Course Script

IN 5110: Specification and Verification of Parallel Systems

IN5110, autumn 2021

Martin Steffen

Contents

Chapter 1 Formal methods

What is it about?

Learning Targets of this Chapter

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

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 but with additional comments added. The slides used in class are kept not too full. Additional information and explanations that are perhaps said in the classroom or when using the whiteboard, without being reproduced on the shown slides, are shown here, as well as links and hints for further readings. In particular, *sources* and *bibliographic information* is shown mostly only here.

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

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

1.2 Exam

The exam will be oral.

1.3 Plan

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

1.4 Motivating example

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

1.4.1 A simple computational problem

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

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

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

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

1.4.2 A straightforward implementation

```
public class Mya {

    static double a(int n) {
      if (n==0)
         return 11/2.0;
      if (n==1)
         return 61/11.0;
      return 111 - (1130 - 3000/a(n-2))/a(n-1);
    }
    public static void main(String[] argv) {
      for (int i=0;i<=20;i++)
        System.out.println("a("+i+") = "+a(i));
    }
}
```

Listing 1.1: A straightforward implementation

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

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

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

Listing 1.2: A different straightforward implementation

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

1.4.3 The solution (?)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.5 How to guarantee correctness?

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

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

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

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

1.5.1 Correctness

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

Examples:

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

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

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

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

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

1.5.2 How to guarantee correctness?

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

Dijkstra's dictum

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

A lesser known dictum from Dijktra (1965)

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

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

Dijksta's well-known dictum comes from [4]. The statements of Dijkstra can, of course, be debated, and have been debated. What about automatic proofs? It is impossible to construct a general proof procedure for arbitrary programs. It's a well-known fact that only programs in the most trivial "programming languages" can be automatically analysed (i.e., programming without general loops or recursion or if one assumes bounded memory). For clarity, one should perhaps be more precise, what can't be analysed. First of all, the undecidability of proplems refers to properties concerning the behavior or semantics of programs. Syntactic properties or similar may well be analyzed. Questions referring to the program text are typically decidable. A parser *decides* whether the source code is syntactically correct, for instance, i.e., adheres to a given (context-free) grammar. In most programming languages, type correctness is decidable (and the part of the compiler that decides on that is the type checker). What is not decidable are semantics properties of what happens when running the code. The most famous of such properties is the question whether the program terminates or not; that's known as the *halting problem*. The halting problem (due to Alan Turing) is only one undecidable property, in fact, *all* semantical questions are undecidable: every single semantical property is undecidable, with the exception of only two which are decidable. Those 2 decidable one are the 2 trivial ones, known as *true* and *false*, which hold for all programs resp. for none. The general undecidability of all non-trivial semantical properties is known as *Rice's theorem*.

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

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

1.5.3 Validation & verification

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

Validation

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

Verification:

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

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

What is a proof?

1.5.4 Approaches for validation

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

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

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

The quote is from Peled’s book [10].

1.6 Software bugs

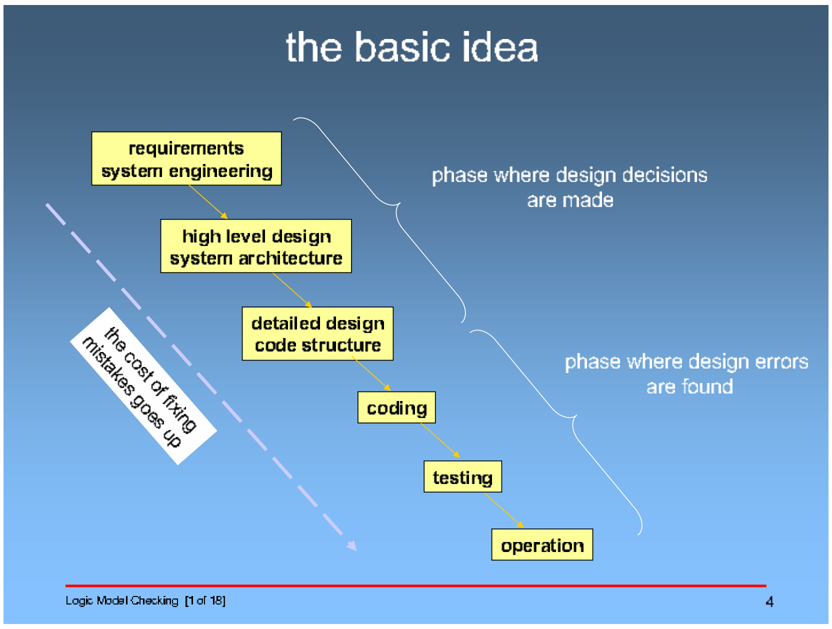
1.6.1 Sources of errors

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

- specification errors (incomplete or wrong specification)
- transcription from the informal to the formal specification
- modeling errors (abstraction, incompleteness, etc.)
- translation from the specification to the actual code
- handwritten proof errors
- programming errors
- errors in the implementation of (semi-)automatic tools/compilers
- wrong use of tools/programs
- ...

