propositional formula** φ.

Examples are

- **mutual exclusion properties**: $\neg \text{crit}_1 \vee \neg \text{crit}_2$,
- **deadlock freedom**: $\neg \text{wait}_0 \vee \ldots \vee \neg \text{wait}_5$. Deadlock freedom does not imply a fair distribution, i.e. $\neg \text{wait}_0$ can always hold.

Others require **conditions on finite fragments**, for example a traffic light with three phases requiring an **orange phase immediately before a red phase**. This is not an invariant.

Safety Properties (cont.)

Definition 3.29 (Safety property)

A LT property M_{safe} is called a safety property, if for all words $\lambda \in (2^{\mathcal{P}rop})^\omega \setminus M_{\text{safe}}$ there exists a finite prefix (a bad prefix) $\hat{\lambda}$ of λ such that

$$M_{\text{safe}} \cap \{\lambda' : \lambda' \in (2^{\mathcal{P}rop})^\omega, \hat{\lambda} \text{ is a finite prefix of } \lambda'\} = \emptyset$$

So it is a **condition on a finite initial fragment**: no extended word resulting from such a bad prefix is allowed.

Liveness Properties

Definition 3.30 (Liveness property)

A LT property M is a liveness property, if the set of finite prefixes of the elements of M is identical to $(2^{\mathcal{P}rop})^*$. I.e. each finite prefix can be extended to an infinite word that satisfies the property.

- Each process will **eventually** enter its critical section.
- Each process will enter its critical section **infinitely** often.
- Each waiting process will **eventually** enter its critical section.

Liveness Properties (cont.)

Starvation freedom in the dining philosophers is a typical example: each philosopher is getting her sticks infinitely often.

Starvation freedom: For all timepoints i, if there is a waiting process at time i, then the process gets into its critical section eventually.

Safety versus Liveness

- Are safety properties also liveness properties? Vice versa?
- There is only one property that is both: $(2^{\mathcal{P}rop})^{\omega}$, i.e. the trivial property that contains all paths.

Theorem 3.31 (LT properties as intersections)

Each LT-property can be represented as the intersection of a safety with a liveness property.

But there are LT properties that are neither safe nor live.

Fairness Properties

Definition 3.32 (Fairness property)

A LT property is a fairness property, if one of the following applies:

Each process gets its turn infinitely often

provided that

unconditional: (no restrictions)

strong: it is enabled infinitely often,

weak: it is continuously enabled from a certain time on.

