# Course Script

# IN 5110: Specification and Verification of Parallel Systems

IN5110, autumn 2019

Martin Steffen, Volker Stolz

# Contents

**Chapter 1**
**Formal methods**

**Learning Targets of this Chapter**

The introductory chapter give some motivational insight into the field of "formal methods" (one cannot even call it an overview).

**Contents**

## 1.1 Introduction

This is the "script" or "handout" version of the lecture's slides. It basically reproduces the slides in a more condensed way. For the presentation, the slides themselves are kept not too full. Additional information and explanations that are perhaps said in the classroom or whiteboard, without being reproduced on the shown slides, are shown here, as well as links and hints for further readings. In particular, *sources* and *bibliographic information* is shown mostly only here.

It's also best seen as "working dcocument", which means it will probably evolve during the semester.

## 1.2 Motivating example

**A simple computational problem**

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

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

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

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

`br/ayala/EVENTS/munoz2006.pdf` or `https://hal.archives-ouvertes.fr/ensl-00086707`. The definition or specification of it seems so simple that it's not even a "problem". It seems more like a first-semester task.

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

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

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

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

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

### The solution (?)

```
$ java mya
a(0)  = 5.5
a(2)  = 5.5901639344262435
```

```
a(4)  = 5.674648620514802
a(6)  = 5.74912092113604
a(8)  = 5.81131466923334
a(10) = 5.861078484508624
a(12) = 5.935956716634138
a(14) = 15.413043180845833
a(16) = 97.13715118465481
a(18) = 99.98953968869486
a(20) = 99.99996275956511
```

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

**Should we trust software?**

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

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

Where

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

We get then

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

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

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

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

The example can also discussed from a different angle. The slides claimed that the implementation is wrong insofar that the result should really be something like 6 (in equation (1.1)). One can figure that out with university or even school level knowledge about real analysis, series, and limits etc. However, the problem statement is really easy. Actually problems are mostly much more complex even if we stick to situations, when the problem may be specified by a bunch of equations, maybe describing some physical environment that needs to be monitored and controlled). It's unlikely to encounter a software problem whose "correct" solution can be looked-up in a beginner's textbook. What's correct anyway? In the motivational example, "math tells us the correct answer should be approximately 6", but what if the underlying math is too complex to have a simple answer to what the result is supposed to be (being unknown or even unobtainable as closed expression).

When facing a complex numerical (or computational) problem, many people nowadays would simply say "let's use a computer to caluclate the solution", basically assuming "what the computer says *is* the solution". Actually, along that lines, one could even take the standpoint that in the example, that a Java program *is* not the solution but the specification of the task. That's not so unrealistic: the program uses recursion and other things, which from some tasks perspective may be seen as quite high-level. Then the task would be, to implement a piece of hardware, or firmware or some controller, that "implement" the specification, given by some high-level recursive description in Java (or some other executable format). One can imagine that the Java program is used for *testing* whether that more low-level implementation does the right thing, like comparing results. The cautioning about "beware of numerical calculations" still applies, but the point more relevant to our lecture would be, that sometimes specifications are not so clear either, not even if they are "computer-aided". Later in the introduction, we say a program is correct only relative to a (formal) specification, but also the specifications themselves may be problematic (or the checking, even the automatic one, that the specification is satisfied.

## 1.3 How to guarantee correctness

### Correctness

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

Examples:

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

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

- **System:** A telephone system

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

A "deadly embrace" is the original term for something that is now commonly called *deadlock*. It's a classical error condition that occurs in concurrent programs, i.e., something that cannot occur in sequential program or algorithms. It occures when two processes obtain access to two mutually dependent shared resources and each decide to wait indefinitely for the other. A classical illustration is the "dining philosophers".

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

## How to guarantee correctness?

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

## Dijkstra's dictum

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

## A lesser known dictum from Dijktra (1965)

On proving programs correct: "One can never guarantee that a proof is correct, the best one can say is: 'I have not discovered any mistakes'. "

## Rest

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

The statements of Disjktra can, of course, be debated, and have been debated.