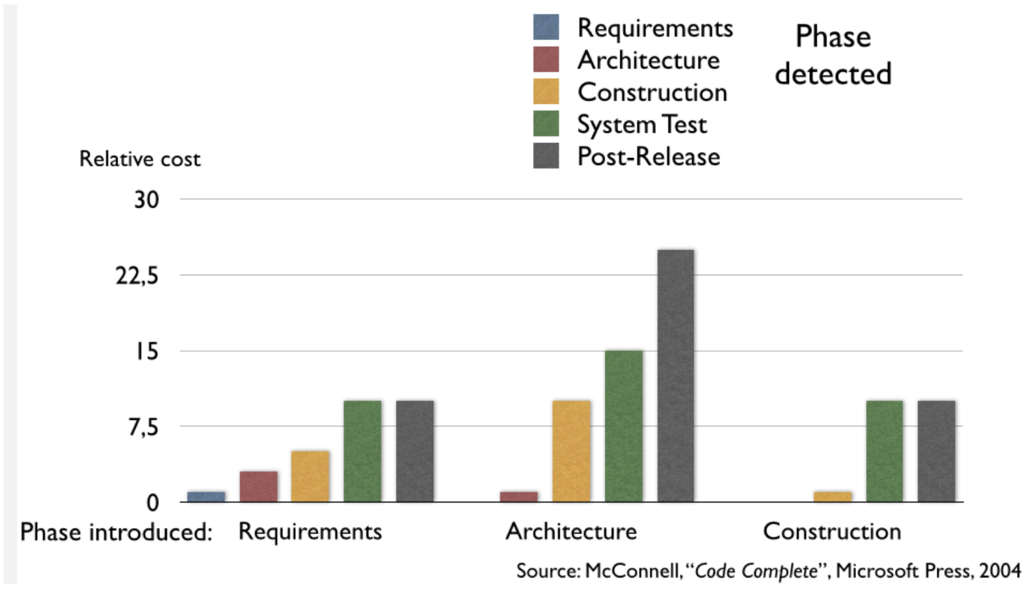
The list of errors is clearly not complete, sometimes a “system” is unusable, even if it’s hard to point with the finger to an error or a bug (is it “a bug or a feature”?). Different kinds of validation and verification techniques address different kinds of errors. Also testing as one (huge) subfield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

1.6.2 Errors in the SE process


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

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

1.6.3 Costs of fixing defects



The book the figures are taken from is [9] (a quite well-known source). The book itself attributes the shown figures to different other sources.

1.6.4 Hall of shame

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

The information is taken from [5]. See also the link to that article.

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

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

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

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

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

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

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

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

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

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

1.7 On formal methods

The slides are inspired by introductory material of the books by K. Schneider and the one by D. Peled ([12, Section 1.1] and [10, Chapter 1]).

1.7.1 What are formal methods?

FM

“Formal methods are a collection of notations and techniques for describing and analyzing systems” [10]

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

1.7.2 Terminology: Verification

The term *verification*: used in different ways

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

Formal verification (reminder)

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

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

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

1.7.3 Limitations

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

For a discussion of issues like these, one may see the papers "Seven myths of formal methods" and "Seven more myths of formal methods" ([6] [2]).