LT properties used in practice

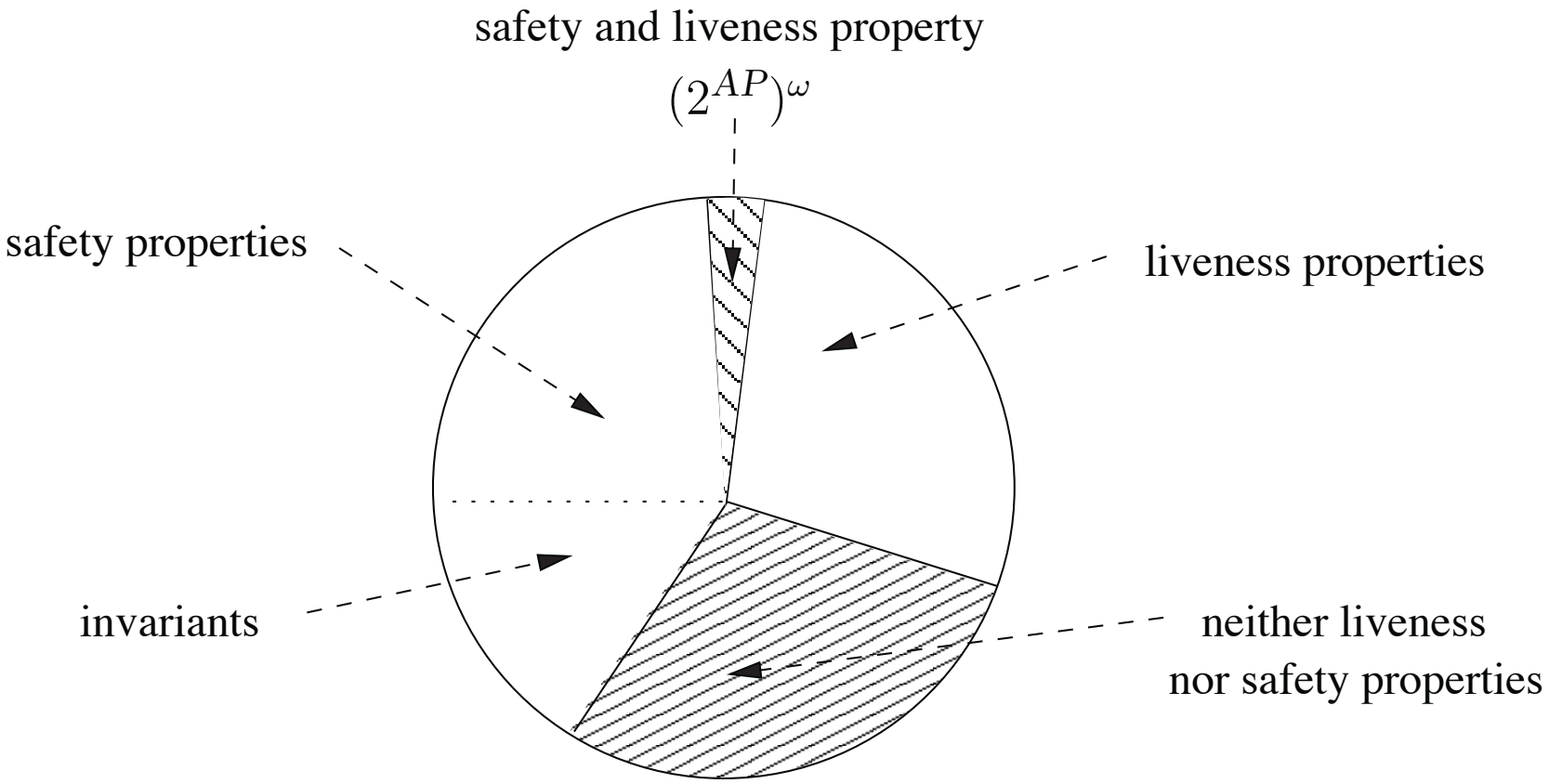


Figure 3.31: Overview LT-properties

3.6 LTL

Typical temporal operators

$\mathbf{X}\varphi$	φ is true in the neXt moment in time
$\mathbf{G}\varphi$	φ is true Globally: in all future moments
$\mathbf{F}\varphi$	φ is true Finally: eventually (in the future)
$\varphi \mathcal{U} \psi$	φ is true Until at least the moment when ψ becomes true (and this eventually happens)

$$\mathbf{G}((\neg \text{passport} \vee \neg \text{ticket}) \rightarrow \mathbf{X}\neg \text{board\_flight})$$

$$\text{send}(\text{msg}, \text{rcvr}) \rightarrow \mathbf{F}\text{receive}(\text{msg}, \text{rcvr})$$

Safety: **Something bad will not happen, something good will always hold.**

$\mathbf{G}\neg$bankrupt,

$\mathbf{G}$fuelOK,

Usually: $\mathbf{G}\neg\ldots$.

Liveness: **Something good will happen.**

$\mathbf{F}$rich,

power_on $\rightarrow \mathbf{F}$online,

Usually: $\mathbf{F}\ldots$.

Fairness: Combinations of safety and liveness:

$\mathbf{FG}\neg$dead or

$\mathbf{G}(\text{request\_taxi} \rightarrow \mathbf{F}\text{arrive\_taxi})$.

Strong fairness: "If something is requested then it will be allocated":

$$\mathbf{G}(\text{attempt} \rightarrow \mathbf{F}\text{success}),$$

$$\mathbf{GF}\text{attempt} \rightarrow \mathbf{GF}\text{success}.$$

Scheduling processes, responding to messages, no process is blocked forever, etc.

Definition 3.33 (Language $\mathcal{L}_{LTL}$ [Pnueli 1977])

The language $\mathcal{L}_{LTL}(\mathcal{P}rop)$ is given by all formulae generated by the following grammar, where $p \in \mathcal{P}rop$ is a proposition:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathcal{U} \varphi \mid \mathbf{X}\varphi.$$

The additional operators

- **F** (eventually in the future) and
- **G** (always from now on)

can be defined as macros :

$$\mathbf{F}\varphi \equiv (\neg\Box)\mathcal{U}\,\varphi \qquad \text{and} \qquad \mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi$$

The standard Boolean connectives $\top, \wedge, \rightarrow$, and $\leftrightarrow$ are defined in their usual way as **macros** (see Definition 2.3 on Slide 64).

Models of $\mathcal{L}_{LTL}$

The semantics is given over paths, which are **infinite sequences of states** from S, and a standard labelling function $\pi : \mathrm{S} \to \mathcal{P}(\mathcal{P}rop)$ that determines which propositions are true at which states.

