Imperative Program Verification with *Theorema*

EXTENDED ABSTRACT

Tudor Jebelean*
RISC-Linz
jebelean@risc.uni-linz.ac.at

Abstract

Approaching the problem of imperative program verification from a practical point of view has certain implications concerning: the style of *specifications*, the *programming language* which is used, the help provided to the user for finding appropriate *loop invariants*, the theoretical frame used for *formal verification*, the language used for expressing generated verification *theorems* as well as the database of necessary *mathematical knowledge*, and finally the *proving power, style and language*.

The *Theorema* system has certain capabilities which make it appropriate for such a practical approach: the logic language of the system is *higher-order predicate logic* expressed in natural style (including two-dimensional formulae); the *procedural language* is simple and intuitive, yet sufficiently expressive and fully integrated in the logical frame of the system; the *language and the style of the proofs are natural*, similar to those used by humans; and finally the proving power of *Theorema* is enhanced by using *specific provers for special domains*, which are integrated with *sophisticated mathematical algorithms*.

Combined with a practically oriented version of the theoretical frame of Hoare-Logic, *Theorema* provides readable arguments for the correctness of programs, as well as useful hints for debugging.

Introduction

Program verification (and verification of informational systems in general) will probably play an important role in the development of information technologies (IT), because, as IT products become more sophisticated, their reliability is more difficult to insure by artizanal means, and as IT products become more present in all aspects of human activity, their un-reliability has more dramatic negative effects.

While several powerful formalisms for program verification have been developed in the last decades, we still witness a wide the gap between theory and practice.

We discuss here about using *Theorema* for imperative program verification using Hoare triples, and we also illustrate the difference between the theoretical and practical approach in Hoare-Logic [4].

The *Theorema* system [2] is an automated assistant for mathematicians and engineers, aimed at proving, computing, and solving in natural style. The system is developed at RISC-Linz under the leadership of Bruno Buchberger, who initiated the main ideas of the project. During the years, several researchers contributed to the design and implementation (see `www.theorema.org`).

*The program verification project is supported by BMBWK (Austrian Ministry of Education, Science, and Culture) in the frame of the e-Austria Timisoara project, which is supported additionally by BMWA (Austrian Ministry of Economy and Work) and by MEC (Romanian Ministry of Education and Research). The Theorema system is supported by FWF (Austrian National Science Foundation) – SFB project P1302.

Main Characteristics of Program Verification

The following elements characterize the process of verification using Hoare–Logic (for a tutorial introduction see also [1, 6]):

- an imperative *program*
- and a logical *specification*
- and (usually) some logical *invariants*
- are fed into a **verification system**,
- which generates a *conjecture*,
- which is fed into a **prover**,
- and one obtains the *answer* whether the program fulfills the specification.

The program is expressed in a `programming language`, the specification, the invariants, and the conjecture are expressed in a `logic language`, and the answer is may be `Yes/Not` or a proof [attempt] of the conjecture expressed in the above logical language and some `proof language`. Each language has its peculiarities of style and notation.

From a theoretical point of view, it is appealing to choose `languages` which are easy to describe and manipulate formally (or even automatically), however such languages are difficult to comprehend by a human user.

Moreover, from the theoretical point of view it appears sufficient to create a process which takes a correct input and answers `Yes` at the end. However, in practice this rarely happens: rather, the user needs a tool which allows him to extract useful information from failures in order to improve the input. Moreover, even in the case of success, the user might desire a more convincing argumentation.

The *Theorema* Language

Specifications, invariants, and conjectures can be expressed in the logical language of *Theorema*, which is practically identical to the mathematical language used by mathematicians and engineers: higher-order predicate logic, including the two-dimensional notations. Moreover, *Theorema* allows the introduction of ones own notations and symbols and even creating new graphical symbols [7]. The system provides few simple and intuitive commands for creating and manipulating mathematical knowledge, including its organization into mathematical theories, as well as proving, computing, and solving with respect to a certain theory.

Thus *Theorema* provides an accessible formal language for expressing specifications, invariants, and conjectures, as well as for the organization of the knowledge bases necessary for working in specific domains (e.g. integers, rationals, lists or arrays, sets). During the last few years, such domain specific knowledge bases have been developed in the system.

Programs written in functional style can be expressed directly into this language, thus the "compilation" step (and its possible errors) is avoided. However, for the users which are more comfortable with the imperative style, *Theorema* features a *procedural language* [5] composed of few simple and intuitive constructs. Again few simple commands allow the definition of programs and procedures, their execution and their analysis (by generating conjectures which are understood by the rest of the system).

The Verification System

A verification system based on Hoare-Logic uses a certain version of the axioms defining the semantics of Hoare-triples as well as the corresponding rules for the generation of weakest preconditions. Such a system is implemented in *Theorema* as a preliminary version, and does not yet exhibit all the requirements discussed below.

Namely, we discuss here some of the features of a Hoare-Logic based system, which appear to be important for practical applications.