What about automatic proof? It is impossible to construct a general proof procedure for arbitrary programs. Undecidability of the halting problem, by Turing. Any hope? In some cases it is possible to mechanically verify correctness, in other cases . . . we try to do our best.

## Validation & verification

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

**Validation**

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

**Verification:**

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

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

**Approaches for validation**

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

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

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

The quote is from Peled's book [9].

overlap, no exaustive

## 1.4 Software bugs

**Sources of errors**

Errors may arise at different stages of the software/hardware development:
- specification errors (incomplete or wrong specification)
- transcription from the informal to the formal specification
- modeling errors (abstraction, incompleteness, etc.)
- translation from the specification to the actual code
- handwritten proof errors
- programming errors

- errors in the implementation of (semi-)automatic tools/compilers
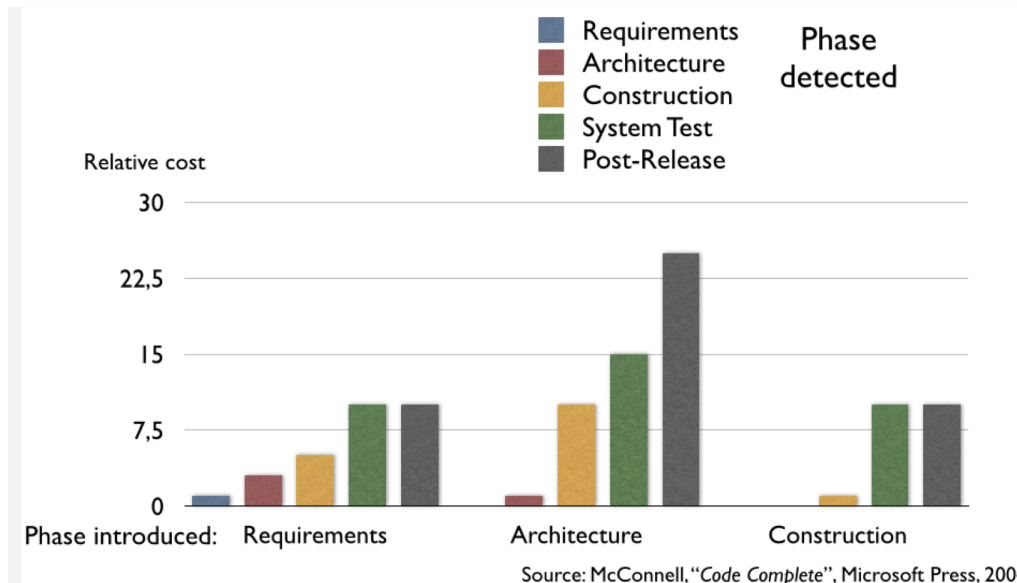- wrong use of tools/programs
- ...