Definition 3.34 (Path $\lambda = q_0 q_1 q_2 q_3 \ldots$)

- A path λ over a set of states S is an infinite sequence of states. We can view it as a mapping $\mathbb{N}_0 \to \mathrm{S}$. The set of all sequences is denoted by S^ω.
- $\lambda[i]$ denotes the ith position on path λ (starting from $i = 0$) and
- $\lambda[i, \infty]$ denotes the subpath of λ starting from i ($\lambda[i, \infty] = \lambda[i]\lambda[i+1]\ldots$).

$$\lambda = q_0 q_1 q_2 q_3 \ldots \in \mathrm{S}^\omega$$

Definition 3.35 (Semantics of $\mathcal{L}_{LTL}$)

Let λ be a path and π be a labelling function over S. The semantics of **LTL**, $\models^{\mathbf{LTL}}$, is defined as follows:

- $\lambda, \pi \models^{\mathbf{LTL}} \mathrm{p}$ *if, by definition,* $\mathrm{p} \in \pi(\lambda[0])$ and $\mathrm{p} \in \mathcal{P}rop$;
- $\lambda, \pi \models^{\mathbf{LTL}} \neg\varphi$ *if, by definition, not* $\lambda, \pi \models^{\mathbf{LTL}} \varphi$ (we write $\lambda, \pi \not\models^{\mathbf{LTL}} \varphi$);
- $\lambda, \pi \models^{\mathbf{LTL}} \varphi \vee \psi$ *if, by definition,* $\lambda, \pi \models^{\mathbf{LTL}} \varphi$ **or** $\lambda, \pi \models^{\mathbf{LTL}} \psi$;
- $\lambda, \pi \models^{\mathbf{LTL}} \mathbf{X}\varphi$ *if, by definition,* $\lambda[1, \infty], \pi \models^{\mathbf{LTL}} \varphi$; and
- $\lambda, \pi \models^{\mathbf{LTL}} \varphi \mathcal{U} \psi$ *if, by definition, there is an* $i \in \mathbb{N}_0$ such that $\lambda[i, \infty], \pi \models \psi$ and $\lambda[j, \infty], \pi \models^{\mathbf{LTL}} \varphi$ for all $0 \leq j < i$.

Other temporal operators

$\lambda, \pi \models \mathbf{F}\varphi$ *if, by definition,* $\lambda[i, \infty], \pi \models \varphi$ for some $i \in \mathbb{N}_0$;

$\lambda, \pi \models \mathbf{G}\varphi$ *if, by definition,* $\lambda[i, \infty], \pi \models \varphi$ for all $i \in \mathbb{N}_0$;

Exercise

Prove that the semantics does indeed match the definitions;

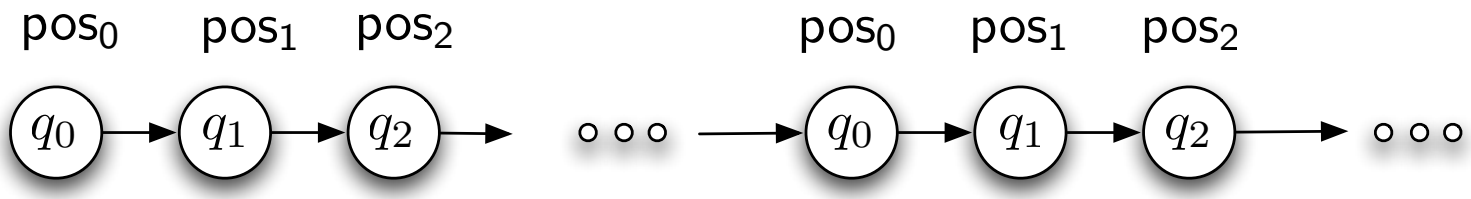
- $\mathbf{F}\varphi$ is equivalent to $(\neg\Box)\,\mathcal{U}\,\varphi$, and
- $\mathbf{G}\varphi$ is equivalent to $\neg\mathbf{F}\neg\varphi$.

Validity, satisfiability

satisfiable: a LTL formula is satisfiable, *if, by definition,* there is a model for it,

valid: a LTL formula is valid, *if, by definition,* it is true in all models,

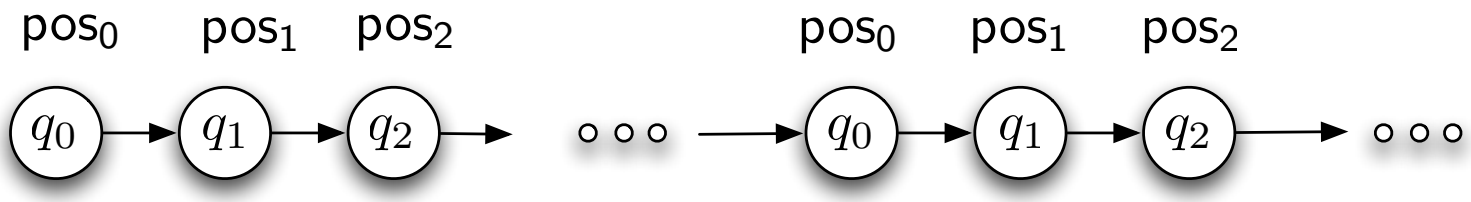
contradictory: a LTL formula is contradictory, *if, by definition,* there is no model for it.



