INF5140 – Specification and Verification of Parallel Systems

Spring 2018

Institutt for informatikk, Universitetet i Oslo

January 26, 2018

Introduction

INF5140 – Specification and Verification of Parallel Systems

Overview, lecture 1

Spring 2018

January 26, 2018

Content

- See the homepage of the course:

`http://www.uio.no/studier/emner/matnat/ifi/INF5140/v18/`

Evaluation System

1. Two (small) mandatory assignments
   - Alternative: Write a research report (paper) on a topic related to the course (specification and model checking)
2. Paper presentation on related topics
3. Oral exam

- The mandatory assignments (as usual) give you the right to take the exam
- A minimum will be required on every item above in order to be approved (e.g. you must present a paper)

Remarks

- We will give you more precise guidelines during the course
- Check the web page regularly.

Formal Methods

Outline

Compute the value of a_{20} given the following definition[1]

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

[1] Thanks to César Muñoz (NASA, Langley) for providing the example.

A Java Implementation

```java
public class Mya {

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

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

The Solution (?)

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

In fact, the value of a_n for any $n \geq 0$ may be computed by using the following expression:

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

Where

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

We get then

$$a_{20} \approx 6$$

Correctness

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

Examples:

- **System:** The previous Java program computing a_n
  Requirement: For any $n \geq 0$, the program should be conform with the previous equation ($lim_{n \to \infty}\ a_n = 6$)
- **System:** A telephone system
  Requirement: If user A want to call user B (and has credit)), then eventually A will manage to establish a connection
- **System:** An operating system
  Requirement: A deadly embrace[2] (nowaday's aka *deadlock*) will never happen

[2] A deadly embrace is when two processes obtain access to two mutually dependent shared resources and each decide to wait indefinitely for the other.

How to guarantee correctness? Is it possible at all?

- How to show a system is correct?
  - It is not enough to show that it can meet its requirements
  - We should show that a system cannot fail to meet its requirements

Dijkstra (1972) on testing

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

Dijkstra (1965) on proving programs correct

"One can never guarantee that a proof is correct,[a] the best one can say is: 'I have not discovered any mistakes"

[a]One may debate that.

  - What about automatic proof? It is impossible to construct a general proof procedure for arbitrary programs[3]
- Any hope? In some cases it is possible to mechanically verify correctness; in other cases . . . we try to do our best

[3]Undecidability of the halting problem, by Turing.

What is validation?

- In general, validation is the process of checking if something satisfies a certain criterion
- Do not confuse validation with *verification*[4]

The following may clarify the difference between these terms:

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

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

[4] Some authors define verification as a validation technique, others talk about V & V –Validation & Verification– as being complementary techniques.

Usual approaches for validation

The following techniques are used in industry for validation:

- Testing
  - Check the actual system rather than a model
  - Focused on sampling executions according to some coverage criteria – Not exhaustive (“coverage”)
  - often informal, formal approaches exist (MBT)
- Simulation
  - A model of the system is written in a PL, which is run with different inputs – Not exhaustive
- Verification
