Lessons From Experience:

Making Theorem Provers More Co-operative

Helen Lowe, Andrew Cumming, Michael Smyth, & Alison Varey

Department of Computer Studies, Napier University, Craiglockhart, Edinburgh EH14 1DJ

Abstract

We describe our experiences in trying to build a co-operative theorem proving system. Our model of co-operation is that of a user and an automaton combining forces to prove theorems in a semi-automated theorem proving system. We describe various undesirable behaviours of interactive and automated systems and set out our initial objectives. We evaluate our early attempts and, in the light of this experience, draw up a tentative wish-list for future systems.

1. Interactive and Automated Theorem Provers

Theorem provers may be completely interactive, totally automated, or somewhere in between. It is unusual for a theorem prover to stay all its life at one end of this spectrum. For example, users of an interactive proof checker (or proof editor) will most probably write tactics to alleviate the tedium, which may be stored in libraries for reuse. Users of automatic theorem provers find in practice that the range of problems that their system can do while they press the button and go off to coffee is disappointingly small, and start implementing facilities for interaction. The danger with this activity is that it tends to be piecemeal and *ad hoc*. Whatever is implemented is never quite enough, at least for users who are not themselves implementors or developers of the system. If we persevere long enough, however, we can get quite good at "workarounds". We become used to the foibles and limitations of our system and push it over new frontiers. We become "expert users".

Where does our term "co-operative" fit in to this spectrum? It is easy to characterize a theorem prover which fails with the single word "no" as uncooperative. However, lack of co-operation takes many guises.

- A *silent* theorem prover. We get no inkling of why it has failed at all.

- A *verbose* theorem prover. At the end of its failed proof attempt we can paper our living room with the output. This will at least give us something to do as we ponder the outcome.

- An *incomprehensible* theorem prover: it is hard to understand its terminology or its syntax.

- An *inflexible* theorem prover. We know what is needed but it will not let us do this.

- A theorem prover which does *a lot of search*. Which failing path is most promising? We do not know.

2. Explanations

A minimum prerequisite for of a co-operative system seems to be some explanation facility. The system should be capable of telling us why something failed (or even, for the curious, why something succeeded).

A claim made for *proof planning* (Bundy, 1988) was that it should facilitate explanation. The CLAM theorem proving system (Bundy *et al*, 1990) specifies tactics by *methods*. These methods describe preconditions (written in a defined metalanguage) under which the tactic will be applicable. In practice, the authors of CLAM's preconditions had not written them with explanation to a human user in mind, so in their initial raw form they were not, in fact, useful for providing explanations.

A first step in developing BARNACLE was to rationally reconstruct the preconditions of CLAM methods so that it was indeed feasible to present textual explanations of what was going on. This was one of our more successful outings and resulted in explanations such as

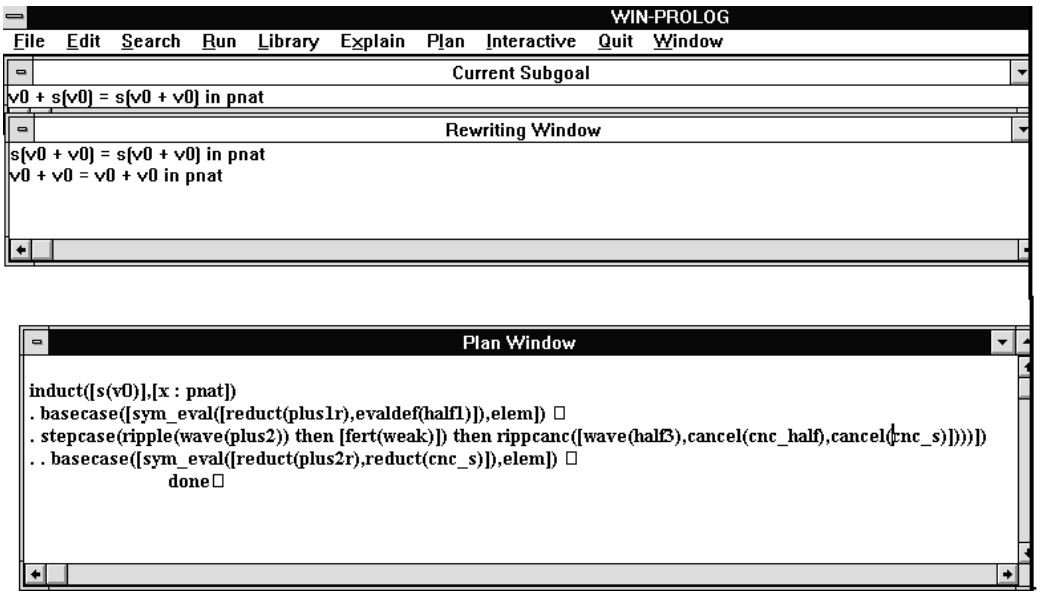
"Rippling is blocked. Can you think of any more wave rules to try?"

