Theorem proving languages for verification[⋆]

Jean-Pierre Jouannaud[⋆⋆]

LIX, École Polytechnique
91400 Palaiseau, France
email: jouannaud@lix.polytechnique.fr
http: //www.lix.polytechnique.fr/˜jouannaud

Introduction

Verification is a hard task, but much progress has been achieved recently. Many verification problems have been shown decidable by reducing them to model-checking finite state transition systems. Verification of infinite state transition systems has achieved tremendous progress too, by showing that many particular cases were themselves decidable, such as timed automata [1] or some forms of pushdown-automata [4]. However, the demand for verification is growing fast, and the industrial needs go far beyond the verification of decidable systems.

Verification of large, complex systems for which the task is actually undecidable is therefore an issue that must be addressed carefully. There are two main requirements. The first is *generality* : any system should be amenable to a user-assisted treatment. The second is *automaticity* : decidable systems should be processed without user-assistance.

There are two main approaches to the verification of complex systems.

The first is based on abstraction techniques. The system is first simplified by finding a suitable abstraction making verification decidable. This approach requires finding the abstraction first, which can be done by using a toolkit of possible abstractions, or by delegating the problem to the user. In the latter case, the abstraction must be proved correct, while the correctness follows from general theorems in the first case

The second is based on theorem proving techniques. The system is first described by using some appropriate language, and the description compiled into some logical formula. The property to be verified is then itself described by a formula of the same language. It is finally checked with the theorem prover.

Both approaches are compatible: abstraction techniques need theorem provers to verify their correctness, and theorem provers need abstraction techniques to ease their proving task. The main difference is therefore in the emphasis. In the following, we concentrate on the second approach.

[⋆] This work was partly supported by the RNTL project AVERROES.

[⋆⋆] Project LogiCal, Pôle Commun de Recherche en Informatique du plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

Theorem Provers

Any theorem prover may be used for a verification task. The questions are : how easy that is ? and, what confidence can we have in the result ?

Let us first address the second, classical question. Proving a logical statement is not enough, since a yes/no answer does not carry any information about its actual proof. Modern theorem provers return a formal proof in case the logical statement is true. The proof can then be checked using a proof-checker which is normally part of the theorem prover. This has important design consequences: a theorem prover should come in two parts, a proof-search mecanism and a proof-checker. If the logic has been appropriately designed, proof-checking is decidable. The confidence in the result of the theorem prover therefore reduces to the confidence in the proof-checker, which is normally a small, easily readable piece of code which can itself be proved[1]. The proof-search mecanism is normally quite complex, since it must contain general mecanisms as well a specific ones for automating the search for a proof for as many as possible decidable cases[2].

The first question is more complex. The requirement here is to ease the work of the user. As already said, the user must first *specify* the problem, which is a question of programming language. Here, we face the usual questions of programming languages: should it be a general logical language, or a specific, problem oriented one ? What are the appropriate logical primitives ? What are the structuring mecanism provided by the language ? Second, the user must specify the property to be proved, a normally easy task. And third, the user should help the system in finding a proof when an automatic search is not possible: deciding which tactics have to be applied, in which order, is his responsibility.

Proposal

We strongly believe that a sensible theorem prover should be based on a simple logical framework whose metatheoretical properties have been thoroughly investigated, ensuring in particular that the logic is consistent (relatively to some formulation of set-theory) and that proof-checking is decidable. This logical framework should provide with the appropriate primitives making the tasks of specification and proof-search easy. In particular, induction is a requirement.

The logic

Since we need to construct proof-terms, the logic should be based on the Curry-Howard principle. So does higher-order intuitionistic logic, but there is no dogma here. More specificaly, we shall use the calculus of modular inductive constructions of Chrzaszcz[5] on which the current version V.8 of the proof-assistant Coq is based. The calculus of inductive constructions is due to Coquand and Paulin.

[1] Typically, its size ranges from a few hundred to a few thousands lines of code.

[2] Being a large collection of so-called *proof-tactics*, its size can range from several ten thousands to several hundred thousands lines of code.

The specification language

The calculus of modular inductive constructions is a very powerful functionnal language, and a very powerful type system with polymorphic, dependent and inductive types. This makes the specification of arbitrary problems at least as easy as in any pure, strongly typed functional programming language. Besides, the module system with parameterized functors eases structuring the specification, specifying the implementation of abstract structures, using parameterized libraries, and reusing it's own abstract code when appropriate. Note that abstract functors may contain universal statements that can later be instantiated.