  - “Is the process of applying a manual or automatic technique for establishing whether a given system satisfies a given property or behaves in accordance to some abstract description (specification) of the system”[5]

[5] From Peled’s book [Peled, 2001]

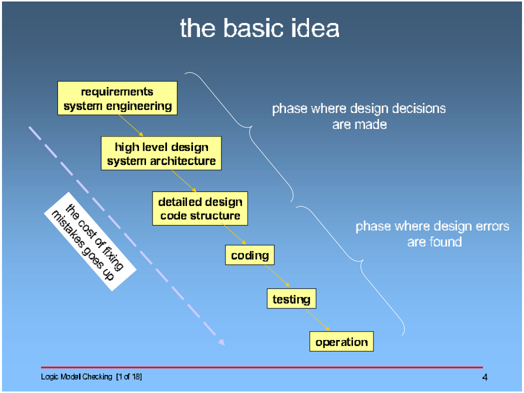
Sources of errors

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

- Specification errors (incomplete or wrong specification)
- Transcription from the informal to the formal specification
- Modeling errors (abstraction, incompleteness, etc.)
- Translation from the specification to the actual code
- Handwritten proof errors
- Programming errors
- Errors in the implementation of (semi-)automatic tools/compilers
- Wrong use of tools/programs
- ...

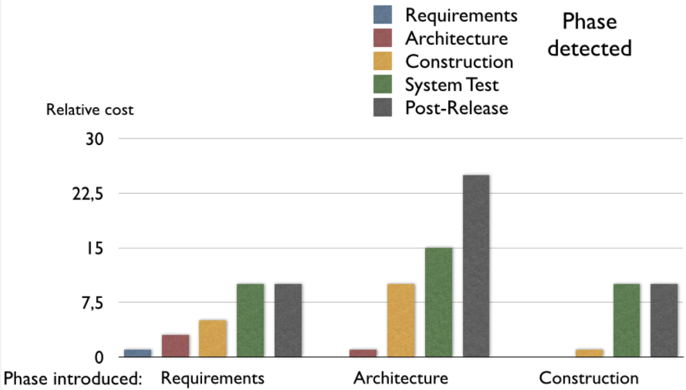
Source of errors

Most errors, however, are detected quite late on the development process[6]



[6]Picture borrowed from G.Holzmann's slides (http://spinroot.com/spin/Doc/course/index.html)

Cost of Fixing Defects



Source: McConnell, "*Code Complete*", Microsoft Press, 2004

Some (in-)famous software bugs [Garfinkel, 2005][a]

[a]Source: Garfinkel's article "History' worst software bugs"

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

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

Some (in-)famous software bugs[a]

[a] Source: Garfinkel's article "History' worst software bugs"

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

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

Some (in-)famous software bugs[a]

[a]Source: Garfinkel's article "History' worst software bugs"

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

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

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

What are formal methods?

FM [Peled, 2001]

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

[a] From D. Peled's book "Software Reliability Methods"

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

- The term verification is used in different ways
  - Sometimes used only to refer the process of obtaining the formal correctness proof of a system (deductive verification)
  - In other cases, used to describe any action taken for finding errors in a program (including model checking and testing)
  - Sometimes testing is not considered to be a verification technique

We will use the following definition (reminder):

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

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

Limitations

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

Any advantage?

Of course

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

Using formal methods

Used in different stages of the development process, giving a classification of formal methods[7]

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

[7] Testing is sometimes including as a formal method if based on a formal methodology.

Formal specification

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

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

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

Proving properties about the specification

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

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

Example

a should be true for the first two points of time, and then oscillates

-
  - First attempt:

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

    INCORRECT! – The error may be found when trying to prove some properties
  - "Correct" (?) specification:
    $a(0) \wedge a(1) \wedge \forall t \geq 0 \cdot a(t+2) = \neg a(t+1)$

Formal synthesis

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

Example: program extraction

1. specify the operational semantics of a programming language in a constructive logic (calculus of constructions)
2. prove the correctness of a given property w.r.t. the operational semantics (e.g. in Coq)
3. extract an *ocaml* code from the correctness proof (using Coq's extraction mechanism)

Verifying specifications w.r.t. implementations

Mainly two approaches:

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

A few success stories

