INF5140 – Specification and Verification of Parallel Systems

Spring 2017

Institutt for informatikk, Universitetet i Oslo

January 20, 2017

Formal Methods

Outline

The problem

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

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

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

A Java Implementation

```
1  public class Mya {
2
3      static double a(int n) {
4        if (n==0)
5           return 11/2.0;
6        if (n==1)
7           return 61/11.0;
8        return 111 - (1130 - 3000/a(n-2))/a(n-1);
9      }
10
11     public static void main(String[] argv) {
12       for (int i=0;i<=20;i++)
13         System.out.println("a("+i+") = "+a(i));
14     }
15  }
```

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) = 99.9999627595651 1
```

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

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

Where

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

We get then

$$a_{20} \approx 6$$

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

Examples:

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

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

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

Dijkstra (1972) on testing

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

Dijkstra (1965) on proving programs correct

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

[a] One may debate that.

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

[3] Undecidability of the halting problem by Turing

What is validation?

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

The following may clarify the difference between these terms:

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

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

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

Usual approaches for validation

The following techniques are used in industry for validation:

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

[5] From Peled's book [7]

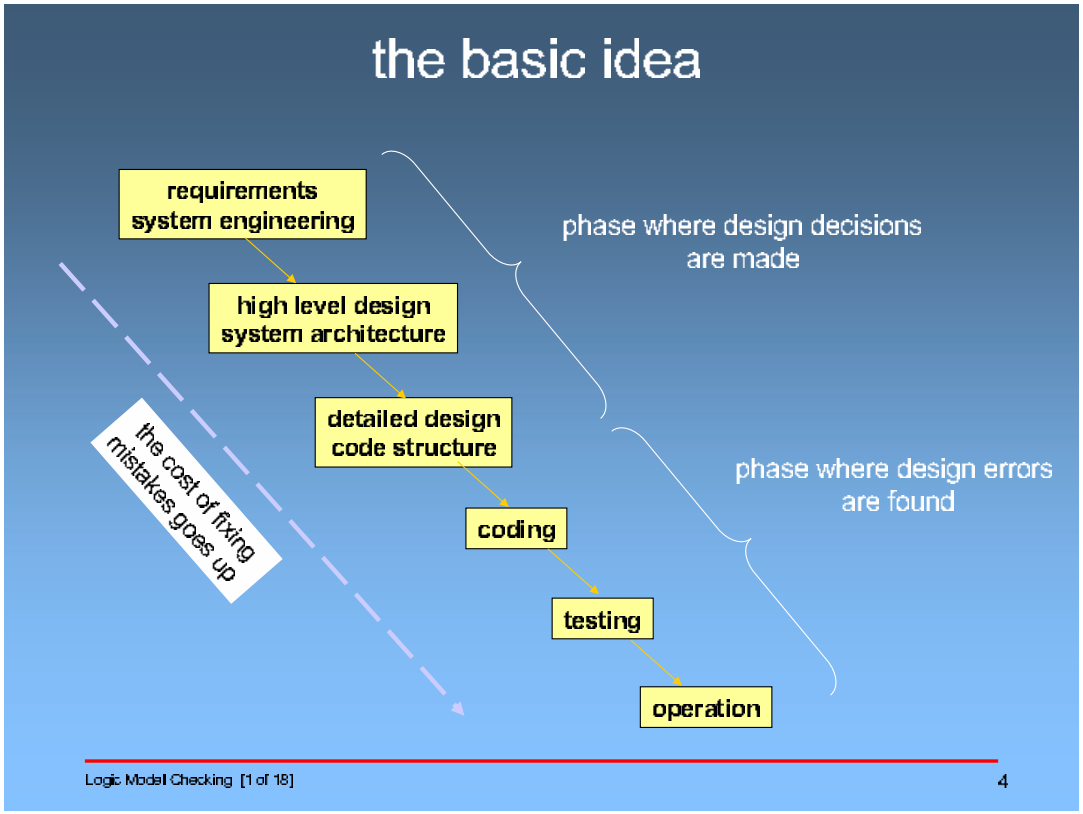
Sources of errors

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

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

Source of errors

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



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

Some (in-)famous software bugs [4][a]

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

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

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

Some (in-)famous software bugs[a]

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

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

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

Some (in-)famous software bugs[a]

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

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

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

FM [7]

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

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

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

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

We will use the following definition (reminder):

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

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

Limitations

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

Any advantage?

Of course

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

Using formal methods

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

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

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

Formal specification

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

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

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

Proving properties about the specification

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

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

Example

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

- - First attempt:

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

INCORRECT! – The error may be found when trying to prove some properties

- "Correct" (?) specification:
  $a(0) \wedge a(1) \wedge \forall t \geq 0 \cdot a(t+2) = \neg a(t+1)$

Formal synthesis

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

Example: program extraction

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

Verifying specifications w.r.t. implementations

Mainly two approaches:

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

A few success stories

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

Classification of systems

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

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

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

Classification of systems

According to the system 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

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

Taxonomy of properties

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

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

  Temporal behavior: The answer arrives in less than 40 seconds

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

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

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

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

When and which formal method to use?

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

Some formalisms for specification

An incomplete list of formalisms for specifying systems:

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

Some techniques and methodologies for verification

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

Summary

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

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

Ten Commandments of formal methods

From "Ten commandments revisited" [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

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

- Klaus Schneider: Verification of reactive systems, 2003. Springer. Chap. 1 [8]
- 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[9] [6]

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

References I

[1] Gregory R. Andrews.
Foundations of Multithreaded, Parallel, and Distributed Programming.
Addison-Wesley, 2000.

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

[3] Jonathan P. Bowen and Michael G. Hinchey.
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, 2005. ACM Press.

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

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

[6] Zohar Manna and Amir Pnueli.
The temporal logic of reactive and concurrent systems—Specification.
Springer Verlag, New York, 1992.

[7] Doron Peled.
Software Reliability Methods.
Springer Verlag, 2001.

References II

[8] Klaus Schneider.
Verification of Reactive Systems.
Springer Verlag, 2004.