Loop invariants and termination terms. For proving the correctness of a `While` or `For` statement, one needs to have a logical invariant. Moreover, for proving the termination of a `While` loop, one also needs a termination term. It is generally agreed [3] that finding automatically such an invariant or term is in general impractical – thus most systems will just ask the user for the appropriate expression. However, in most of the practical situations finding the expression – or at least giving some useful hints – is quite feasible. For practical applications this may be very helpful to the user.

A "hidden" problem in the theoretical treatment of the invariant is the fact that in most practical situations it will also contain information about other parts of the program, which is not related to the respective loop. This may make the task of finding the invariant more difficult, however it may be relatively easy to separate the specific information from the non-specific one by an analysis of the free variables and other characteristics which are easy to detect automatically. This could also provide useful hints to the user.

Procedure calls. From the theoretical point of view, the simplest way of providing semantics (as well as weakest preconditions) for procedure calls is to embed the body of the procedure into the calling program (with the appropriate assignments in order to instantiate and use the formal parameters and to avoid clash with local variables). This is of course not acceptable from the practical point of view (modularity, efficiency).

A better approach is to prove independently $\{P_p\}S_p\{Q_p\}$ (P_p precondition, S_p body, and Q_p post-condition of procedure p), and then to use only P_p and Q_p for defining the semantics of $\{P\}p[\ldots]\{Q\}$. Here again a simple way is to require P and Q to be the appropriate instances of P_p and Q_p. However, this approach exhibits a similar problem with the one mentioned in the previous paragraph: P and Q contain, in general, additional information which is not related to the procedure call. The weakest precondition generator which is now implemented in *Theorema* uses a method [3] which "carries over" the additional information.

It is interesting to remark that verification of code containing function calls can also benefit from the application of a similar method. The effect would be to generate the necessary tests for the applicability of a function (like e.g. divisor not null in a division) directly at verification stage, instead of delaying it until the proof of the conjecture. This is a very simple, but yet relevant, example where the system can help the user to detect an error in the code. Indeed, a failed proof is usually long and complicated, and it may be difficult to detect the missing assumptions which result from failure in functions applicability.

Another possibility for simplifying the task of the prover comes from the treatment of so called "external variables" in procedures. The specification of procedures which use input–output parameters (as e.g. `increment`, `swap`) usually must employ some additional (global) variables. In this case the weakest precondition is relatively complicated, using both universal and existential quantifiers. However, the weakest precondition is greatly simplified if one describes the effect of the procedure by a rewrite rule (e.g. $x \to (x+1)$ for `increment`, and $[a,b] \to [b,a]$ for `swap`).

Proving in *Theorema*

The proofs of *Theorema* use the language of predicate logic for the formal part, and the *natural language* for the explanation of the inference steps. Moreover, the provers use *natural style inferences*, that is, the same type of inferences which are used by human provers. This makes it much easier (in contrast to other systems) for humans to read and understand the proofs produced by *Theorema*, thus to be "convinced" about the correctness of a program. Moreover, by studying a failed proof, the user may get hints about the missing assumptions and consequently about the error in his program.

The use of natural style in proving is not just a matter of proof presentation, but it is also a matter of proving. By imitating the style of humans in predicate logic, but also in various special domains, we are able to incorporate in our system the expertise which was accumulated in the history of mathematics, thus making them more powerful. An important component of natural style proving is the use of complex mathematical algorithms (e.g. CAD, Groebner Bases, combinatorics), which is also smoothly incorporated in the *Theorema* provers by using the underlying library of functions provided by the Computere Algebra system *Mathematica*[8].

Conclusions

Using *Theorema* and some application-oriented version of Hoare-Logic appears to provide a good basis for a practical usage of formal methods in program verification. A successful system can however be constructed only on the basis of an extensive interaction with the possible users, including the case studies of many industrial examples.

References

[1] Franz Lichtenberger Bruno Buchberger. *Mathematik fuer Informatiker I: Die Methode der Mathematik*. Springer, 1981.

[2] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuţa, and D. Văsaru. An Overview of the *Theorema* Project. In *ISSAC 97 (International Symposium on Symbolic and Algebraic Computation*. ACM Press, 1997.

[3] G. Futschek. *Programmentwicklung und Verifikation*. Springer, 1989.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12, 1969.

[5] Martin Kirchner. Program verification with the mathematical software system theorema. Technical Report 99-16, RISC-Linz, Austria, 1999. PhD Thesis.

[6] Ernst-Ruediger Olderog Krzystof R. Apt. *Verification of Sequential and Concurrent Programs*. Springer, 1997.

[7] Koji Nakagawa. Logico-grafic symbols in theorema. In *LMCS'02 (Logic, Mathematics, and Computer Science: Interactions*, 2002. RISC-Linz technical report 02-60.

[8] Stephen Wolfram. *The Mathematica Book, 3rd ed.* Wolfram Media / Cambridge University Press, 1996.