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

Classification of systems

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

- Different kind of systems and not all methodologies/techniques may be applied to all kind of systems
- Systems may be classified depending on [Schneider, 2004]: [8]
  - Their architecture
  - The type of interaction

[8] Here we follow Klaus Schneider's book "Verification of reactive systems".

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

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

Taxonomy of properties

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

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

  Temporal behavior: The answer arrives in less than 40 seconds

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

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

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

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

When and which formal method to use?

- It depends on the problem, the underlying system and the property we want to prove
  Examples:
  - Digital circuits ... (BDDs, model checking)
  - Communication protocol with unbounded number of processes.... (verification of infinite-state systems)
  - Overflow in programs (static analysis and abstract interpretation)
  - ...
- Open distributed concurrent systems with unbounded number of processes interacting through shared variables and with real-time constraints ⇒ Very difficult!!
  Need the combination of different techniques

Some formalisms for specification

An incomplete list of formalisms for specifying systems:

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

Some techniques and methodologies for verification

- Algorithmic verification
  - Finite-state systems (model checking)
  - Infinite-state systems
  - Hybrid systems
  - Real-time systems
- Deductive verification (theorem proving)
- Abstract interpretation
- Formal testing (black box, white box, structural, ...)
- Static analysis

Summary

- Formal methods are useful and needed
- Which FM to use depends on the problem, the underlying system and the property we want to prove
- In real complex systems, only part of the system may be formally proved and no single FM can make the task

- Our course will concentrate on
  - Temporal logic as a specification formalism
  - Safety, liveness and (maybe) fairness properties
  - SPIN (LTL Model Checking)
  - Few other techniques from student presentation (e.g., abstract interpretation, CTL model checking, timed automata)

Ten Commandments of formal methods

From "Ten commandments revisited" [Bowen and Hinchey, 2005]

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

Further reading

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

- Klaus Schneider: Verification of reactive systems, 2003. Springer. Chap. 1 [Schneider, 2004]
- G. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming, 2000. Addison Wesley. Chap. 2 [Andrews, 2000]
- Z. Manna and A. Pnueli: Temporal Verification of Reactive Systems: Safety, Chap. 0[9] [Manna and Pnueli, 1992]

[9] This chapter is also the base of lectures 3 and 4.

INF5140 – Specification and Verification of Parallel Systems

Logics, lecture 2

Spring 2018

January 26, 2018

Logic is “the” specification language for us.[10]

There are many logics to choose from.
Today we see two of them:

- First-order logic (FOL) can be used to describe the state of a program.
- Modal logic can be used to describe the change of state of a program.

Other logics that we will see in other lectures:

- Temporal logics has features not available in FOL like possibility to describe sequences of states.
- Hoare logic is specially designed to reason about (imperative) programs.
- Dynamic logics: more expressive than Hoare logic, more abstract constructs and is more in the tradition of modal logic.

[10] Note: there is no such thing as “*the* logics”. There are many . . .

First-order logic

Language

The symbols of our first-order language are

