ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Gerard O'Regan and Sharon Flynn (Eds)

1st Irish Workshop on Formal Methods

Proceedings of the 1st Irish Workshop on Formal Methods, Dublin 3-4
July 1997

Paper:

The Use of Theorem Provers in the Teaching and Practice of Formal Methods

Helen Lowe

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

The Use of Theorem Provers in the Teaching and Practice of Formal Methods

Helen Lowe

Department of Computing, Napier University
Edinburgh, EH14 1DJ

Abstract

Our aim is to make formal methods, in particular that which has to do with proving properties of programs, more accessible. Our immediate objective is the provision of *usable* tools by applying principles from HCI to the design of semi-automated theorem provers. In this paper we describe the *XBarnacle* semi-automated proof tool based on the CLAM proof planning system. This system is more powerful and more usable than the automated system on which it is based. The user can interact with the graphical user interface, which is in the form of a proof tree, directly, to redo steps or seek more information about the proof process. We have observed the use of the tool amongst both researchers and novice students of formal methods. A pilot evaluation has already taken place, and actively informs future development.

1 Introduction

In this paper we describe the use of the *XBarnacle* proof tool. A description of this system may be found in [12]. We have used an earlier version of this tool, based on the CLAM theorem proving system, for instructing students in proof techniques. We learnt from this experience [11] and as a consequence built *XBarnacle*, which although incorporating many of the original features, and the same underlying theorem prover, employs a different interaction style. We believe that this style is more in keeping with that needed to develop useful, *useable* proof tools for formal methods practitioners, not just for students.

2 Automated theorem provers

2.1 Automation versus interaction

Interactive theorem provers require the user to determine the steps to carry out, and should perhaps be regarded as proof checkers. They are characteristically difficult to use. Complete automation is difficult to achieve. Automated theorem provers are limited in what they can achieve unaided, and the user often has a large role to play, essentially in providing the right environment for the system. We report on this elsewhere, but one example is the use of lemmas in a proof – it may be the user's perspicuity in spotting that a lemma is needed, and providing it, that saves a proof attempt from failure.

2.2 Tactic-based theorem systems

Tactic-based theorem provers, for example NuPRL [6] arose from the need (for humans) to raise the level of interaction. Many are used as interactive systems in which much of the tedious low level and repetitive steps are carried out automatically, the user providing the high level guidance for the proof. This style of theorem prover shows the way to incorporate more automation: if the reasoning behind the user's choices can be encapsulated in a computer program, we may be able to build automated theorem provers.

2.3 The CLAM-OYSTER system

The theorem proving system CLAM [4] uses the underlying logic provided by Oyster, a tactic-based system which is essentially a rational reconstruction of NuPRL. Both Oyster and CLAM are implemented in Prolog. CLAM's tactics are specified in a meta-logic by the pre- and post-conditions of *methods* of the following form:

```
method(name(Args),
      Input,
      Preconditions,
      Effects,
      Output,
      tactic
    ).
```

The *preconditions* slot provides a specification of when a tactic is applicable, and the *effects* slot computes the output: this may be empty, as when a tactic finishes off a proof (or a branch of a proof – proofs are tree structures in general); or it may contain one or more sub-goals.

CLAM uses a technique known as *proof planning* [3] to prove theorems automatically. First a method is found which is *applicable* to the original conjecture (goal), by checking that the *preconditions* hold for the goal. The planner then finds suitable methods for the sub-goals in the *Output* slot, and so on until all branches have been proved.

Although CLAM has been used for other proof methods, the most work has been carried out on proving theorems by induction. This is because of their difficulty and therefore interest, and because of the duality between proof by induction and recursive programs. Proving properties of a recursive function will involve proof by induction. In what follows, we restrict the discussion to these proofs.

