Apprendre la vérification avec Cameleer

Structures de données et algorithmes vérifiés

© Pedro Gasparinho[†]
Mário Pereira[†]

[†] Nova School of Science and Technology, Portugal

To my someone and someone

Pedro Gasparinho

To my someone and someone

Mário Pereira

If you make a mistake and do not correct it, this is a mistake.

Confucius

Acknowledgements

Foreword

Preface

There is a Latin saying that states "Errare humanum est", which can be translated into English as "To err is human" or into Portuguese as "Errar é humano", and programmers are no exception. While making errors is human nature, critical software must not fail, as it can endanger people and animals lives, lead to the theft of confidential information, cause environmental damage, or result in countless other catastrophic consequences. Even if incidents can still occur due to other sources, such as human negligence or mechanical failure, it is imperative to guarantee the correctness of such programs, as it is a great leap forward into minimizing the probability of catastrophes occurring.

Unlike other engineering fields, in Computer Science, testing before release is highly accessible due to the virtual nature of software. While this should lead to more reliable products, that is not necessarily the case, in fact, there is no shortage of program failure incidents. There are numerous reasons for this, ranging from human tiredness to incoordination between developers, but most important is that providing exhaustive test coverage is unfeasible for any industrial-grade software, due to the sheer number of possible scenarios. In fact, this was noted by Dijkstra, as early as 1969, with his famous quote *"Program testing can be used to show the presence of bugs, but never to show their absence!"*.

If testing is not enough to guarantee correctness, then what should we do? The answer lies in using formal methods, which consist in applying mathematical techniques during the process of developing software. We are particularly interested in Formal Verification, which allows proving if a program complies To a given specification or not. Even though the specification itself may still be incomplete or even incorrect, Formal Verification offers much stronger promises regarding software correctness, when compared to testing.

The history of Formal Verification is long and deeply rooted in the academia, despite this, there have been numerous success stories in the industry and hopefully many more to come. Moreover, with the evolution of computer power and investment in this research area, automated verification tools may reach a maturity level necessary to become widespread in the critical software industry, due to their ease of use, when compared to their predecessors, the proof assistants.

This book is meant to be an introductory learning tool for the new generation of verification engineers or anyone with interest in the area. We will cover proofs on well-known problems, algorithms and data structures, since most readers at this level have likely studied them beforehand, or at the very least may quickly find more information on these topics in other easily accessible sources.

Contents

Chapter 1

Installation Procedures

We recommend to our readers the use of a Unix operating system, since the tools used in this book have better support in these types of systems.

OCaml & *Opam*

The first step in this process is to install the *OCaml* programming language, or rather, it's package manager, *opam*, which comes with *OCaml's* compiler, the basic packages and auxiliary tools.

This can be done either through the operating system's package manager (not to be confused with a programming language package manager, such as *opam*), or using the provided script.

For a simple guide regarding *opam*'s installation process visit:

https://ocaml.org/install

For a more complete guide visit:

https://opam.ocaml.org/doc/Install.html

Via Script

Open the terminal an execute this command:

```
bash -c "sh <(curl -fsSL https://opam.ocaml.org/install.sh)"
```

Via OS's Package Manager

For debian, ubuntu, mint or similar users, one may use execute the following command in the terminal:

```
apt install opam
```

Opam is available in Homebrew and MacPorts for macOS users:

```
brew install opam # Homebrew
port install opam # MacPorts
```

For other operating systems check the complete guide mentioned above.

Setting up *opam*

After installing *opam*, one needs to initialize it using:

```
opam init
```

At the end of the last operation, a prompt appears to set the environment variables, it is recommended to answer **yes**. Otherwise, one must run the following command when accessing *opam*'s installation:

```
eval ‘opam config env‘
```

Installing *Why3* and provers

Why3 is a platform for deductive program verification with use cases in both academia and industry. If features a rich specification library, along many other important features, however its main drawback, from an industrial standpoint, is that it only accepts directly programs written in its own programming and specification language, *WhyMl*. This language is not widely used as general-purpose programming language in the industry, when compared to *OCaml*. Despite this, *WhyMl* has been used as an intermediary language in various projects to prove more mainstream languages, such as *C*, *Java*, *Ada* or *OCaml*.

More information about installing *Why3* and the automated provers can be found at:

https://marche.gitlabpages.inria.fr/lecture-deductive-verif/install.html

Installing *Why3*

Why3 and its *IDE* can be installed through *opam*:

```
opam install why3 why3-ide
```

Why3 relies on third-party provers, which allows combining multiple provers in the same proof, and each may have its own strengths and limitations. To start with we recommend installing *Alt-Ergo*, *cvc5* and *Z3*.

Installing *Alt-Ergo*

Similarly, it can be installed through *opam*:

```
opam install alt-ergo
```

Installing *cvc5*

Unlike *Alt-Ergo*, *cvc5* and *Z3* are not available in *opam*, as of the release date of this version of the book. Instead, one must download the provers from the corresponding *github* pages.

The latest versions of *cvc5* include numerous options for each OS and architecture, which may be hard to distinguish which is the best suited for the user, one may experiment or go back to version 1.1.0, which features simpler variations.

To install version 1.1.0 for Linux, via terminal, one can use the following sequence of commands:

```
wget https://github.com/cvc5/cvc5/releases/download/cvc5-1.1.0/cvc5-Linux
sudo cp cvc5-Linux /usr/local/bin/cvc5
sudo chmod +x /usr/local/bin/cvc5
cvc5 --version
```

This sequence, by step, is equivalent to:

1. Downloading the desired file from *github*
2. Moving it from the working directory to */usr/local/bin*
3. Granting the file permission to execute
4. Checking the version (and if it is actually installed)

Installing *Z3*

Similarly to *cvc5*, to install the latest version (4.15.0) of *Z3*, via terminal, one may use the following sequence of commands:

```
wget https://github.com/Z3Prover/z3/releases/download/z3-4.12.2/z3-4.15.0-x64-glibc-2.39.zip
unzip z3-4.15.0-x64-glibc-2.39.zip
sudo cp z3-4.15.0-x64-glibc-2.39/bin/z3 /usr/local/bin
sudo chmod +x /usr/local/bin/z3
z3 --version
```

Installing Cameleer

For the reasons mentioned above, *Cameleer* emerged. This piece of software is used to verify OCaml programs with GOSPEL (Generic OCaml SPEcification Language) annotations. *Cameleer*, internally, translates these programs into *WhyMl*, which can then be verified in the *Why3* platform. For more information check:

https://github.com/ocaml-gospel/cameleer

Cameleer can be installed via terminal with the following sequence of commands:

```
git clone https://github.com/ocaml-gospel/cameleer
opam pin add path/to/cameleer
cameleer --version
```

Chapter 2

Programming with OCaml

Chapter 3

Introduction to Cameleer

Chapter 4

Mathematical Problems

Chapter 5

Sorting Algorithms

Chapter 6

Searching Algorithms

Chapter 7

Data Structures

Chapter 8

Graph Algorithms

Chapter 9

Selected Topics