- *variables* (a countable set of them $V = \{x, y, \dots\}$)
- *relation symbols* $\mathcal{P} = \{P, Q, \dots\}$ of varying arity (incl. $\doteq$ of arity 2)
- *function symbols* $\mathcal{F} = \{f, g, \dots\}$ of varying arity (if the arity of f is 0 then f is called a *constant* symbols)[a]
- the propositional *connectives* $\neg$, $\vee$, $\wedge$, $\rightarrow$ and $\leftrightarrow$
- the *quantifiers* $\forall$ and $\exists$

---

[a] Cf. also the notion of *signature* in the term-rewriting talk later (by L. Tveito, 2015)

Expressions (terms)

- Variables are *atomic* expressions.
- If f is a function symbol of arity n, and $t_1, \ldots, t_n$ are terms, then the following is also an expression.

$$f(t_1, \ldots, t_n)$$

If $n = 0$, f is a *constant*.

Example

Using infix notation, the following are expressions:

x	$U \cup V$
$y - 1$	$U \cap V$
$(x + y) + z$	$U \setminus V$

Atomic formulae

- $\top$ (top) and $\bot$ (bottom) are atomic formulae.
- If P is a relation symbol of arity n, and $t_1, \dots, t_n$ are terms, then the following is an atomic formulae.

$$P(t_1, \dots, t_n)$$

Example

Using infix notation, the following are atomic formulae.

$$\begin{array}{ll} \top & x \in U \\ x < y + 1 & U \subseteq V \\ x \doteq x - 1 & U \cap V \doteq \emptyset \end{array}$$

Boolean formulae

- All atomic formulae are boolean formulae.
- If φ and ψ are boolean formulae, so are the following.

$$\neg\varphi \qquad (\varphi \vee \psi) \qquad (\varphi \wedge \psi) \qquad (\varphi \rightarrow \psi) \qquad (\varphi \leftrightarrow \psi)$$

Example

Some examples of Boolean formulas are:

$$\neg\neg\top$$
$$\neg(x < y + 1) \rightarrow \bot$$
$$P \rightarrow (Q \rightarrow P)$$

FO formulae

First-order formulae

- All boolean formulae are first-order formulae.
- Let x be a variable. If φ is a first-order formulae, so are the following.

$$(\exists x)\varphi \qquad (\forall x)\varphi$$

- If φ and ψ are first-order formulae, so are the following.

$$\neg\varphi \qquad (\varphi \vee \psi) \qquad (\varphi \wedge \psi) \qquad (\varphi \rightarrow \psi) \qquad (\varphi \leftrightarrow \psi)$$

- $\mathcal{L}$ denotes the set of first-order formulae.

Example

$Q(y) \vee (\forall x)P(x)$
$(\forall x)(\forall y)(x < y \rightarrow (\exists z)(x < z \wedge z < y))$

First-order model

Definition

A model is a pair $M = (D, I)$, such that

- D is a non-empty set (the *domain*)
- I is mapping (the *interpretation*), such that
  - $f^I : D^n \to D$ for every function symbol f of arity n
  - $P^I \subseteq D^n$ for every relation symbol P of arity n

Observation

- We will assume an implicit model, whose domain will include the natural numbers and sets of natural numbers, and it will be obvious what function and relation symbols should be mapped to.
- E.g.: if $+$ is a function symbol $+^I$ is the addition function on the natural numbers, and $\doteq$ is mapped to a suitable $=$.
- Simplification here: no "sorts" or "types": only one sort $\Rightarrow$ only one domain. Normally: *many-sorted*

Valuation / state

Given a model

Definition (Valuation)

A *valuation* s over a set of variables V is a mapping from V to D.

- other names: variable assignment
- here, in the context of using logics to speak about programs, where variables in the formula may refer to *program variables:* we will often call a valuation a state

Example

Let $V = \{x, y, z\}$, let x and z be variables of type natural number, and y a variable of type "set of natural numbers".

- $s(x) = 256$
- $s(y) = \{1, 2, 3\}$
- $s(z) = 512$

Definition

To every FOL expression t we associate a value $s(t)$ from the domain D in a homomorphic way:

$$s(f(t_1, \dots, t_n)) = f^I(s(t_1), \dots, s(t_n))$$

Example

$$\begin{aligned} s((2 * x) + z) &= s([2 * x]^I) +^I s(z) \\ &= (s(2^I) *^I s(x)) +^I s(z) \\ &= (2 * s(x)) + s(z) \\ &= (2 * 256) + 512 \\ &= 1024 \end{aligned}$$

Free and bound variable occurrences

Definition

- A variable *occurrence* is free in a formula if it is not within the scope of a quantifier. A variable occurrence that is not free is bound.