Assuming that the user know what rippling (Bundy *et al*, 1993) was, and recognised a wave rule when they saw one, this allowed extra rules (lemmas) to be loaded and the proof to proceed otherwise uninterrupted.

BARNACLE could also give accounts of the recursion analysis implemented in CLAM which is (largely) a rational reconstruction of that of NQTHM (Boyer & Moore, 1979). It is a heuristic measure of how likely each possible induction schema is to succeed on the current conjecture and as such does not always provide the best answer, although it quite often does.

3. Evaluation of BARNACLE

As reported in Lowe, Bundy, & McLean (1996), we built a version of BARNACLE comprised of an explanation mechanism which the user could switch on and off for each method to allow explanations to be provided for either or both of succeeding or failing preconditions; and a veto facility: this allowed the user to veto the use of either all methods proposed by BARNACLE, or certain selected methods known to cause problems: for example induction (because CLAM sometimes chooses the wrong one, as intimated above), and generalization (which can result in a non-theorem being conjectured). This interface, written largely in LPA Prolog for windows, is shown in Figure 1.



Three windows are open: current subgoal; rewriting; and plan. The current subgoal window shows the formula that BARNACLE *is currently trying to prove. The rewriting window shows the chunk of proof just tackled. The plan window shows the plan which has been constructed. We were initially pleased that we had separated out different kinds of information into different windows.*

Figure 1: The first BARNACLE interface

The immediate failing of this system which is obvious from the figure is that the screen is somewhat cluttered. In addition to the three windows open there is a bewildering array of iconized windows which the user may double click on to solicit more information. The question which we (eventually) asked ourselves was: how much the information does the user *need*? Unfortunately it seemed at first that we potentially needed all of it and made use of iconization and focus to reduce the number of windows which were actually open at one time. When BARNACLE begins the planning process, we see only the conjecture and the empty plan window. If the first applicable method is induction, then following recursion analysis (Bundy *et al*, 1992) the plan window shows the induction schema chosen. This is followed by one or more base cases: a rewriting window shows the steps of each and then disappears. For the step case(s) the induction hypothesis is shown, above the rewriting window for the induction conclusion. The plan expands in its window as each applicable method is chosen. A common comment on this scenario was that the screen seemed “rather busy”.

When do we want these rewritings? Usually only when CLAM is unable to carry out the planning process unaided. However, a surplus of output, however organized, was making it again hard to see the wood for the trees. We started to analyse more

carefully the tasks involved in guiding CLAM when it is failing. We came to the conclusion there are two phases:

1. alerting the user;
2. letting the user decide what action to take.

We hope to undertake a detailed study of how users discover failing behaviour in theorem provers with our new interface, currently under development (Figure 2). In this we show by default *only* the plan tree with the current subgoal and graphical (tree) representation of the plan.