Using such a calculus requires some expertise, as does the use of a high-level programming language. Engineers may not want learning all the subtleties of that language for using the system. It may therefore be important to provide with an input language fitting their needs. In Coq, this input language is provided as a library of Coq modules (in the previous sense) implementing some form of timed automata. This library, called *Calife*, comes with a graphic interface allowing the specification of transition systems directly on the screen[3]. Writing such a library is a hard task, of course, since it implies proving a lot of basic properties of the Coq implementation of the input language.

Proof-search

In Coq, proof-search is done via an imperative language of proof-tactics. Automating this task would be difficult, although it could be done in a PROLOG-like manner by using a semi-decision procedure for higher-order unification. Depending on the user specification, proof-search could therefore be non terminating. Restricting the format of the user specifications (as it can be done with an input language) could avoid the problem, but this issue has not been investigated for Coq which favours the user-interaction schema.

Incorporating decision procedures

In calculi with dependent types, computations can be made transparent in proofs by identifying two propositions which differ by computationally equivalent subterms only via the *conversion* rule: if p is a proof of Q, and P, Q are *convertible*, then p is a proof of P. Proofs become easier to read, and of a manageable size.

In Coq V8.1, convertibility is based upon functional application and a generalisation of higher-order primitive recursion of higher type. In a prototype version of Coq developped by Blanqui, convertibility incorporates user-defined higher-order rewrite rules[4]. In another developped by Strub, convertibility incorporates Shostak's mecanism for combining decision procedures, as well as a few simple decision procedures satisfying the required properties [3]. With these new, recent developments, proofs are much easier to develop by allowing to use such a proof-assistant as a programming language for verification.

[3] Calife is a trademark of the project AVERROES, see http://calife.criltechnology.com

[4] These rules must of course satisfy poperties that ensure consistency of the logic on the one hand, and decidability of type-checking on the other hand [2].

Using Coq for verification

There has been many experiments of using Coq for verifying complex systems:

Various telecommunications or security protocols have been analysed, some of which by industrial tems (France-Telecom, Trusted Logics, Schlumberger) with the help of the Calife interface, such as ABR, PGM and PIM.

Imparative software written in a powerful input language including higher-order functions, polymorphism, references, arrays and exceptions can be verified with *Why*, a verification conditions generator front-end for Coq.

Security properties of large sofware systems have been proved with *Krakatoa* (for JAVA) and *Caduceus* (for C), which are both front-ends for *Why* that translate their input into *Why*'s language (See http://why.lri.fr/index.en.html).

Future developments

Because the conversion rule may include computations of an arbitrary complexity (full higher-order arithmetic can be encoded in Coq), it becomes important to compile reduction rather than to interpret it. Gonthier and Werner faced this problem with the proof of the four colour theorem, for which they realized that checking all particular cases (by computation, and so the proof term reduces to a single use of reflexivity of equality) would take for ever. A prototype version of Coq is provided with a compiler for computations, which makes the use of conversion feasible for proofs involving large computations. It remains to extend the compiler in order to incoporate the recent extensions of conversion.

For secure applications, it is important to extract the code from the proof rather than to prove an a priori given code. Coq is provided with a proof extraction mecanism. Because of smart-cards applications, the code may be ressource sensitive. We are currently developping a version of extraction in which both the code and its complexity (in time and space) will be extracted from the proof.

Some properties of systems may not be true, but may be true with probality one, a problem we had to face with the protocol PGM. We are again developping an extension of Coq in order to prove probabilistic properties of deterministic protocols, and also prove properties of random algorithms.

Aknowledgements: To the past and current members of the *LogiCal* project.

References

1. R. Alur and D. L. Dill. A theory of timed automata. TCS 126(2):183–235, 1994.
2. F. Blanqui. Definitions by rewriting in the Calculus of Constructions, 2003. To appear in Mathematical Structures in Computer Science.
3. F. Blanqui, J. P. Jouannaud and P. Y. Strub. A calculus of congruent construction, 2004. Check web for updates.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In CONCUR '97: Concurrency Theory, LNCS 1243:135–150, 1997. Springer-Verlag.
5. J. Chrzaszcz. Modules in Coq Are and Will Be Correct. In TYPES 2003, Revised Selected Papers in LNCS 3085:130–146, 2004. Springer-Verlag.