



Chapter 1

Formal methods

Course “Model checking”

Martin Steffen

Autumn 2021



Chapter 1

Learning Targets of Chapter “Formal methods”.

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



Chapter 1

Outline of Chapter “Formal methods”.

Motivating example

How to guarantee correctness?

Software bugs

On formal methods

Formalisms for specification and verification

Summary



Section

Motivating example

Chapter 1 “Formal methods”

Course “Model checking”

Martin Steffen

Autumn 2021

A simple computational problem



IN5110 –
Verification and
specification of
parallel systems

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

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

A straightforward implementation



IN5110 –
Verification and
specification of
parallel systems

```
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: A straightforward implementation

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

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
```



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

Should we trust software?

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



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

Should we trust software?



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

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

Where

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

We get then

$$a_{20} \approx 6 \quad (2)$$



Section

How to guarantee correctness?

Chapter 1 “Formal methods”

Course “Model checking”

Martin Steffen

Autumn 2021

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. $\lim_{n \rightarrow \infty} a_n \approx 6$)

- **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



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

How to guarantee correctness?

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

Dijkstra's dictum

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

A lesser known dictum from Dijkstra (1965)

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



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

Validation & verification



IN5110 –
Verification and
specification of
parallel systems

- 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

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

Approaches for validation



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

- 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”



Section

Software bugs

Chapter 1 “Formal methods”

Course “Model checking”

Martin Steffen

Autumn 2021

Sources of errors

- 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/compiler
- wrong use of tools/programs
- ...



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

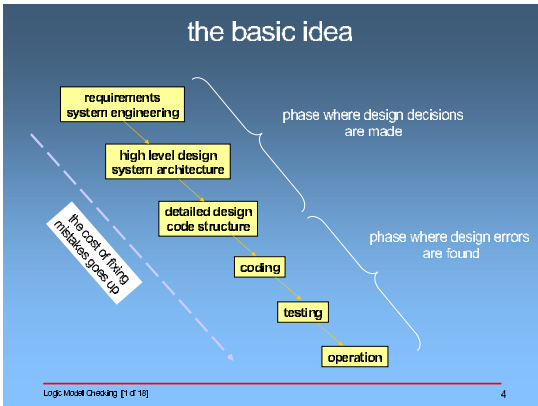
Formalisms for
specification and
verification

Summary

Errors in the SE process



IN5110 –
Verification and
specification of
parallel systems



Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

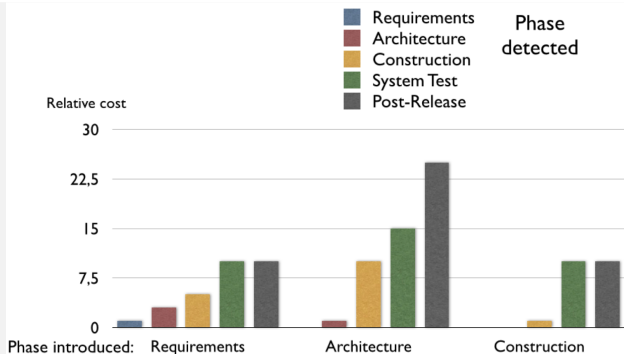
Formalisms for
specification and
verification

Summary

Costs of fixing defects



IN5110 –
Verification and
specification of
parallel systems



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

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

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



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

**Motivating
example**

**How to guarantee
correctness?**

Software bugs

**On formal
methods**

**Formalisms for
specification and
verification**

Summary



Section

On formal methods

Chapter 1 “Formal methods”

Course “Model checking”

Martin Steffen

Autumn 2021

What are formal methods?



IN5110 –
Verification and
specification of
parallel systems

FM

“Formal methods are a collection of notations and techniques for describing and analyzing systems” [2]

- **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)

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

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



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

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*)



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

Any advantage?



IN5110 –
Verification and
specification of
parallel systems

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

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

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



IN5110 – Verification and specification of parallel systems

Targets & Outline

Motivating example

How to guarantee correctness?

Software bugs

On formal methods

Formalisms for specification and verification

Summary

Introduction

Motivation
Sjogren's End

Proof of Stake

Social Elements of Money

Designing in Layers – Cardano

Settlement Layer

Scripting

Sidechains

Signatures

User Issued Assets (UFA)

Scalability

Cardano Computation Layer

Regulation

What is the Point of all of it?

Science and Engineering

The Art of Iteration

Facts and Opinions

Functional Sins

Why Haskell?

Formal Specification and Verification

Transparency

Interoperability

The Grand Myopia

Legacy

Cryptocurrency Interoperability

The Maze of Daedalus

Regulation

The False Dichotomy

Metadata

Authentication and Compliance

Marketplace DACs

Sustainability

Conclusion

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 abnormally 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 [SetL4 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 SMF provers via [LiquidHaskell](#) and using Isabelle/HOL.

SMF (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 Bjerner](#), use cases of SMF 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 SMF 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 common. They

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



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

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



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

Proving properties about the specification



IN5110 –
Verification and
specification of
parallel systems

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:

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

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

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)



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

Verifying specifications w.r.t. implementations



IN5110 –
Verification and
specification of
parallel systems

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}

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

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

...



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

Classification of systems



IN5110 –
Verification and
specification of
parallel 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*
 - *type of interaction*

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

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



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

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)



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

Taxonomy of properties

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



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

When and which formal method to use?



IN5110 –
Verification and
specification of
parallel systems

Examples:

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

Open distributed, concurrent systems \Rightarrow **Very difficult!**
Need the combination of different techniques

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary



Section

Formalisms for specification and verification

Chapter 1 “Formal methods”

Course “Model checking”

Martin Steffen

Autumn 2021

Some formalisms for specification



IN5110 –
Verification and
specification of
parallel systems

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

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

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



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary



Section

Summary

Chapter 1 “Formal methods”

Course “Model checking”

Martin Steffen

Autumn 2021

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)



IN5110 –
Verification and
specification of
parallel systems

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

Ten Commandments of formal methods



IN5110 –
Verification and
specification of
parallel systems

From “Ten commandments revisited” [1]

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

Targets & Outline

Motivating
example

How to guarantee
correctness?

Software bugs

On formal
methods

Formalisms for
specification and
verification

Summary

References I



IN5110 –
Verification and
specification of
parallel systems

Bibliography

- [1] 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.
- [2] Peled, D. (2001). *Software Reliability Methods*. Springer Verlag.

Targets & Outline

**Motivating
example**

**How to guarantee
correctness?**

Software bugs

**On formal
methods**

**Formalisms for
specification and
verification**

Summary