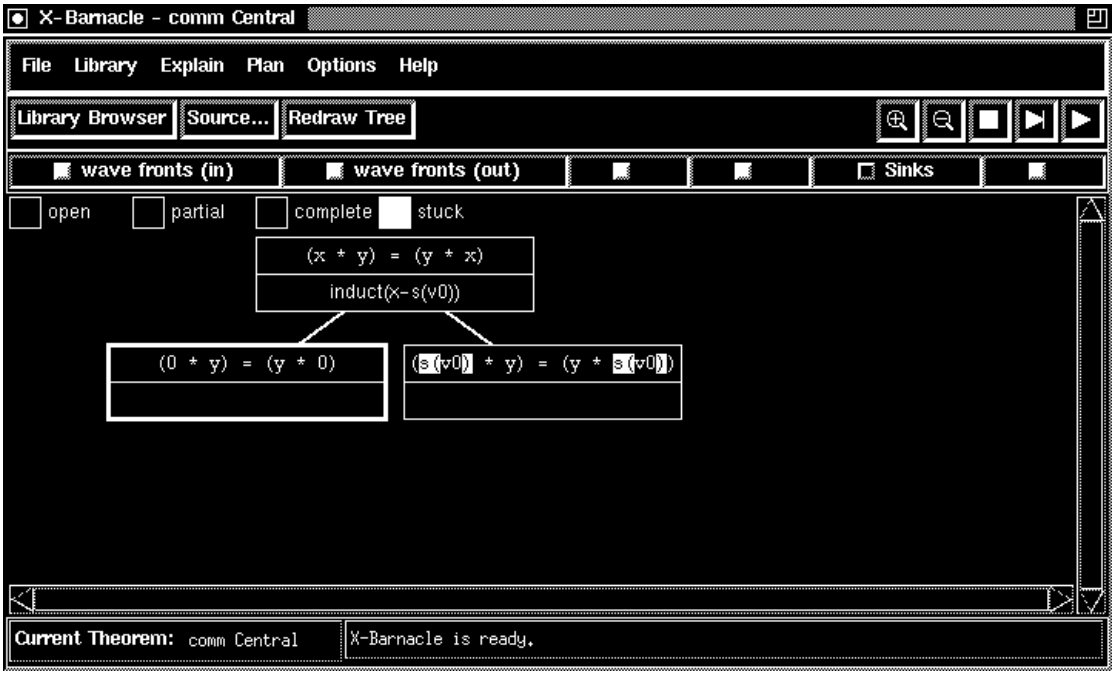


Figure 2: Designing the new BARNACLE interface (UNIX platform)

For now, we have identified some patterns, for example:

- The user spots that the current subgoal is a non-theorem (CLAM will gamely try to prove this subgoal; if we are lucky then it will fail and if we are not then it will not terminate). The knowledgeable user looks above the faulty conjecture for a generalization step: if the original conjecture was true (and of course this will be the next thing to consider) then this is the only way falsehood could have been introduced.
- The user spots that the current subgoal is in some way "worse" than a previous subgoal. It is likely
  - ♦ either that a bad choice was made,
  - ♦ or that a lemma might usefully be introduced.

Once the user knows where to look and what further information they need they may click in the appropriate place to expand a node or elicit more details (see Figure 3).