2.4 Limitations and potential

CLAM employs a small collection of powerful methods. The methods of induction, generalization, symbolic evaluation, and *rippling* (of which, more later) are all that is needed for the majority of theorems. It also does surprisingly well without the provision of lemmas: for some theorems where Nqthm [2], for instance, would need lemmas provided by the user, CLAM can prove these – often special cases of them – in line as part of the main proof. However, users of CLAM, particularly novices, experience difficulties in using CLAM.

1. Entering new definitions and theorems is problematic. There are not good browsing or editing facilities. Getting a definition both correct and in a useful form is non-trivial, especially for inexperienced people.
2. If the planner makes a wrong move, it is not possible to intervene to change it. Very experienced users have a battery of tricks and workarounds to prevent this – novices do not, and it could be argued that others should not need to.
3. The output is hard to understand, and there is not much flexibility in *what* is shown – the “all or nothing syndrome”, common to many systems.

CLAM has many positive features which suggest how these might be rectified.

1. The level at which the planner operates – that of the methods – is similar to how humans tackle problems, and suggests a way of displaying the output.
2. The method preconditions are written in a meta-logic; they were originally thought of as being declarative in nature, which suggests that they could be used as the basis of communication with the user, for example in providing explanations.

However, to make use of these features we needed:

1. To rationally reconstruct the preconditions so that they were truly declarative.
2. To provide a proper interactive architecture for the system.

The resulting system was known as Barnacle.

3 The Barnacle proof tool

3.1 Interactive features

We originally built a system for use on PC Windows systems using LPA Prolog. This system was able to give explanations (see below), and the user could interact with the system. Our original architecture did not allow for unforeseen interaction: in some cases the system could second-guess the user, and invite them to, for example, input a crucial lemma, but otherwise the user had to decide in advance (or maybe as a result of a previous attempt) which controls to set. The user could

- demand explanations for some or all methods
- request the power of veto over some or all methods.

This inadvertently echoed the architecture of previous, non-interactive versions of CLAM, where premeditation was the key to successfully proving some theorems, notably by providing hints, or by using Prolog pattern matching, to force CLAM to take some step which it would not otherwise carry out by default. After watching students trying to use this system, we soon decided that the user, not the program, must be in control, but that the program should attempt to prove the theorem unaided, stopping only on a signal from the user. This, whilst retaining the proof planning system, necessitated a fundamental change in architecture. With the new mode of use, presentation of the proof attempt became all-important – without good visual clues, the user does not know when or how to intervene in a proof. Our previous work reported in [10] provided a basis for designing the graphical interface, demonstrating that users preferred a tree-structured, hierarchical mode of presentation, and we built a new version, XBarnacle, incorporating these principles, shown in Figure 1.

Figure 1: The XBarnacle Graphical User Interface

3.2 Explanations

An early claim for proof planning with its specification of tactics by means of preconditions was that this would enhance the explanatory power of systems. For example, the induction method has the following preconditions (we assume for the sake of simplicity that induction is over one variable only but the argument generalizes to two or more variables for simultaneous induction):

1. There is at least one universally quantified variable.
2. Given an induction schema applicable to such a variable, there is at least one matching rewrite rule that can be applied to the induction conclusion.
3. Optionally, various other heuristics may apply. For example, it is good if *all* occurrences of the induction variable can be rewritten using available rewrite rules.

If preconditions are encoded in this declarative way, then we can easily produce explanations of the form

We can perform induction on x
 because x is a universally quantified variable
 and there are available rewrite rules for ...

or

We cannot perform induction on y
 because although y is a universally quantified variable
 there are no available rewrite rules for ...

When we came to develop Barnacle, however, we found that because preconditions had not hitherto been used for explanations, most were *ad hoc* and procedural in nature, written in a style of Prolog that while perhaps efficient was not suitable for use in explanations. As an example, take the preconditions for generalizing a compound term by a variable as in

$$s(s(x)) + (y + z) = (s(s(x)) + y) + z \Rightarrow^{u=s(s(x))} u + (y + z) = (u + y) + z$$

CLAM's existing preconditions were of the form

There is a sub-term X on the left-hand side

X obeys test₁
 X obeys test₂

 X obeys test_n

The same sub-term X exists on the right-hand side

In practice, this involves a lot of backtracking as CLAM comes up with sub-terms which fail one or other of the tests (various heuristics designed to avoid "trivial" rewritings); when finally a term exists which passes all the tests, it may be found not to in the right hand side of the expression. Reordering the preconditions just causes a different backtracking pattern. Any explanations produced from such preconditions are wallpaper-like in scope and not very useful.