$$\lambda, \pi \models \mathbf{F}\mathsf{pos}_1$$

$$\lambda' = \lambda[1, \infty], \pi \models \mathsf{pos}_1$$

$$\mathsf{pos}_1 \in \pi(\lambda'[0])$$



$\lambda, \pi \models \mathbf{GF}\mathsf{pos}_1$ *if and only if*

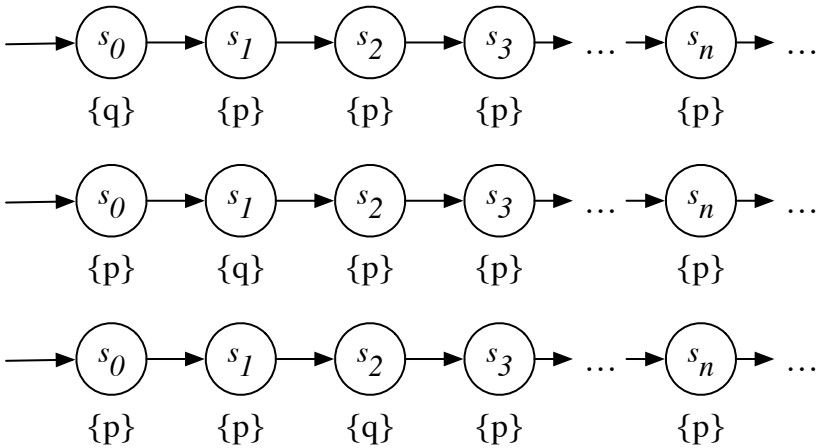
$\lambda[0, \infty], \pi \models \mathbf{F}\mathsf{pos}_1$ and
$\lambda[1, \infty], \pi \models \mathbf{F}\mathsf{pos}_1$ and
$\lambda[2, \infty], \pi \models \mathbf{F}\mathsf{pos}_1$ and
$\ldots$

Paths and infinite sets of paths

- Paths are infinite entities, and so are infinite sets of them.
- They are both theoretical constructs.
- In order to work with them we need a finite representation:
- namely transition systems (also called pointed Kripke structures).

A set of paths (1)

We reconsider the paths $\lambda_0, \lambda_1, \ldots, \lambda_i, \ldots$ from Example 3.26 on Slide 238:



- $q \wedge \neg p \wedge \mathbf{XG}(p \wedge \neg q)$
- $p \wedge \mathbf{X}q \wedge \mathbf{XXG}p$
- $p \wedge \mathbf{X}p \wedge \mathbf{XX}q$

etc.

Can we distinguish between them (using LTL)?

Indistinguishable paths

Observation

While any two paths can be distinguished by appropriate LTL formulae, these formulae get more and **more complicated**: operators need to be nested.

- The first two paths can be distinguished by just propositional logic, no LTL connectives are needed.
- But the second and third cannot: we need **X**.
- For the third and fourth we need a nesting **XX**.

By induction over the structure of φ

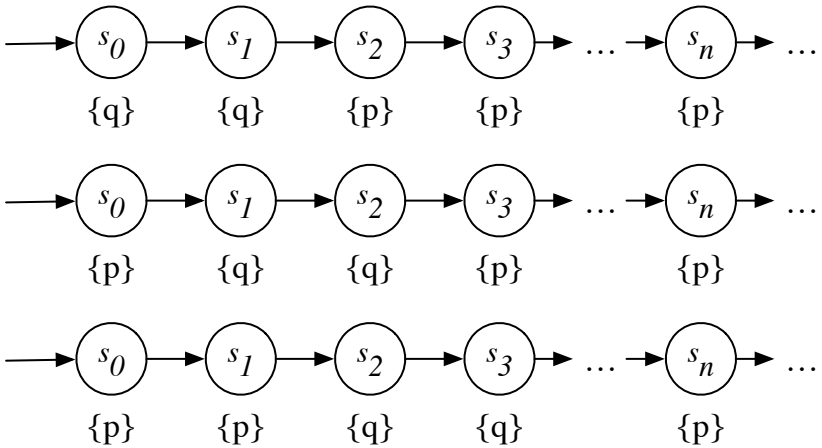
Given any LTL formula φ, there is a $i_0 \in \mathbb{N}$ such that for all $i, j \geq i_0$: $\lambda_i \models \varphi$ if and only if $\lambda_j \models \varphi$.

