Semantics and Applications to Verification

Jérôme Feret and Xavier Rival
Notes by Antoine Groudiev

9th February 2024

Contents

Introduction

This document is Antoine Groudiev's class notes while following the class *Sémantique et applications à la vérification de programmes* (Semantics and Applications to Verification) at the Computer Science Department of ENS Ulm. It is freely inspired by the class notes of Xavier Rival.

1 Introduction to Semantics

1.1 Case studies

We will study some examples of software errors: what are the causes of these, what kind of properties do we want to verify in order to prevent such failures?

1.1.1 Ariane 5 – Flight 501

Ariane 5 was a satellite launcher, aimed at replacing Ariane 4. Its first flight, June, 4th, 1996, was a failure, with more than $370 000 000 of damages. 37 seconds after the launch, the rocket exploded.

The system contained sensors, two calculators (SRI, OBC), actuators, and redundant systems (failure tolerant system). The failure was due to an unhandled arithmetic error. Each register of the SRI has a size of 16, 32, or 64 bits. The error was due to a conversion of a 64-bit float to a 16-bit integer. The value was too large to be represented in 16 bits, and the conversion failed. The software was not able to handle this error, and the system crashed.

Several solutions would have prevented this mishappening:

- Desactivate interruptions on overflows
- Fix the SRI code, so that no overflow can happen. All conversions must be *guarded against overflows*:

```
double x = /* ... */ ;
short i = /* ... */ ;
if ( -32768. <= x && x <= 32767. )
    i = ( short ) x ;
else
    i = /* default value */ ;
```

  This may be costly, but redundant tests can be removed.
- Handle converson errors (not trivial): identify the problem and fix it at run time.

The piece of code that generated the error was used to do a useless task, the re-calibration process is not usefull after take-off. Furthermore, the code was already used in Ariane 4; initially protected by a safety guard, many conversions and tests were removed for the sake of performance after being tested on Ariane 4.

The crash was not prevented by redundant systems: the two calculators were running the same code, and the same error was made on both. Redundancy can prevent hardware errors, but is not enough to prevent software errors.

1.1.2 Lufthansa Flight 2904

On November 22, 2003, a Lufthansa Airbus A320-200 crashed at the airport of Warsaw, Poland. The plane was landing, and the weather was bad. The plane was not able to stop before the end of the runway, and crashed into a building. The cause of the crash was a software error in the plane's computer. The plane was not able to compute the correct deceleration, and the pilot was not able to stop the plane in time.

1.1.3 Patriot missile (anti-missile system), Dahran

The purporse of the Patriot system was to destroy foe missiles before they reach their target, and was used in the first Gulf War with a success rate around 50%. The system was used to destroy Scud missiles, and the system was not able to destroy one of them, which hit a barrack, killing 28 people. The cause of the failure was a software error in the system's clock. The system was not able to compute the time correctly due to fixed precision arithmetic error, and the missile was not destroyed.

1.1.4 General remarks

The examples given so far are not isolated cases. The typical causes of software errors are:

- Improper specification
- Incorrect implementation of a specification (the code should be free of runtime errors, and should produce a result that meets some property)
- Incorrect understanding of the execution model (generation of too imprecise results)

This creates new challenges to ensure embedded systems do not fail. The main techniques to ensure software safety are software development techniques:

- software engineering
- programming rules
- make software cleaner

In this class, we will instead dive into formal methods:

- should have sounds mathematical foundations
- should allow guaranteeing software meet some complex properties
- should be trustable
- increasingly used in real life applications

This course will contain two main parts. The first part will be about Semantics, which allow describing precisely the behavior of programs, express the properties to verify, and to transform and compile programs. The second part, Verification, aims at proving semantic properties of programs. A very strong limitation of verification is indecidability; several approaches make various compromises around indecidability.

1.2 Approaches to verification

1.2.1 The termination problem

Definition (Termination)**.** A program P terminates on input X if and only if any execution of P with input X eventually reaches a final state. A final state is a final point in the program (i.e., not an error).

Definition (Termination problem)**.** Can we find a program Pt that takes as argument a program P and data X and that returns `True` if P terminates on X, and `False` otherwise?

Property. *The termination problem is not computable.*

Proof. We assum there exists a program Pa such that Pa always terminates, and returns 1 if and only if P terminates on input X. We consider the following program:

```
void P0 ( P ) {
    if ( Pa ( P , P ) == 1 ) {
        while ( 1 ) {
            // loop forever
        }
    } else {
        return ; // do nothing
    }
}
```

and we consider the return value of $Pa(P0, P0)$. If $Pa(P0, P0) == 1$, then $P0$ loops forever, and if $Pa(P0, P0) == 0$, then $P0$ terminates. This is a contradiction, and the termination problem is not computable. □

Property. *The absence of runtime errors is not computable. We cannot find a program Pc that takes a program P and input X as arguments, always terminates, and returns 1 if and only if P runs on input X without a runtime error.*

Theorem (Rice theorem)**.** *Considering a Turing complete language, any non-trivial semantic specification is not computable. Therefore, all interesting properties are not computable (termination, absence of runtime/arithmetic errors, etc.).*

The initial verification problem is not computable. Several compromises can be made: simulation, testing, assisted theorem proving, model checking, bug-finding, static analysis with abstraction.

Definition (Safety verification problem)**.** The Semantics $[\![P]\!]$ of a program P is the set of behaviors of P (e.g. states). A property to verify $\mathcal{S}$ is the set of admissible behaviors (e.g. safe states). Our goal is to establish $[\![P]\!] \subseteq \mathcal{S}$

$[\![P]\!]$ can be sound (identify any wrong program), complete (accept all correct programs), and automated, but not all three at the same time.

Testing by simulation The principle of testing by simulation is to run the program on finitely many finite inputs, to maximize coverage and inspect erroneous traces to fix bugs. It is very widely used, through unit testing, integration testing, etc. It is both automated and complete, but is unsound and costly.

Machine assisted proof The principle of machine assisted proof is to have a machine check proof that is partly human written: tactics or solvers may help in the inference, and the hardest invariants have to be user-supplied. It is sound and quasi-complete, but not fully automated and costly.

Model checking We consider finite systems only, using algorithms for exhaustive exploration, symmetry reduction, ... It is automated, sound, and complete *with respect to the model*.

Bug finding The principle of bug finding is to identify "likely" issues, patterns known to often indicate an error: it uses bounded symbolic execution, model exploration, and rank "defect" reports using heuristics. It is neither sound nor complete, but is fully automated.

Static analysis with abstraction The principle of static analysis with abstraction is to use some approximation, but always in a conservative manner. We can use under-approximation of the property to verify:

$$\mathcal{S}_{\text{under}} \subseteq \mathcal{S}$$

and over-approximation of the semantics:

$$[\![P]\!] \subseteq [\![P]\!]_{\text{upper}}$$

We let an automatic static analyser attempt to prove that:

$$[\![P]\!]_{\text{upper}} \subseteq \mathcal{S}_{\text{under}}$$

If it succedds, then we have proven that $[\![P]\!] \subseteq \mathcal{S}$. It is automated, sound, but incomplete.

1.2.2 A summary of common verification techniques

1.3 Orderings, lattices and fixpoints

	Automatic	Sound	Complete	Source level
Simulation	Yes	No	Yes	Yes
Assisted Proving	No	Yes	Almost	Partially
Model-checking	Yes	Yes	Partially	No
Bug-finding	Yes	No	No	Yes
Static analysis	Yes	Yes	No	Yes