Formal Analysis and Verification of Software

Ramon Béjar Torres (UdL)

DIEI
Universitat de Lleida

Course 2022-2023

Why Formal Verification of Programs ?

How can we be sure that a program will work as expected ?

Can we verify in some way that a program will work right for any inputs ?

What kind of analysis should we perform on a program to verify its correctness ?

Why Formal Verification of Programs ?

The most naive way of verifying programs is not really verifying them, but testing them on some particular cases

For most of the programs, a limited set of hand designed tests will not be enough to ensure that the program will work right in all the situations !

We need to use methods based on formal verification, that give guarantees about the corretness of the program

Why Formal Verification of Programs ?

Do not you think software bugs cause real troubles ?

What software bugs cause

http://en.wikipedia.org/wiki/List_of_software_bugs

Famous Last Bugs (Before Death Came) - Mariner 1 Probe

Destroyed 293 seconds after launch.
Implementation error due to erroneous transcription of the original specification of a function

From $\overline{\dot{R}_n}$ to $\dot{R}_n$

$\overline{\dot{R}_n}$ nth smoothed value of derivative of function R. Without the over bar, the implemented function returned higher values than expected

Shows the importance of carefully checking that implementation satisfies the specification !

Famous Last Bugs (Before Death Came) - Ariane 5

Ariane 5 Flight 501. Destroyed 37 seconds after launch.
Software bug in its inertial navigation system.

Arithmetic Overflows

Implicit conversion from 64-bit floating point to 16-bit signed integer caused **arithmetic overflow**

Efficiency considerations omitted range checks for a particular variable

Shows the importance of carefully checking that the details of data type conversions do not generate unexpected behaviours !

More info: Read this http://en.wikipedia.org/wiki/Ariane_5_Flight_501

Disastrous Bugs - Spirit Rover

Landed on January 4, 2004. Ceased communicating on January 21

Checking Critical Situations

The system reached an exceptional case: file system almost full

It was expected that this situation would never arise (because of *manual* checking of the file system by mission technicians)

Too many files in the file system ! This time, the problem was that the situation was not even specified as something the system should check automatically

More info: http://www.computerworld.com/s/article/89829/Out_of_memory_problem_caused_Mars_rover_s_glitch

Disastrous Bugs - Northeast Blackout

A race condition software bug stalled the control room alarm system of an electric company (FirstEnergy).
The lack of alarm left operators unaware of a critical situation in the power-grid, causing a delayed response of the company in front of different overload situations, that together with other events caused major power outages in different USA and Canada states.

Race Condition

A race condition indicates a situation in which the result of certain computation can depend on the particular ordering of certain events that cannot be predicted/controlled by the system
This tipically happens in concurrent programs where the possible concurrent access to some shared variables can produce a different (an erronous) result if a particular ordering of events takes place

What can we (software engineers) do ?

How to avoid software bugs ?

- Specification. Specify clearly, **no ambiguity**, the expected behaviour of the program $\Rightarrow$ Use formal languages with clear semantics !
- Implementation. Two options:
  - Use tools for semi-automatic generation of code such that it will satisfy the specification. For example, the B-method: http://en.wikipedia.org/wiki/B-Method
  - Implement the program by hand, with the help of any desired methodology and tools. Verify it later !
- Verification. If code implemented *by hand*, need to verify that it satisfies its specification

We will introduce **the basics** behind the current methods and tools that try to solve these problems, but based on the three steps approach: specify, design-implement, verify.

What Amazon (software engineers) Are Starting to Do

Software engineers at Amazon have started to use formal methods for specifying and verifying the correctness of some critical distributed algorithms. Read the paper here !

They use a set of tools based on the formal specification language TLA+ (Temporal Logic of Action) and the pseudo-code language PlusCal

By 2014, they have used the TLA+ tools in 10 complex systems in Amazon, allowing them to find subtle bugs on the design of the systems

A TLA+ Specification for a Simple Clock

MODULE *HourClock*

EXTENDS *Naturals*

VARIABLE *hr*

$$HCini \triangleq hr \in (1 \,..\, 12)$$

$$HCnxt \triangleq hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$$

$$HC \triangleq HCini \wedge \Box[HCnxt]_{hr}$$

We specify that our clock system always satisfies the *HCini* predicate

THEOREM $HC \Rightarrow \Box HCini$

- hr' means the value of hr in the next state of the system
- $\Box[HCnxt]_{hr}$ means that in any state, either HCnxt is true or the variable *hr* does not change its value

Static Verification of Code in Amazon

Amazon is also developing tools, based on some formal verification methods, to find errors in existing code developed in C/C++/Java:

- Generic errors, like memory leaks
- Specific contracts (conditions we need to ensure they are satisfied in certain parts of a program)

You can find more information is this blog:
https://www.amazon.science/blog/how-automated-reasoning-improves-the-prime-video-experience

Course Outline

1. Complete Verification. Verify that software completely satisfies its specification. That is, for **any given input situation** the software will respond **as expected**.
   We will present the basics behind tools based on First-Order Logic (FOL) specification and (semi-)automatic proof generation. Concrete tool presented and used: `KeY-Hoare` *(Formal Verification of Imperative Programs with Hoare-style Logical Calculus)*

2. Partial Verification. Verify that software satisfies some **critical safety properties**. So, no intention to check complete specifications.
   We will present the basics behind tools based on Bounded Model Checking (BMC). Concrete tool presented and used: `CBMC` *(Bounded Model Checking for ANSI-C programs)*

   We will also give a glimpse about te use of a BMC tool for the verification of hardware designs