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

**Errors in the SE process**
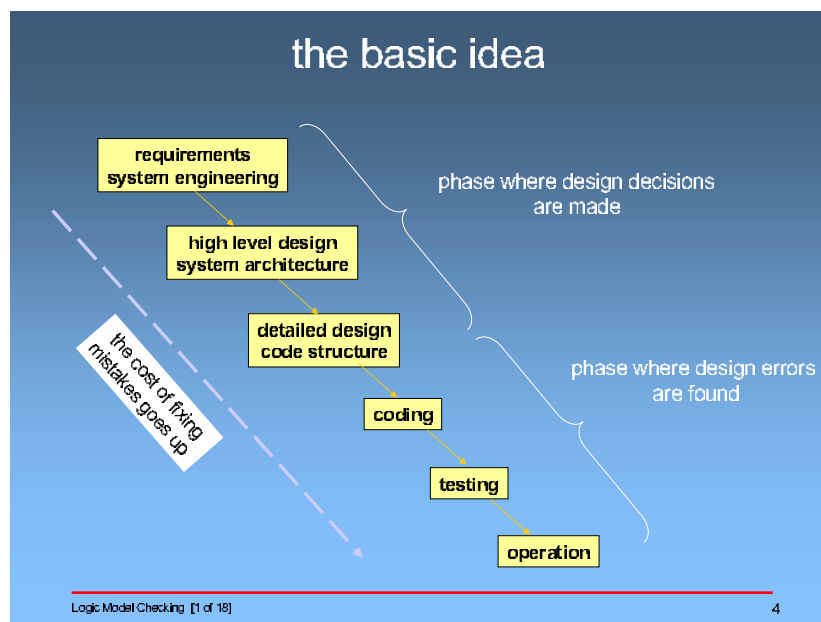


The picture borrowed from G. Holzmann's slides. Most software is developed according to some process with different phases or activities (and by diffent teams and with specific tools); often, the institution of even the legal regulators insist on some procedures etc. Many of such software engineering practices have a more or less pronounced "top-down" aspects (most pronounced in a rigid waterfall development, which, however, is more an academic abstraction, less pronouced in agile processes. Now matter how one organizes the development process, "most" errors are detected quite late on the development process, at least that's what common wisdom, experient, and empirical results show. The figure (perhaps unrealistically in simplifying manner shows a top-down process, and illustrates that certain kinds of errors (like design errors) are often detected only later. It should be clear (at least for such kind of errors), that the later the errors are detected, the more costly they are to repair.

## Costs of fixing defects



Source: McConnell, "*Code Complete*", Microsoft Press, 2004 The book the figures are taken from is [8] (a quite well-known source). The book itself attributes the shown figures to different other sources.

## Hall of shame

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

The information is taken from [4]. See also the link to that article.

**July 28, 1962: Mariner I space probe** The Mariner I rocket diverts from its intended direction and was destroyed by the mission control. 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".

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

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

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

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

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

The errors on that list are quite known in the literature (and have been analysized and discussed). Note, however, that some of those cases, the cause of the error is non uncontroversial (despite the fact of lenghty (internal) investigations and sometimes even hearings in the US congress or other external or political institutions. The list is from 2005, other (and newer) lists certainly exists. A well-known collection of computer-related problems, especially those which imply societal risks and often based on insider information is Peter Neumann's Risk forum (now hosted by ACM), which is moderated and contains reliable information (in particular speculations when the reasons are unclear are called speculations. Not all one finds on the internet is reliable, there are many folk tales on "funny" software glitches. Many may never see the public light or are openly analysed (especially when concerned with security related issues, military or financial institusions). Of course, when a space craft explodes moments after lift-off or crash-lands on Mars on live transmission from Houston, it's difficult to swipe it under the carpet. but not even then, it's easy to nail it down to the/a causing factor not to mention to put the blame somewhere or find ways to avoid it the next time. For instance, if it's determined that the ultimate cause was a missing semicolon (as some say was the case for the failuer of the Mariner mission, but see below), then how to react? Tell all NASA programmers to double-check semicolons next time, and that's it? Actually, looking more closely, one should not think of the bug as a "syntactic error".

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

The Therac-incident is very widely known. It's an example of

## 1.5 On formal methods

The slides are inspired by introductory material of the books by K. Schneider and the one by D. Peled ([10, Section 1.1] and [9, Chapter 1]).

**What are formal methods?**

**FM**

"Formal methods are a collection of notations and techniques for describing and analyzing systems" [9]

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

**Terminology: Verification**

The term *verification*: used in different ways

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

**Formal verification (reminder)**

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

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

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

**Limitations**

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

For a discussion of issues like these, one may see the papers "Seven myths of formal methods" and "Seven more myths of formal methods" ([5] [2]).