- Let s_1 and s_2 be states over V, and $x \in V$. s_2 is an x-variant of s_1 if

  $$s_1(y) = s_2(y) \text{ for all } y \in V \setminus \{x\}.$$

  Thus, x is the only variable the states disagree on.

Definition (Substitution)

- Let φ be a first order formula, x a variable and t an expression.
- Then $\varphi[t/x]$ is φ, only with every free occurrence of the x replaced with t.

- Note, the same definition is also used in the lecture about term rewriting (used on terms, not on general FOL formula, but it's "the same".)
- Some other notation has been used like $\varphi_{x \leftarrow c}$. The one used here is the (most) standard one.
- A really exact definition would have to cater for situations like $(\forall x. x + y = 19)[x + 1/y]$.

Example

$$\varphi = (\forall x)P(x) \vee P(x)$$
$$\varphi[c/x] = (\forall x)P(x) \vee P(c)$$

Definition (Satisfaction)

We define the notion that a state formula φ is true (false) relative to a model $M = (D, I)$ in a state s, written $M, s \models \varphi$ ($M, s \not\models \varphi$) as follows.

$$
\begin{array}{lll}
M, s \models \top & \text{and} & M, s \not\models \bot \\
M, s \models R(t_1, \ldots, t_n) & \text{iff} & (s(t_1), \ldots, s(t_n)) \in R^I \\
M, s \models \neg\varphi & \text{iff} & M, s \not\models \varphi \\
M, s \models \varphi \vee \psi & \text{iff} & M, s \models \varphi \text{ or } M, s \models \psi \\
M, s \models \varphi \wedge \psi & \text{iff} & M, s \models \varphi \text{ and } M, s \models \psi \\
M, s \models \varphi \rightarrow \psi & \text{iff} & M, s \not\models \varphi \text{ or } M, s \models \psi \\
M, s \models \varphi \leftrightarrow \psi & \text{iff} & M, s \models \varphi \rightarrow \psi \text{ and } M, s \models \psi \rightarrow \varphi \\
M, s \models (\forall x)\varphi & \text{iff} & M, t \models \varphi \text{ for every } t \text{ that is an } x\text{-variant of } s \\
M, s \models (\exists x)\varphi & \text{iff} & M, t \models \varphi \text{ for some } t \text{ that is an } x\text{-variant of } s
\end{array}
$$

"Truth" and validity

Definition

- We say that φ is true in the model M, written $M \models \varphi$, if
$$M, s \models \varphi \text{ for every state } s.$$
- We say that φ is valid, written $\models \varphi$, if
$$M \models \varphi \text{ for every model } M.$$

Observation

- We will abuse this notation, and write $\models \varphi$ if φ is true in our implicit model, and refer to this as state-validity.
- For instance: $\models x + y \doteq y + x$.
- In a model where $+^I$ is the subtraction function, this will obviously not hold.

Exercises

- Model the statement: "There are infinitely many primes".

  $(\forall x)(\exists y)(x \leq y \wedge (\forall z)(z \text{ divides } y \to (z = 1 \vee z = y)))$
  where we define: z divides $y \triangleq (\exists w)(z \cdot w = y)$.
  Can define $prime(y) \triangleq (\forall z)(z \text{ divides } y \to (z = 1 \vee z = y))$

- "There is a person with at least two neighbors"

  $(\exists x, y, z)(y \neq z \wedge Neigh(x, y) \wedge Neigh(x, z))$
  where $Neigh(\cdot, \cdot)$ is a binary relation.

  Model now: "There is a person with exactly two neighbors"

  $(\exists x, y, z)(y \neq z \wedge Neigh(x, y) \wedge Neigh(x, z) \wedge$
  $((\forall w)Neigh(x, w) \to (w = y \vee w = z)))$.

- "Every even number can be written as a sum of two primes"

  $(\forall x)((even(x) \wedge x > 2) \to$
  $(\exists y, z)(prime(y) \wedge prime(z) \wedge y + z = x))$
  where the shorthand $even(x) \triangleq (\exists w)(2 \cdot w = x)$.

We assume the domain $-$ with standard $\cdot, +, >$.

Definition

A proof system for a given logic consists of

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

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

where $\varphi_1, \ldots, \varphi_n$ are premises and ψ the conclusion.

Derivations and proofs

Definition

- A derivation from a set of formulae S is a sequence of formulae, where each formula is either in S, an axiom or can be obtained by applying an inference rule to formulae earlier in the sequence.