A set of paths (2)

- Can we find a LTL formula, or a set of LTL formulae, that **characterize exactly the whole set of paths**?
- So what holds true in all of these paths?
- $p \mathcal{U} q$. But this is also true in other paths not listed above.
- $(p \wedge \neg q) \mathcal{U} (q \wedge \neg p)$. Again, this is also true in other paths.
- $(p \wedge \neg q) \mathcal{U} (q \wedge \neg p \wedge \mathbf{XG}(p \wedge \neg q))$. That is it. This describes exactly the set of paths above.

Another set of paths (1)

We reconsider the second set of paths from Example 3.26 on Slide 238:



etc.

- $q \wedge \mathbf{X}q \wedge \mathbf{XXG}(p \wedge \neg q)$
- $p \wedge \mathbf{X}q \wedge \mathbf{XX}q \wedge \mathbf{XXXG}p$
- etc.

What holds true in exactly these paths?

$$(p \wedge \neg q)\,\mathcal{U}\,(q \wedge \neg p \wedge \mathbf{X}(q \wedge \neg p) \wedge \mathbf{XXG}(p \wedge \neg q))$$

LTL formulae and transition systems: $TS \models \phi$

- Up to now we have defined LTL formulae **only for paths**.
- For a transition system *TS*, we say that an LTL formula is true in a state s, if it is true in all runs resulting from that state.
- A LTL formula is true in the whole transition system, if it is true in all runs resulting from the initial states.

LTL formulae and transition systems: $TS \models \varphi$

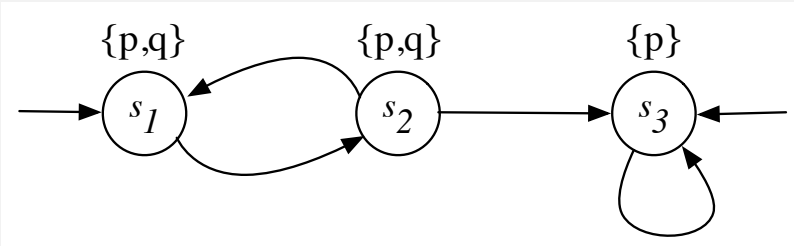
Definition 3.36 (*TS* and LTL formulae: $TS \models \phi$)

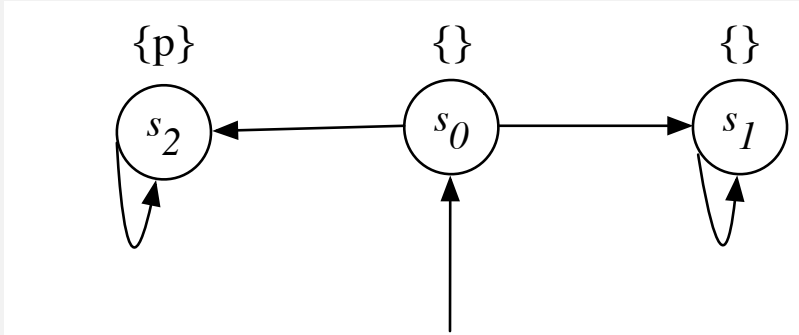
Let a *TS* and a LTL formula φ be given.

- **A** LTL **formula φ is true in a state s of *TS***, *if, by definition,* it is true in **all runs resulting from s**.
- *A* LTL formula φ is true in TS, *if, by definition,* it is true in all runs resulting from all initial states.

LTL formulae and transition systems

Example 3.37 (LTL formulae)





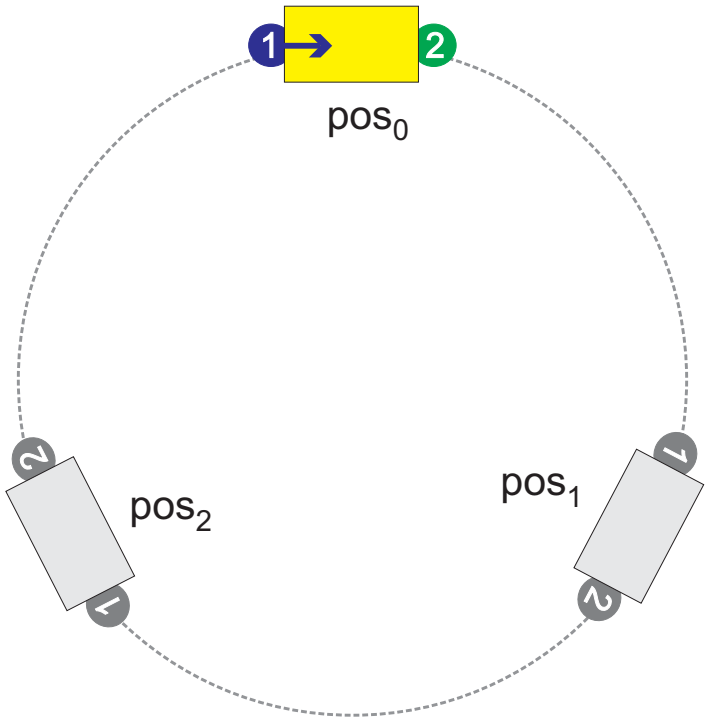
Which formulae hold true?