1.7.4 Any advantage?

be modest

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

- remember the *VIPER* chip

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

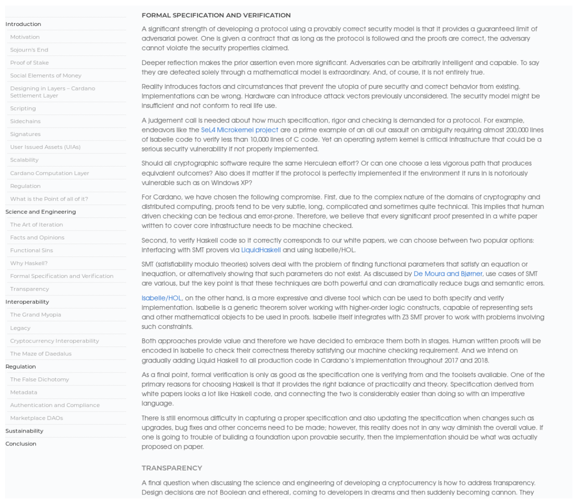
However, the view is also a bit silly. The argument that *no one* uses it is certainly an exaggeration. There are areas where formal methods are at least encouraged by regulatory documents, for instance in the avionics industry. One could make the argument that high-security applications (like avionics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

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

Maybe formal methods also suffer a bit from similar bad reputation as (sometimes) artificial intelligence has (or had). Techniques as investigated by the formal method community are opposed, ridiculed and discounted as impractical until they "disappear" and then become "common practice". So, as long as the standard practicioner does not use something, it's "useless formal methods", once incorporated in daily use it's part of the software process and quality assurance. Artificial intelligence perhaps suffered from a similar phenomenon. At the very beginning of the digital age, when people talked about "electronic brains" (which had, compared to today, ridiculously small speed and capacity), they trumpeted that the electronic brains can "think rationally" etcetc., and the promised that soon they would beat humans in games that require strategic thinking like tic-tac-toe. The computers very soon did just that, with "fancy artifical intelligence" techniques like back-tracking, branch-and-bound or what not (branch-and-bound comes from operations research). Of course the audience then said: Oh, that's not intelligence, that's just brute force and depth-first search, and nowadays, depth-first seach is taught in first semester or even school. And Tic-tac-toe is too simple, anyway, the audience said, but to play chess, you will need "real" intelligence, so if you can come up with a computer that beats chess champions, ok, then we could call it intelligent. So then the AI came up with much more fancy stuff, heuristics, statistics, bigger memory, larger state spaces, used faster computers, but people would still state: well, actually a chess playing computer is not intelligent, it's "just" a complex search. So, the "intelligence" those guys aim at is always the stuff that is not yet solved. Maybe the situation is similar for formal methods.

Perhaps another parallel which has led to negative opinions like the one of Parnas is that the community sometimes is too big-mouthed. Like promising an "intelligent electronic brain" and what comes out is a tic-tac-toe playing back-tracker… For the formal methods, it's perhaps the promise to "guarantee 100% correctness" (done based on "math") or at least perceived as to promise that. For instance, the famous dictum of Disjktra that testing cannot guarantee correctness in all cases is of course in a way a triviality (and should be uncontroversial), but it's perhaps perceived to mean (or used by some to mean) that "unlike testing, the (formal) method can guarantee that". Remember the Viper chip (a "verified" chip used in the military, in the UK). See [11] for a short historical account of the rise and the fall of the Viper chip and the (legal) battles around it. Interestingly an important part of the battle was about the question, discussed in course "what is a mathematical proof". Also the book [7] discusses that.

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



FORMAL SPECIFICATION AND VERIFICATION

A significant strength of developing a protocol using a provably correct security model is that it provides a guaranteed limit of adversarial power. One is given a contract that as long as the protocol is followed and the proofs are correct, the adversary cannot violate the security properties claimed.

Deeper reflection makes the prior assertion even more significant. Adversaries can be arbitrarily intelligent and capable. To say they are defeated solely through a mathematical model is extraordinary. And, of course, it is not entirely true.

Reality introduces factors and circumstances that prevent the utopia of pure security and correct behavior from existing. Implementations can be wrong. Hardware can introduce attack vectors previously unconsidered. The security model might be insufficient and not conform to real life use.

A judgement call is needed about how much specification, rigor and checking is demanded for a protocol. For example, endeavors like the SeL4 Microkernel project are a prime example of an all out assault on ambiguity requiring almost 200,000 lines of Isabelle code to verify less than 10,000 lines of C code. Yet an operating system kernel is critical infrastructure that could be a serious security vulnerability if not properly implemented.

Should all cryptographic software require the same Herculean effort? Or can one choose a less vigorous path that produces equivalent outcomes? Also does it matter if the protocol is perfectly implemented if the environment it runs in is notoriously vulnerable such as on Windows XP?

For Cardano, we have chosen the following compromise. First, due to the complex nature of the domains of cryptography and distributed computing, proofs tend to be very subtle, long, complicated and sometimes quite technical. This implies that human driven checking can be tedious and error-prone. Therefore, we believe that every significant proof presented in a white paper written to cover core infrastructure needs to be machine checked.