- A proof is a derivation from the empty set.
- A theorem is the last formula in a proof.
- A proof system is
  - sound if every theorem is valid.
  - complete if evey valid formula is a theorem.
- We do not study soundness and completeness in this course.

Proof systems and proofs: remarks

- the “definitions” from the previous slides: not very formal
- in general: a proof system: a “mechanical” (= formal and constructive) way of conclusions from axioms (= “given” formulas), and other already proven formulas
- Many different “representations” of how to draw conclusions exists
- the one sketched on the previous slide
  - works with “sequences”
  - corresponds to the historically oldest “style” of proof systems (“Hilbert-style”)
  - otherwise, in that naive form: impractical (but sound & complete).
  - nowadays, better ways and more suitable for computer support of representation exists (especially using trees). For instance natural deduction style system
- for the course, those variations don’t matter.

Observation

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

$$\varphi \to (\psi \to \varphi) \tag{A1}$$

$$(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi)) \tag{A2}$$

$$((\varphi \to \bot) \to \bot) \to \varphi \tag{DN}$$

$$\frac{\varphi \quad \varphi \to \psi}{\psi} \tag{MP}$$

Let us call this logic PPL.

Note: As said, it's only one of many different ways and styles to axiomatize logic (here prop. logic)

Example

$p \to p$ is a theorem of PPL:

$(p \to ((p \to p) \to p)) \to$ $((p \to (p \to p)) \to (p \to p))$	AX2	(1)
$p \to ((p \to p) \to p)$	AX1	(2)
$(p \to (p \to p)) \to (p \to p)$	MP on (1) and (2)	(3)
$p \to (p \to p)$	AX1	(4)
$p \to p$	MP on (3) and (4)	(5)

Observation

A proof can be represented as a tree of inferences where the leaves are axioms.

Modal logics

- Modal logic: logic of "*necessity*" and "*possibility*", in that originally the intended meaning of the *modal* operators $\Box$ and $\Diamond$ was
  - $\Box\varphi$: φ is necessarily true.
  - $\Diamond\varphi$: φ is possibly true.
- Depending on what we intend to capture: we can interpret $\Box\varphi$ differently.

temporal	φ will always hold.
doxastic	I believe φ.
epistemic	I know φ.
intuitionistic	φ is provable.
deontic	It ought to be the case that φ.

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

Definition (Kripke model)

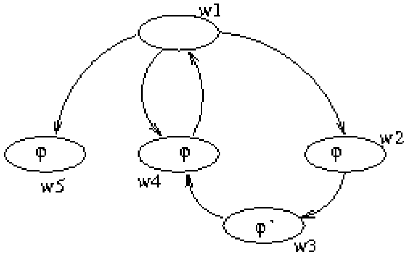
- A Kripke frame is a structure (W, R) where
  - W is a non-empty set of *worlds*, and
  - $R \subseteq W \times W$ is called the *accessibility relation* between worlds.
- A Kripke model M is a structure (W, R, V) where
  - (W, R) is a frame, and
  - $V : W \to 2^{\varphi}$ labels each world with a set of propositional variables.

Remark: some also consider propositional variables as propositional constants, propositional "symbols", it's unimportant. Kripke models are sometimes called Kripke structures.

Example

Let $M = (W, R, V)$ be the Kripke model such that

- $W = \{w_1, w_2, w_3, w_4, w_5\}$
- $R = \{(w_1, w_5), (w_1, w_4), (w_4, w_1), \dots\}$
- $V = \langle w_1 : \emptyset, w_2 : \{\phi\}, w_3 : \{\phi'\}, \dots\rangle$



Definition (Satisfaction)

A modal formula φ is true in the world w of a model M, written $M, w \models \varphi$, if:

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

But what does box and diamond intuitively "mean"?

Observation

- The semantics only differs for $\Box$ and $\Diamond$.
- We don't put any restriction on the accessibility relation R.
- The "mental picture" of what to think of $\Box$ and $\Diamond$ depends on the properties of R (and what we think R actually represent)

Different kinds of accessibility relations

Definition

A binary relation $R \subseteq W \times W$ is

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

$$(\forall a) aRa$$

- transitive if

$$(\forall abc)(aRb \wedge bRc \rightarrow aRc)$$

- euclidean if

$$(\forall abc)(aRb \wedge aRc \rightarrow bRc)$$

- total if

$$(\forall a)(\exists b)(aRb)$$

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