TS_1: $TS_1 \models \mathbf{G}p$, $TS_1 \not\models \mathbf{X}(p \wedge q)$,
$TS_1 \not\models \mathbf{G}(q \rightarrow \mathbf{G}(p \wedge q))$,
but $TS_1 \models \mathbf{G}(q \rightarrow (p \wedge q)))$

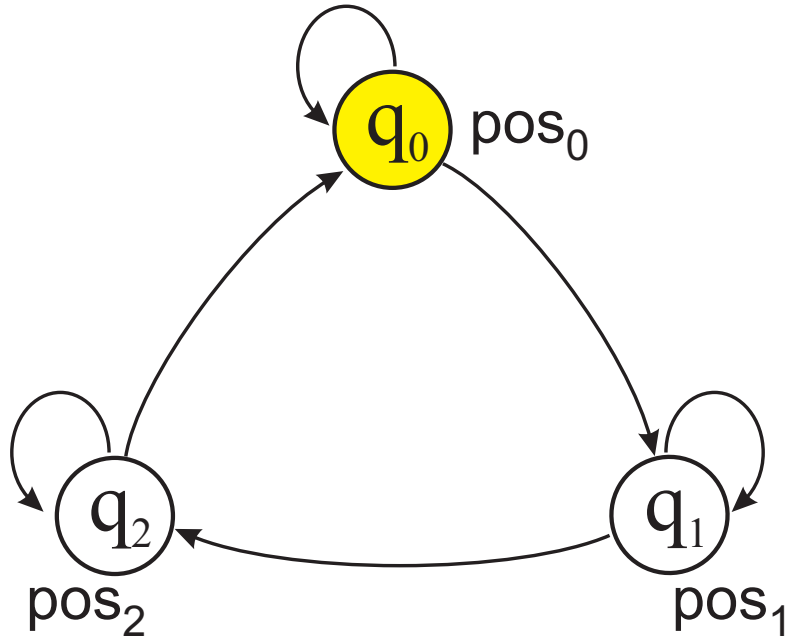
TS_2: $TS_2 \not\models \mathbf{F}p$ and $TS_2 \not\models \neg\mathbf{F}p$

From system to behavioral structure

System

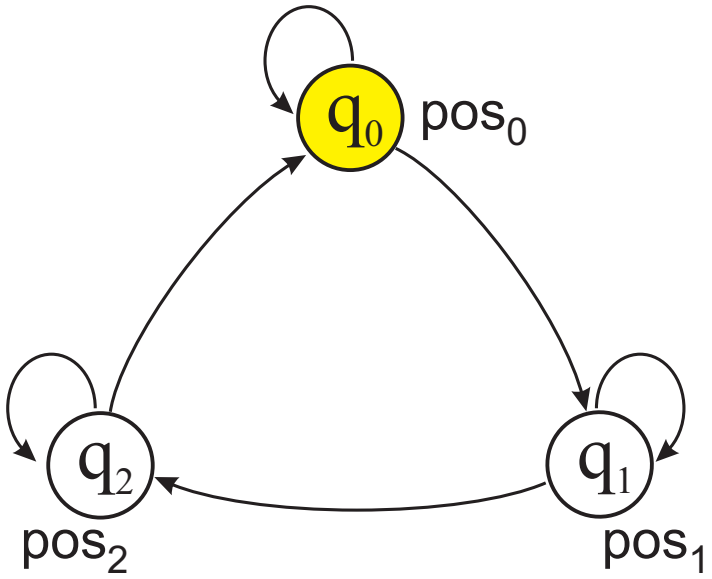


Computational str.