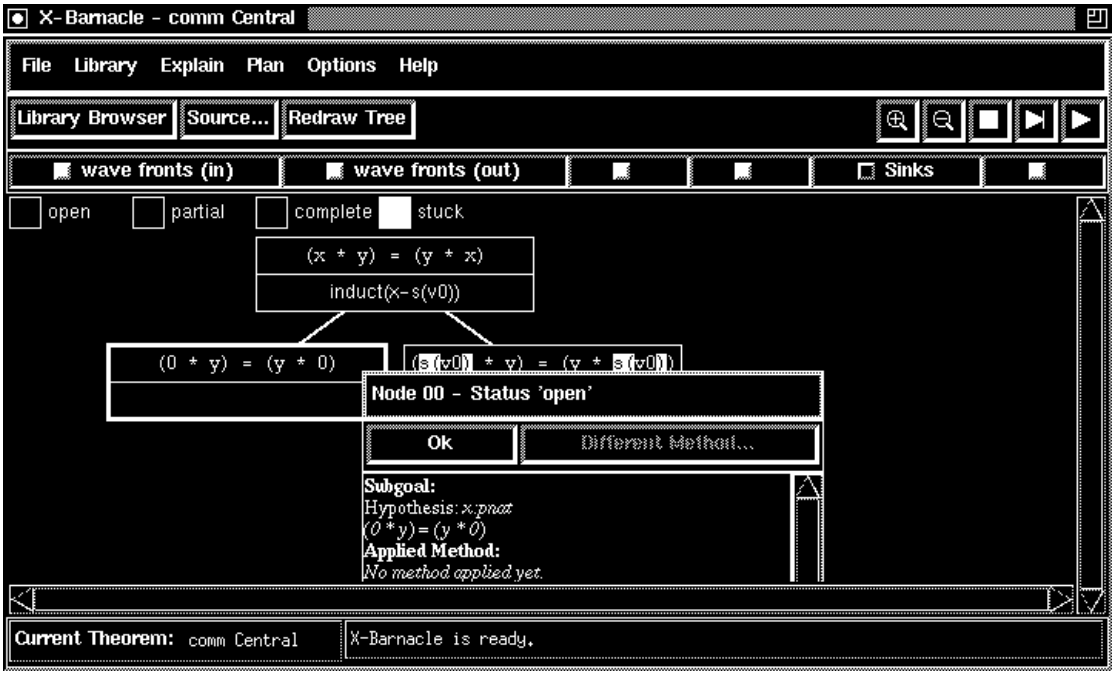


Figure 3: User elicits more details (UNIX platform)

We had already added a graphical representation of the plan to our previous version of BARNACLE; this seemed useful from the point of view of showing the user what was going on (Lowe, Bundy & McLean, 1996), but was frustrating in that the user could not interact with the plan, other than to display more, or less, detailed steps. We decided for the next release to make the central feature of our design a plan with the user could interact. Thus we intend that, for example:

1. The user spotting an overgeneralization should undo that step (and of course those below it), leaving the rest of the plan intact.
2. The user should be able immediately to spot choice points in a toiling branch, and click on them to see what those choices were (and choose alternatives).
3. The user can save their efforts, however meagre, and come back to them refreshed another day.

We do not have a user model, since we have no base of users of the kind of system we are building. We have some idea of the kind of tasks that a user might want to perform, if they could. These seem dependent partly at least on the user's experience and would seem to require a good rendition of the planning process at the right level of detail. The question of introducing lemmas is more of an open-ended problem and one solution is to automate the process of reasoning about them (Ireland & Bundy, 1994). It is an open question as to whether the use of annotations aids the user and we have yet to find a good representation which shows all the options whilst not (again) overwhelming the user. One thing seems obvious, however, and that is the necessity for good library browsing facilities. Figure 3 shows the browser which is incorporated into the new PC version of BARNACLE.