What is really needed is

There is a sub-term X on both the left and the right hand side such that

X obeys test₁
 X obeys test₂
 $\dots \dots \dots$
 X obeys test _{n}

This trivial recasting of the preconditions makes all the difference to explanatory power. Most preconditions in CLAM needed this recasting.

The next question to be answered is who the users are, in other words: who are the explanations intended for? Are the needs of students and formal methods practitioners different, for example? The early versions of Barnacle were aimed at the developers of CLAM, in common with many interfaces to theorem provers which are developed in the first place to serve the needs of existing users of the systems. Here we might see explanations such as the one for induction above cast as:

We can perform $s(x)$ induction on x
 because x is a universally quantified variable
 and all/some wave occurrences of x are unflawed

This does not make much sense to anyone outside a small community of user-developers, yet it is of great interest to them, as the unfamiliar terminology comes from the rippling method use of which, as we shall see, is a vital guarantor of termination, a highly desirable property. Although students can use learn this terminology, they do not seem to remember the details very well, and it is an open question – pending further evaluation and testing – as to whether it is necessary to understand a failing induction in terms of which heuristics are broken for the various alternative induction schemas. In our investigations, we found that most users are happy with a “suck it and see” approach: they will watch the first few steps of a chosen induction and on that basis decide whether to leave it be or force the prover to try the next one on the list instead. We believe that the graphical user interface obviates the need for much textual explanation – a picture truly being worth a thousand words.

We hope that a one-to-many mapping of preconditions to explanations is possible, where different explanations, including none in some cases, are produced from the same declarative preconditions according to the class of user.

4 Evaluation and Conclusions

4.1 Method of evaluation

Jackson [9] reports on a pilot evaluation of the new architecture carried out by observing users’ performance on theorem proving tasks using XBarnacle. In this study we observed that users with a good understanding of the task – in this case, proof by induction – were able to use XBarnacle very effectively, intervening quickly in exactly the right places to steer the system towards a proof. We intend to carry out more extensive studies; this pilot study was primarily to teach us about the issues involved in evaluation. There is virtually no evaluation of theorem provers in terms of tasks or users – most evaluation is in performance related terms of the systems themselves such as theorems provable, time taken, and so on. We are undertaking a systematic study whereby we observe and evaluate users attempting various tasks. Subjects will range from novice students of formal methods to more experienced users.

Students using the new interface appear to be more successful than the previous cohort, who used the more premeditated style. When doing paper proofs, they structure these better, reflecting the hierarchical presentation of XBarnacle.

Some researchers remarked that they would like to delve inside the boxes to discover which individual rules had been applied. We asked the students to provide such steps (and explanations) themselves. This they appear able to do without too much difficulty, provided they have the rules in front of them.

Overall, the policy of allowing the user to invoke the extra information they need, rather than either thrusting it upon them or making them use switches beforehand or with hindsight, seems to work well.

4.2 User misconceptions

Our early studies show that users may be surprised and disoriented by the complexity and length of “simple” proofs, forgetting that although the *theorem* may be simple, its *proof* might not. A user may be tempted to intervene in a proof when in fact the theorem prover left alone will find a proof eventually. This problem may be partially overcome by exposing the novice to example “easy hard” proofs. Thereafter, success is reliant on the user being able to distinguish between slow progress and floundering. Developers of methods need to pay more attention to producing more human-like proofs. The critics mechanism [7] is a good technique for this end, as for example a method critic might try to speculate (and prove) a lemma to be used to further the current inductive step, instead of abandoning the rewriting in favour of a nested induction. Using lemmas in this way is a very human-like activity. Michael Jackson aims to incorporate critics into Barnacle as part of his PhD work, building on previous experiences of interactive proof critics [8].