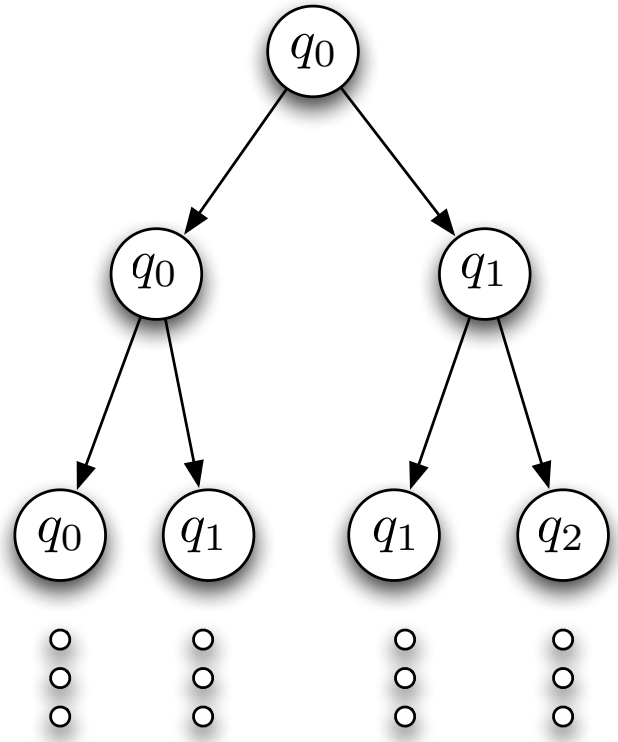


From computational to behavioral structure

Computational str.



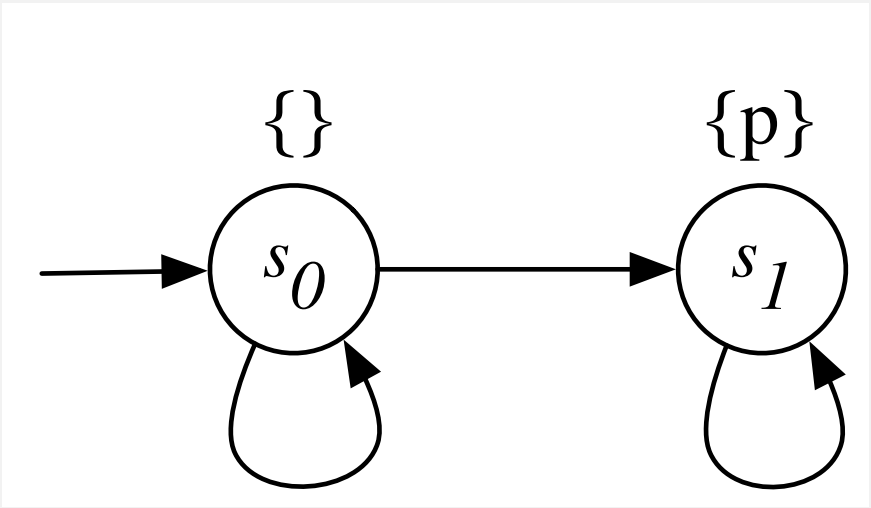
Behavioral str.



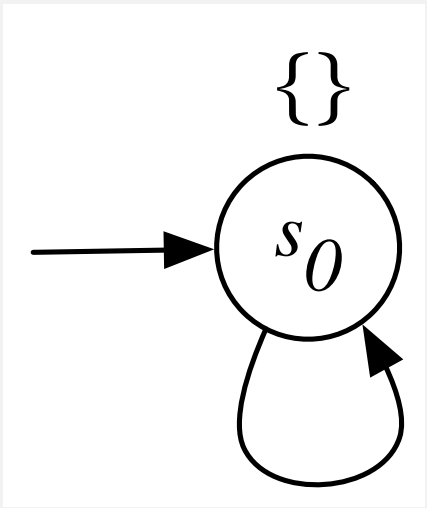
The behavioral structure is usually infinite! Here, it is an infinite tree. We say it is the q_0-unfolding of the model.

Example 3.38 (LTL indistinguishable transition systems)

TS_1:



TS_2:



Both systems can be distinguished by the property **"a state where p holds can be reached"**.

- Each trace of TS_2 is also one of TS_1: **each LTL formula true in TS_1 is also true in TS_2.**
- So the above property cannot be expressed in LTL.

