Chapter X

Certifying Properties of Programs Using Theorem Provers

J. Santiago Jorge, University of A Coruña, Spain
Víctor M. Gulías, University of A Coruña, Spain
David Cabrero, University of A Coruña, Spain

Abstract

Proving the correctness of a program, even the simplest one, is a complex and expensive task; but, at the same time, it is one of the most important activities for a software engineer. In this chapter, we explore the use of theorem provers to certify properties of software; in particular, two different proof-assistants are used to illustrate the method: Coq and PVS. Starting with a simple pedagogic example, a sort algorithm, we finally reproduce the same approach in a more realistic scenario, a model of a block-allocation algorithm for a video-on-demand server.

Introduction

The difficulty of certifying the correctness of a programmakes the process very expensive in the cycle of software development. The most frequent validation method consists of running the target program on a set of selected inputs. Unfortunately, although testing can detect some bugs, it does not guarantee the correctness of our software because the...
input set is generally incomplete (Ghezzi, Jazayeri, & Mandrioli, 1991). Formal methods, on the other hand, are proposed to complement traditional debugging techniques, assuring that some relevant property holds in any execution of the program (Clarke et al., 1996; Gunter & Mitchell, 1996). Through formal methods we understand tools, techniques, or languages that help us to develop certified software. Those methods state that a program is correct with respect to a particular specification. Two main techniques are taken into account:

- **Model checking**: Proves that every possible state of the program satisfies a specification
- **Theorem proving**: Derives a proof establishing that the program satisfies a specification

It must be mentioned that formal methods are not intended to provide absolute reliability. The strength of formal methods resides in increasing the reliability (Clarke et al., 1996; Peled, 2001). They assist us in producing quality software by improving our confidence in programs. However, they are not the panacea for detecting software errors. Software verification methods do not guarantee the correctness of actual code, but rather allow the verification of some abstract model of it. In addition, this verification is done with respect to a given specification, which is written manually, and it is sometimes incomplete or may not express accurately the expected program behaviour. Besides, large systems are difficult to study as a whole and therefore, they must be separated into small pieces in order to study each one.

The literature about model checking (Bérard et al., 2001; Clarke, Grumberg, & Peled, 1999; Nielson, Nielson, & Hankin, 1999; Roscoe, 1994) is more prolific than that of theorem proving; however, in the present chapter we explore the use of theorem provers to check particular properties of software. These formal proofs should help us understand programs better. Two approaches are considered:

- **Manual proofs**: We reason directly over the source code, applying techniques like equational reasoning and structural induction. The proof methods are rigorous, though error-prone.
- **Theorem provers**: Besides assisting us in the development of the proofs, they guarantee their correctness, preventing bugs that may be introduced in a hand-made proof.

Theorem provers are usually based on a logic, like first-order logic, higher-order logic, or a version of set theory which provides a framework for the formalization of mathematics. Interactive proof assistants require a human user to give hints to the system. Depending on the degree of automation, significant proof tasks can be performed automatically. Available implementations of generic theorem provers are ACL2 (Kaufmann, Mannolios, & Moore, 2000a, 2000b), Coq (Bertot & Casteran, 2004; The Coq Development Team, 2004), HOL (Gordon & Melham, 1993), Isabelle (Paulson,
Related Content

Project Contexts and the Possibilities for Mixing Software Development and Systems Approaches
[www.igi-global.com/chapter/project-contexts-possibilities-mixing-software/77713?camid=4v1a](www.igi-global.com/chapter/project-contexts-possibilities-mixing-software/77713?camid=4v1a)

A Novel Approach for Ontology-Based Feature Vector Generation for Web Text Document Classification
[www.igi-global.com/article/a-novel-approach-for-ontology-based-feature-vector-generation-for-web-text-document-classification/191205?camid=4v1a](www.igi-global.com/article/a-novel-approach-for-ontology-based-feature-vector-generation-for-web-text-document-classification/191205?camid=4v1a)

PRISM: Visualizing Personalized Real-Time Incident on Security Map
[www.igi-global.com/article/prism/210454?camid=4v1a](www.igi-global.com/article/prism/210454?camid=4v1a)

From Frequent Features to Frequent Social Links
[www.igi-global.com/article/from-frequent-features-to-frequent-social-links/80197?camid=4v1a](www.igi-global.com/article/from-frequent-features-to-frequent-social-links/80197?camid=4v1a)