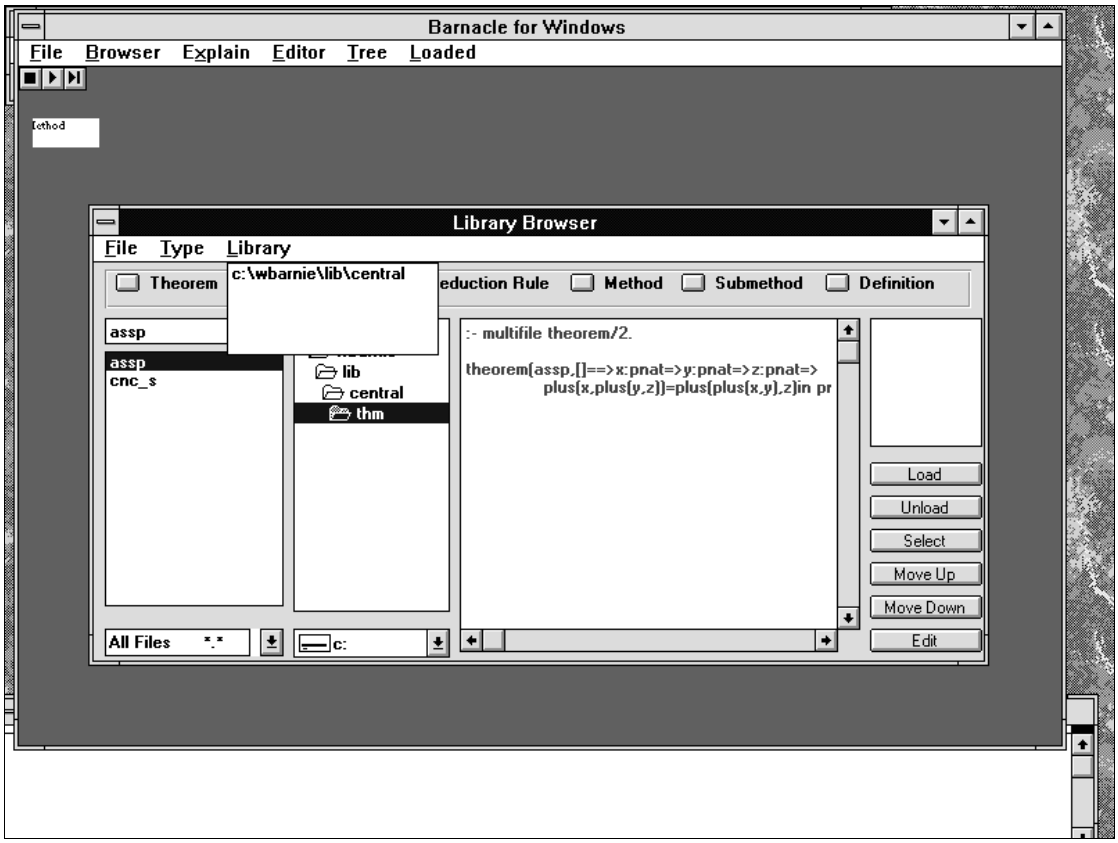


Figure 4: The Browser (PC platform)

4. What do users want? Some untested hypotheses

An early problem in developing BARNACLE was lack of focus - on a sufficiently narrow user group or task. The first motivation for co-operative theorem proving was simply to increase the power of such systems. The next was to increase the power of non-expert users of such systems. Next we wanted novices to be able to use them. Choice of task fluctuated between whatever the trendy theorem of the month was in theorem proving circles, and the "real" examples which occasionally surfaced in our other software development work.

In the end we realized that we should focus more narrowly - and if the system thus developed turned out to have a wider usage then this would be a bonus. We currently think of our catchment area as people like ourselves and our students, who develop software, using formal specifications which call for proofs, which up to now they may well have done by hand. This choice is a pragmatic one - we want to be able to evaluate our system; and to educate our students in using it if it turns out to be useful.

We decided not to focus on expert users of the CLAM system. By definition, expert users have become used to CLAM and have their own workarounds to counter its limitations. We guess that CLAM *is* limited, on the grounds that it does not have many users and people find the learning curve for it quite steep. Expert users have forgotten this and will complain if we provide an interface where they cannot input arbitrary Prolog; and whatever features we provide they will probably want more. This is no bad

thing in itself, but feedback from twenty expert users, if acted upon, would result in twenty different new features and our novices would be baffled again.

So: what do our users want? This is what we think.

1. They want familiar notation. For example, $\forall x \; \forall y \; . \; x + y = y + x$ is better than x:pnat=>y:pnat=>plus(x,y) = plus(y,x) in pnat
2. They would like to be able to see all relevant information on one screen, if that is possible.
3. They prefer to have rigorous proof as well as formal proof; the formal proof is given by executing the proof plan and may be incomprehensible; a good enough rigorous proof may be found secretly lurking in the proof plan with a judicious use of lemmas. Compare this with the work on turning proofs into natural language (Huang, 1994).
4. They want guidance in introducing lemmas. The most up to date and complete account of the use of annotation is found in Basin & Walsh (1994). Since this is such a fundamental feature of CLAM it seems unlikely that the serious CLAM user can escape having to understand something of how annotations are used, if they are to combine with the system to prove theorems co-operatively. Since the story has become increasingly complicated over the years, we seek both a minimalist account and a good minimalist representation.
5. They want to see easily where the choices are. For this reason we are reviving the heuristic planner reported in Manning, Ireland, & Bundy (1993). This finds all applicable methods at each node, and some visual artifice can indicate to the user where the choice points are, so that they do not click aimlessly on nodes where no choice was available.
6. They want to do as little as possible. They want the machine to make as many choices as is feasible.