**Any advantage?**

**be modest**

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

**Rest**

- remember the *VIPER* chip

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

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

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

Maybe formal methods also suffer a bit from similar bad reputation as (sometimes) artificial intelligence have (or had). Techniques as investigated by the formal method community are opposed, ridiculed and discounted as impractical until the "disappear" and then become "common practice". So, as long as the standard practicioner does not use

something, it's "useless formal methods", once incorporated in daily use it's part of the software process and quality assurance. Artificial intelligence perhaps suffered from a similar phenomenon. At the very beginning of the digital again, when people talked about "electronic brains" (which had, compared to today, ridiculously small speed and capacity), they trumpeted that the electronic brains can "think rationally" etcetc., soon they beat humans in tic-tac-toe. The computer soon did just that, with fancy search techniques like back-tracking, branch-and-bound or what not (branch-and-bound comes from operations research). Of course the audience then said: Oh, that's not intelligence, that's just brute force and depth-first search, and nowadays, depth-first seach is taught in first semester or even school. And Tic-tac-toe is too simple, anyway, but to play chess, you need "real" intelligence. So then the AI came up with much more fancy stuff, heuristics, statistics, bigger memory, larger state spaces, but people would still state: a chess playing computer is not AI, it's "just" complex search. So, the "intelligence" those guys aim at is always the stuff that is not yet solved. Maybe the situation is similar for formal methods.

Perhaps also a parallel which has led to negative opinions like the one of Parnas is, that the community sometimes is too big-mouthed. Like promising an "intelligent electronic brain" and what comes out is a tic-tac-toe playing back-tracker... For the formal methods, it's perhaps the promise to "guarantee 100% correctness" (done based on "math") or at least perceived as to promise that. For instance, the famous dictum of Disjktra that testing cannot guarantee correctness in all cases is of course in a way a triviality (and should be uncontroversial), but it's perhaps perceived to mean (or used by some to mean) that "unlike testing, the (formal) method can guarantee that". Remember the Viper chip (a "verified" chip used in the military)

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



## Using formal methods

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

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

## Formal specification

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

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

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

K. Schneider describe only the first four classes. Peled also includes testing as part of formal methods. See Peled p. 114/115

## Proving properties about the specification

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

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

## Example

$a$ should be true for the first two points of time, and then oscillate.

## Rest

- some attempt attempt:

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

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

So perhaps a correct (?) specification might be

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

## Remarks (source)

From K. Schneider, p.5.

## Formal synthesis

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

**Example: program extraction**

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

**Verifying specifications w.r.t. implementations**

Mainly two approaches:

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

It's from K. Schneider, p.6.

**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
    - architecture

The classification here follows Klaus Schneider's book "Verification of reactive systems" [10]. Obviously, one can classify "systems" in many other ways, as well.

## Classification of systems: architecture

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

## Classification of systems: type of interaction

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

## Taxonomy of properties

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?**

Examples:

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

Open distributed, concurrent systems ⇒ *Very difficult!!*

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

## 1.6 Formalisms for specification and verification

**Some formalisms for specification**

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

– Petri nets

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

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

### Some techniques and methodologies for verification

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

The "Cleanroom" method, see Peled's Sec. 10.3

## 1.7 Summary

### Summary

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

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

## Ten Commandments of formal methods

From "Ten commandments revisited" [3]

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

## Further reading

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

- Klaus Schneider: Verification of reactive systems, 2003. Springer. Chap. 1 [10]
- G. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming, 2000. Addison Wesley. Chap. 2 [1]
- Z. Manna and A. Pnueli: Temporal Verification of Reactive Systems: Safety, Chap. 0 This chapter is also the base of lectures 3 and 4. [7]

# Bibliography

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

[2] Bowen, J. P. and Hinchey, M. G. (1995). Seven more myths of formal methods. *IEEE Software*, 12(3):34–41.

[3] 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.

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

[5] Hall, J. A. (1990). Seven myths of formal methods. *IEEE Software*, 7(5):11–19.

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

[8] McConnell, S. (2004). *Code Complete, Second Edition.* Microsoft Press, Redmond, WA, USA.

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

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