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

Example

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

Observation

The axioms above are said to "hold on a frame", which means, for *any* valuation and at *any* state.

Prove the double implications from the slide before!

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

Every normal modal logic has the following inference rules.

$$\frac{\varphi \text{ is a tautology instance}}{\varphi} \qquad \text{(PL)}$$

$$\frac{\varphi \quad \varphi \to \psi}{\psi} \qquad \text{(MP)}$$

$$\frac{\varphi}{\Box\varphi} \qquad \text{(G)}$$

We will only be concerned with normal modal logics.

Sample axioms for different accessibility relations

Formulae that can be used to axiomatize logics with different properties.

$$\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi) \tag{K}$$

$$\Box\varphi \to \Diamond\varphi \tag{D}$$

$$\Box\varphi \to \varphi \tag{T}$$

$$\Box\varphi \to \Box\Box\varphi \tag{4}$$

$$\neg\Box\varphi \to \Box\neg\Box\varphi \tag{5}$$

$$\Box(\Box\varphi \to \psi) \to \Box(\Box\psi \to \varphi) \tag{3}$$

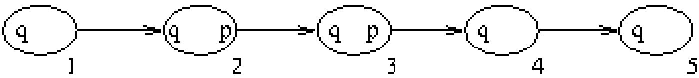
$$\Box(\Box(\varphi \to \Box\varphi) \to \varphi) \to (\Diamond\Box\varphi \to \varphi) \tag{Dum}$$

- Every normal logic has K as axiom schema.
- Observe that T implies D.

Different "flavors" of modal logic

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

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

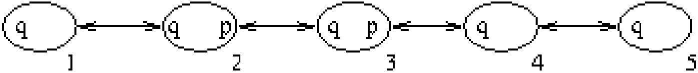


Choose the valuation $V(p) = \{2, 3\}$ and $V(q) = \{1, 2, 3, 4, 5\}$ to get the model $M = (W, R, V)$.
Which of the following statements are correct in M and why?

1.1 $M, 1 \models \Diamond\Box p$ Correct
1.2 $M, 1 \models \Diamond\Box p \to p$ Incorrect
1.3 $M, 3 \models \Diamond(q \wedge \neg p) \wedge \Box(q \wedge \neg p)$ Correct
1.4 $M, 1 \models q \wedge \Diamond(q \wedge \Diamond(q \wedge \Diamond(q \wedge \Diamond q)))$ Correct
1.5 $M \models \Box q$ Correct . . . but why?

Exercises 2 (bidirectional frames)

We call a frame (W, R) bidirectional iff $R = R_F \uplus R_P$ s.t. $\forall w, w'(wR_F w' \leftrightarrow w' R_P w)$.
i.e.: The R can be separated into two disjoind relations R_F and R_P, which one is the inverse of the other.



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

0.1 $M, 1 \models \Diamond\Box p$ Incorrect
0.2 $M, 1 \models \Diamond\Box p \rightarrow p$ Correct
0.3 $M, 3 \models \Diamond(q \wedge \neg p) \wedge \Box(q \wedge \neg p)$ Incorrect
0.4 $M, 1 \models q \wedge \Diamond(q \wedge \Diamond(q \wedge \Diamond(q \wedge \Diamond q)))$ Correct
0.5 $M \models \Box q$ Correct . . . but is it the same explanation as before?
0.6 $M \models \Box q \rightarrow \Diamond\Diamond p$

Exercises 3 (validities)

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

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

Further readings

- [Harel et al., 2000]
- [Blackburn et al., 2001]

References I

[Andrews, 2000] Andrews, G. R. (2000).
Foundations of Multithreaded, Parallel, and Distributed Programming.
Addison-Wesley.

[Blackburn et al., 2001] Blackburn, P., de Rijke, M., and Venema, Y. (2001).
Modal Logic.
Cambridge University Press.

[Bowen and Hinchey, 2005] Bowen, J. P. and Hinchey, M. G. (2005).
Ten commandments revisited: a ten-year perspective on the industrial application of formal methods.
In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 8–16, New York, NY, USA. ACM Press.

[Garfinkel, 2005] Garfinkel, S. (2005).
History's worst software bugs.
Available at `http://archive.wired.com/software/coolapps/news/2005/11/69355?currentPage=all`.

[Harel et al., 2000] Harel, D., Kozen, D., and Tiuryn, J. (2000).
Dynamic Logic.
Foundations of Computing. MIT Press.

[Manna and Pnueli, 1992] Manna, Z. and Pnueli, A. (1992).
The temporal logic of reactive and concurrent systems—Specification.
Springer Verlag, New York.

[Peled, 2001] Peled, D. (2001).
Software Reliability Methods.
Springer Verlag.

[Schneider, 2004] Schneider, K. (2004).
Verification of Reactive Systems.
Springer Verlag.