5. Conclusions

5.1 Testing some hypotheses

We do not yet know what users want from a co-operative theorem prover because we have not yet built one. Our early attempts suffered many flaws and we made many mistakes from which to learn. Our new design is fundamentally different from what has gone before, but uses the proof planning techniques which have proved successful in providing a more flexible kind of automated theorem proving and which seems to give a good basis for co-operative interaction.

We believe we have come far enough to test a prototype system on its intended real life users. Only then can we begin to build a user model, and feed this back into the next prototype.

5.2 Wider issues

In parallel with the need for the theorem proving community to catch up with HCI research, there is a body of work specifically on building *co-operative* human-computer systems (Clarke & Smyth, 1993; Smyth & Clark, 1990). A common initial

objective, reasonable but often misguided, is to try to build an interface on top of an existing system. Unless the original system was designed with co-operative use in mind, it is most unlikely that this will succeed. Usually it is necessary to fundamentally alter the architecture of the "underlying" system. Although we were able to reuse much code, this turned out to be the case here. We had to make the interface the centre of our universe, and arrange the rest around it.

In many ways we were lucky. We were right about the proof planning approach and its amenability for providing explanations. Additionally, the CLAM is quite mature and has a user base and a human databank of accumulated experience. It is comparatively easy to modularize the different mechanisms - planning, testing applicability of methods, use of heuristics, and choice of methods - and experiment by mixing and matching. We look forward to reporting our initial trials.

References

Basin, D A. and Walsh, T (1994). Annotated rewriting in inductive theorem proving. Technical report, Max Planck Institute, Saarbrüken

Boyer, R S and Moore, J S (1979). *A Computational Logic*. Academic Press, ACM monograph series.

Bundy, A (1988). The use of explicit plans to guide inductive proofs. In Lusk, R and Overbeek, R (eds.), *9th Conference on automated Deduction*, pages 111-120. Springer-Verlag. Longer version available from the University of Edinburgh as DAI Research Paper 419.

Bundy, A, van Harmelen, F, Hesketh, J, Smaill, A, and Stevens, A (1992). A rational reconstruction and extension of recursion analysis. In Sridharan, N S (ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359-365. Morgan Kaufmann.

Bundy, A, van Harmelen, F, Horn, C, and Smaill, A (1990). The Oyster-Clam system. In Stickel, M E (ed.), *10th Conference on Automated Deduction*, pages 647-648. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449.

Bundy, A, Stevens, A ,van Harmelen, F, Ireland, A, Smaill, A, and (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62, pages 185-253.

Clarke, A and Smyth, M (1993). A co-operative computer based on the principles of human co-operation, *International Journal of Man-Machine Studies*, 24, pages 3-22.

Huang, X (1994). Reconstructing Proofs at the Assertion Level. In Bundy, A (ed), *12th Conference on Automated Deduction*, pages 738-752.

Ireland, A and Bundy, A (1994). Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16.

Lowe, H, Bundy, A, and McLean, D (1996). The Use of Proof Planning for Co-operative Theorem Proving, University of Edinburgh DAI Research Paper 745.

Manning, A, Ireland, A, and Bundy, A (1993). Increasing the versatility of heuristic based theorem provers. In Voronkov, A (ed.), *International Conference on Logic Programming and Automated Reasoning - LPAR 93, St Petersburg*, Lecture Notes in Artificial Intelligence, pages 194-204. Springer Verlag.

Smyth, M and Clarke, A (1990). Human-human co-operation and the design of co-operative mechanisms, *ICL Technical Journal*, Vol 7 Issue 1, pages 110-126.