4.3 Some unresolved issues

In order to enable effective interaction, users need systems that communicate at their level, and which present the task in the same terms as they usually think about it, thus appearing to have the same task view as they do. Some of the detail of CLAM’s methods – notably the use of rippling – represents a departure in this respect.

Rippling is a powerful rewriting technique used for the step cases of inductive proof, in which both rewrite rules and the induction conclusion are annotated. The details may be found in [5] and in [1]. Annotations are used on the induction conclusion to depict the differences between it and the induction hypothesis. For example, if the induction hypothesis is

$$x + (y + z) = (x + y) + z$$

then the induction conclusion will be annotated as

$$\mathbf{s}(x) + (y + z) = (\mathbf{s}(x) + y) + z$$

Rewrite rules are also annotated. During the rewriting process, where the aim is to mutated the induction conclusion as far as possible towards a copy of the induction hypothesis, when a rule is compared with a sub-term in the induction conclusion, not only the syntactic structure but also the annotations must match.

Annotations are placed on rules so as to decrease a metric: repeated application of such rules to the induction conclusion can be guaranteed to terminate, since the metric will decrease. One condition on annotated rules is that they must be *skeleton preserving*: for example the rule

$$\mathbf{s}(X) + Y \Rightarrow \mathbf{s}(X + Y)$$

may be annotated thus:

$$\mathbf{s}(X) + Y \Rightarrow \mathbf{s}(X + Y)$$

Ignoring the parts annotated by heavy type – the *skeletons* – the left and right hand sides are identical.

We see that this rule may be applied to both the left and right hand sides of the example induction conclusion above to give

$$s(x + (y + z)) = (s(x + y)) + z$$

and again to the right hand side to give

$$s(x + (y + z)) = s(x + y) + z$$

Another (annotated) rule cancels the ss to give an exact copy of the induction hypothesis and the step case is done.

This particular rule above has other possible measure-decreasing, skeleton-preserving annotations (not shown here) so it does have at least one, and is admissible in a step-case proof. However, the rule

$$X + Y = Y + X$$

has no such annotations, since it is not skeleton-preserving. Disallowing such rules in the cause of termination means that human users maybe surprised by the seemingly unnecessarily complexity of proofs.

The choices to be made, and the trade-offs, are essentially between supporting the user's style of theorem proving, and the "rippling" style. An allied question is how far we can expect the user to come in learning some of the theory behind the theorem prover's approach, and whether this might be beneficial.

As far as the proof planning method is concerned, there may well be benefits in conveying this to the user. After all, what we would be teaching is a *strategy* for proving theorems by induction – something which is missing, or could do with reinforcing, in the mathematical education of computing students.

As to the rippling story, this is more difficult. In the author's own experience, it is comparatively easy to put over the idea of a measure, since this accords with students' knowledge of metrics in general, and the concept of such a measure guaranteeing termination as a good thing. The details need not be taught, as long as students are happy to accept that certain rules – notably commutativity – will not be used by the theorem prover in the step cases of proofs.

A related question is that of notation. Developers of CLAM, who understand the rippling annotation, expect to see some graphical representation of it in any displays of sub-goals or proof fragments. For students learning about rippling, this is also useful. But for practitioners who do not wish to analyse the proof at that level of detail, such displays are distracting and perhaps confusing. In XBarnacle, the display of different kinds of annotation can be switched on or off, and the system customized so that XBarnacle starts up in the mode most desirable for the given user.

Two choices, and the associated benefits and problems are:

- Strict application of the rippling method, all rules applied must be skeleton-preserving and measure-decreasing.