Second, to verify Haskell code so it correctly corresponds to our white papers, we can choose between two popular options: interfacing with SMT provers via LiquidHaskell and using Isabelle/HOL.

SMT (satisfiability modulo theories) solvers deal with the problem of finding functional parameters that satisfy an equation or inequation, or alternatively showing that such parameters do not exist. As discussed by De Moura and Bjørner, use cases of SMT are various, but the key point is that these techniques are both powerful and can dramatically reduce bugs and semantic errors.

Isabelle/HOL, on the other hand, is a more expressive and diverse tool which can be used to both specify and verify implementation. Isabelle is a generic theorem solver working with higher-order logic constructs, capable of representing sets and other mathematical objects to be used in proofs. Isabelle itself integrates with Z3 SMT prover to work with problems involving such constraints.

Both approaches provide value and therefore we have decided to embrace them both in stages. Human written proofs will be encoded in Isabelle to check their correctness thereby satisfying our machine checking requirement. And we intend on gradually adding Liquid Haskell to all production code in Cardano's implementation throughout 2017 and 2018.

As a final point, formal verification is only as good as the specification one is verifying from and the toolsets available. One of the primary reasons for choosing Haskell is that it provides the right balance of practicality and theory. Specification derived from white papers looks a lot like Haskell code, and connecting the two is considerably easier than doing so with an imperative language.

There is still enormous difficulty in capturing a proper specification and also updating the specification when changes such as upgrades, bug fixes and other concerns need to be made; however, this reality does not in any way diminish the overall value. If one is going to trouble of building a foundation upon provable security, then the implementation should be what was actually proposed on paper.

TRANSPARENCY

A final question when discussing the science and engineering of developing a cryptocurrency is how to address transparency. Design decisions are not Boolean and ethereal, coming to developers in dreams and then suddenly becoming cannon. They

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

provably correct

1.7.6 Using formal methods

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

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

1.7.7 Formal specification

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

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

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

1.7.8 Proving properties about the specification

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

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

Example

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

- some attempt:

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

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

So perhaps a correct (?) specification might be

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

1.7.9 Formal synthesis

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

Example: program extraction

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

1.7.10 Verifying specifications w.r.t. implementations

Mainly two approaches:

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

1.7.11 A few success stories

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

. . .

1.7.12 Classification of systems

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

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

The classification here follows Klaus Schneider's book "Verification of reactive systems" [12]. Obviously, one can classify "systems" in many other ways, as well.

1.7.13 Classification of systems: architecture

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

1.7.14 Classification of systems: type of interaction

- **Transformational** systems: Read inputs and produce outputs – These systems should always terminate
- **Interactive** systems: Idem previous, but they are not assumed to terminate (unless explicitly required) – Environment has to wait till the system is ready

- **Reactive** systems: Non-terminating systems. The environment decides when to interact with the system – These systems must be fast enough to react to an environment action (real-time systems)

1.7.15 Taxonomy of properties

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

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

Temporal behavior The answer arrives in less than 40 seconds

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

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

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

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

1.7.16 When and which formal method to use?

Examples:

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

Open distributed, concurrent systems ⇒ *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.

1.8 Formalisms for specification and verification

1.8.1 Some formalisms for specification

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

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

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

1.8.2 Some techniques and methodologies for verification

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

1.9 Summary

1.9.1 Summary

- **Formal methods** are useful and needed
- which FM to use depends on the problem, the underlying system and the property we want to prove
- for real complex systems, only part of the system may be formally proved and no single FM can do the task
- our course will concentrate on
  - temporal logics as a specification formalism
  - safety, liveness and (maybe) fairness properties
  - Spin (LTL model checking)
  - few other techniques from student presentation (e.g., abstract interpretation, CTL model checking, timed automata)

1.9.2 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

1.9.3 Further reading

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

- Klaus Schneider: Verification of reactive systems, 2003. Springer. Chap. 1 [12]
- 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. [8]

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] Dijkstra, E. W. (1969). Notes on structured programming. Technical Report EWD-249, Technological University, Eindhoven.

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

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

[7] MacKenzie, D. A. (2001). *Mechnanizing Proof.* MIT Press.

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

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

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

[11] Pygott, C. (2019). The VIPER microprocessor. In *2019 6th IEEE History of Electrotechnology Conference (HISTELCON)*, pages 14–19.

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

Index