LT-Properties and their expressibility

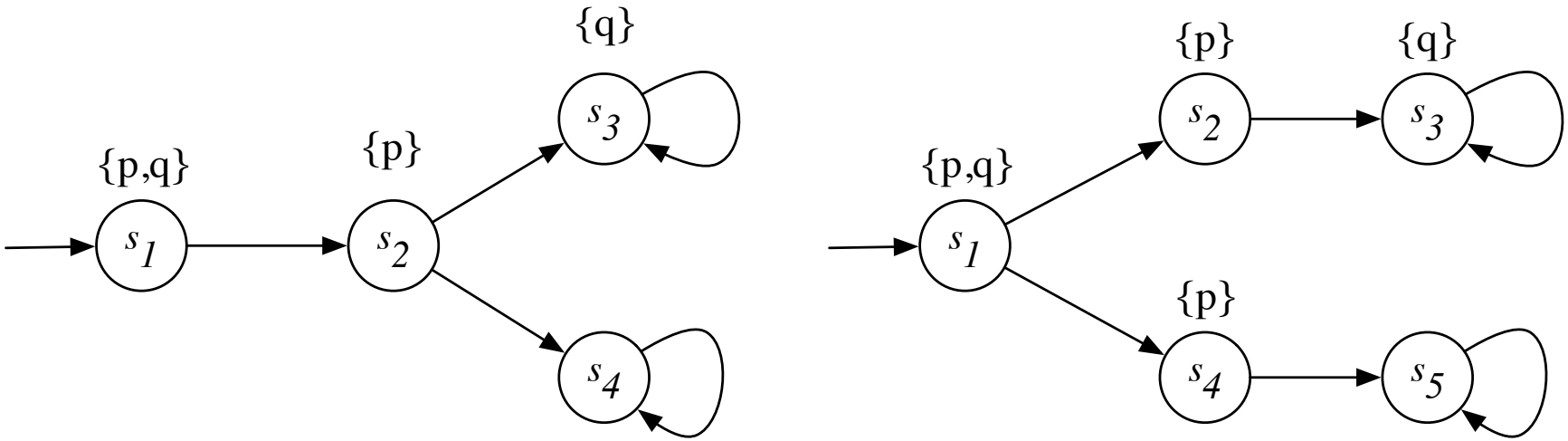
- The example on the preceding slide shows that a certain property is not a LT property.
- **This does not mean that the two transition systems cannot be distinguished.**
- In fact the formula $\mathbf{X}\neg p$ is true in TS_2 but not in TS_1.
- The traces of the two systems are also different, so both define different LT properties.

Are there TS_1 and TS_2 that **define different LT-properties but cannot be distinguished by any set of $\mathcal{L}_{LTL}$ formulae**?

Yes. $\mathcal{L}_{LTL}$ formulae express exactly "$*$-free ω-regular properties", a strict subset of "ω-regular properties", which is itself a strict subset of LT-properties.

LT-Properties and their expressibility (cont.)

We consider again Example 3.23 with the two transition systems TS_1 and TS_2.



1. The property **whenever p a state with q can be reached** distinguishes them.
2. Can this property be expressed in LTL?
3. No, because both have the same set of traces.

Some Exercises

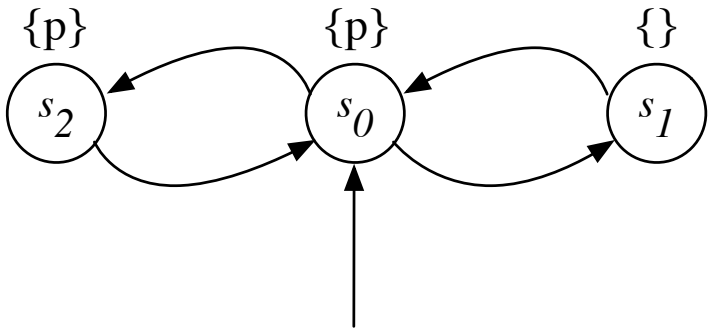
Example 3.39 (Formalizing properties in LTL)

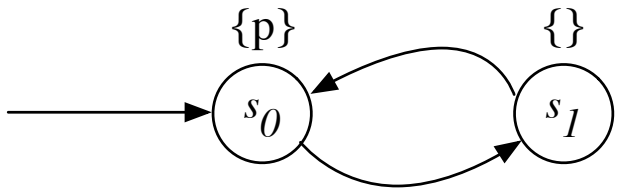
Formalise the following properties as **LTL** formulae over $\mathcal{P}rop = \{\mathsf{p}\}$

1. p should never occur.
2. p should occur exactly once.
3. At least once p is directly be followed by $\neg$p.
4. p is true at exactly all even states.

Some Exercises (cont.)

Compare the following two transition systems:





Example 3.40 (Evenness)

Formalise the following as a **LTL** formula: p is true at **all even** states (the odd states do not matter).

Does $p \wedge \mathbf{G}(p \rightarrow \mathbf{XX}p)$ work?

Satisfiability of LTL formulae

A formula is satisfiable, if there is a model (i.e. path) where it holds true. Can we **restrict the structure** of such paths? I.e. can we restrict to simple paths, for example paths that are periodic?

- If this is the case, then we might be able to **construct counterexamples** more easily, as we need only check very specific paths.
- It would be also useful to know **how long the period is** and **within which initial segment** of the path it starts, depending on the length of the formula φ.

Satisfiability of LTL formulae (cont.)

Theorem 3.41 (Periodic model theorem [SistlaClarke 1985])

*A formula $\varphi \in$ **LTL** is satisfiable if and only if there is a path λ which is ultimately periodic, and the period starts within $2^{1+|\varphi|}$ steps and has a length which is $\leq 4^{1+|\varphi|}$.*