The benefits of restricting rewriting to this powerful control technique have been outlined above. The style of interaction implied is that the user lets the prover go, with a high degree of trust, even when it does not always apply the "obvious" rule, stopping it only when it is "obviously" going wrong. Cases of the latter might be applying an over-generalization, when the user spots that the new conjecture is not a theorem (this happens sometimes), or when there is an obvious divergence of the sub-goals, as where an $s(x)$ expression in the previous induction is replaced by an $s(s(x))$ term in the next induction below (users will generally halt CLAM once they spot a $(s(s(s(s(...))))$ pattern).

- The user intervenes early and often – when they spot a rewriting that they would make, and ignoring questions of measure and skeleton preservation.

This approach is fraught with problems, not nonetheless feasible. First, some preconditions must be made into soft constraints, overridable by the user. Secondly, some measures must be taken to save users from themselves. The chief problem is that of non-termination and looping, which of course rippling is designed to avoid. One way would be to allow commutativity laws etc. as lemmas, and to restrict their use. For example, with a good library browser they could be stored as lemmas and invoked by the user only to finish off a branch of a proof.

5 Summary

Starting from a theorem prover with a command line interface, CLAM, we have built two versions of a semi-automated theorem prover with a graphical user interface. The first put too much onus on the user to predict what intervention would be required, although it made some useful contributions to the study of how users' prefer to see proofs displayed, and how they wish to interact. The second, whilst *allowing*, does not *force* the user to interact, using an effective graphical, hierarchical, tree display to provide the user with visual clues as to when and how to intervene in the proof attempt. Having built such a tool, we are now in a position to investigate much more systematically than hitherto:

- what the user *needs* to know
- what the user *wants* to know
- *how, when, and at what level* to represent information.

We hope in this way that we can provide a tool which is not only useful for teaching students formal methods, but that they will also find it useful enough to take away with them, thus furthering the cause of proof in software development and verification.

6 References

- [1] DA Basin and T Walsh (1994). Annotated rewriting in inductive theorem proving. Technical report, Max Planck Institute, Saarbrücken.
- [2] RS Boyer and JS Moore (1979). *A Computational Logic*. Academic Press.
- [3] A Bundy (1988). The use of explicit proof plans to guide inductive proofs. In R Lusk and R Overbeek (eds.), *Ninth Conference on Automated Deduction*, pages 111 – 120. Springer Verlag.
- [4] A Bundy, F van Harmelen, C Horn, and A Smaill (1990). The Oyster-Clam system. In Stickel, M E (ed.), *10th Conference on Automated Deduction*, pages 647-648. Springer-Verlag, Berlin. Lecture Notes in Artificial Intelligence No. 449.
- [5] A Bundy, A Stevens, F van Harmelen, A Ireland, and A Smaill, (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62, pages 185-253.
- [6] RL Constable, SF Allen, HM Bramley, *et al.* Implementing Mathematics with the NuPRL Proof Development System. Prentice Hall, 1986.
- [7] A Ireland, and A Bundy (1994). Productive use of failure in inductive proof. Technical Report, Research Paper No. 716, Department of Artificial Intelligence, University of Edinburgh.
- [8] M Jackson (1996). HCI Techniques for Theorem Proving. Unpublished MSc dissertation, Department of Computing and Electrical Engineering, Heriot Watt University, Edinburgh.
- [9] M Jackson (1997). A Pilot Study of an Automated Theorem Prover Interface. To be submitted to the *Third Workshop on User Interfaces for Theorem Proving*, September, 1977.
- [10] H Lowe, A Bundy, and D McLean (1996). The Use of Proof Planning for Co-operative Theorem Proving, University of Edinburgh DAI Research Paper 745.

- [11] H Lowe, A Cumming, M Smyth, and A Varey. Lessons from Experience: Making Theorem Provers More Co- operative. Proceedings of the Second Workshop on User Interfaces for Theorem Provers, York, July 1996
- [12] H Lowe and D Duncan. XBarnacle: Making Theorem Provers More Accessible. To appear in the Proceedings of the Fourteenth Conference on Automated Deduction, Townsville, Australia, July 1997.