Agent-Based Distributed Software Verification

Chris Hunter **Peter Robinson** **Paul Strooper**

School of Information Technology and Electrical Engineering
The University of Queensland, Brisbane, Australia
Email: {chris,pjr,pstroop}@itee.uq.edu.au

Abstract

Despite decades of research, the takeup of formal methods for developing provably correct software in industry remains slow. One reason for this is the high cost of proof construction, an activity that, due to the complexity of the required proofs, is typically carried out using interactive theorem provers. In this paper we propose an agent-oriented architecture for interactive theorem proving with the aim of reducing the user interactions (and thus the cost) of constructing software verification proofs. We describe a prototype implementation of our architecture and discuss its application to a small, but non-trivial case study.

Keywords: Formal methods, software engineering, trusted systems

1 Introduction

Formal methods is the area of research dealing with developing verifiably correct software. Although we have known since the 1970s, for example, how to prove the correctness of imperative programs, such techniques are still not used widely in industry. One of the main reasons concerns the cost of developing the required proofs — constructing proofs by hand is both tedius and error-prone, hence the need for automated mechanical tools to support the process. Due to the complexity of the proofs, tactic-style user-driven interactive theorem provers are the only viable tools. Although they allow the user to mix manual proof construction for the more difficult subproofs with automatic proof construction for the rest via tactics, the number of user interactions required and hence the cost of using such tools is still too great.

One way to increase the cost-effectiveness of software verification is via *distributed* interactive theorem proving, where background execution of tactics is used to both speedup automated reasoning and to make such execution more appealing to the user by allowing them to continue with their own proof construction in the foreground. Accounts in the literature of constructing a distributed interactive theorem prover include Hickey's distributed MetaPRL system (1999), in which the multi-threaded nature is hidden from the user, and the Distributed Larch Prover of Vandevoorde and Kapur (1996), which allows the user to manually spawn proof jobs for background execution.

Although such systems provide some advantage over conventional single-threaded theorem provers, we believe background processing alone is not sufficiently cost-effective for two reasons.

1. The size of the search space typically means the size of the required proof does not increase linearly with the time taken to find the proof, thus limiting the size of subproofs that can be tackled by an automated reasoning component
2. The cost of choosing an appropriate reasoning component for background execution can itself be of comparable expense to constructing the proof manually, since the user usually must understand the proof obligation to make a sensible choice of what to execute.

We address these issues by proposing an architecture for distributed interactive theorem proving based on the Multi-Agent System (MAS) paradigm (Jennings et al. 1998). The architecture allows both autonomous proof construction and cooperation between reasoning components. In the terminology of Zambonelli et al. (2003), it belongs to the class of *distributed problem solving systems* in which component agents are expicitly designed to achieve a given goal. This contrasts with so-called *open* systems in which the agents are not co-designed to achieve a common goal.

The structure of the paper is as follows. We begin by presenting an overview of our agent-based architecture in Section 2. In Sections 3,4 and 5 we discuss the individual agents in more detail. In Section 6 we describe a prototype implementation of our architecture before briefly presenting the results of a software verification case study that uses this implementation in Section 7. We present our conclusions and plans for future work in Section 8.

2 Architecture Overview

Our architecture for agent-based interactive theorem proving consists of a *personal assistant*, multiple *proof agents* and a *broker* — Fig. 1 provides an overview.

A user will have one associated personal assistant whose job it is to monitor the proof the user is working on and (sometimes semi-) autonomously make use of proof agents to construct parts of the proof tree. The personal assistant also provides an interface that allows the user to assist the proof agents by completing problematic subproofs (complementing the autonomous agent-based proof construction). In this capacity, the personal assistant must make sure the user is not inundated with information, especially proof requests.

Proof agents provide the automated reasoning capabilities of the system. A proof agent will typically

Copyright ©2005, Australian Computer Society, Inc. This paper appeared at The Twenty-Eighth Australasian Computer Science Conference (ACSC2005),Newcastle, Australia, January 2005. Conferences in Research and Practice in Information Technology, Vol. 38. Vladimir Estivill-Castro, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

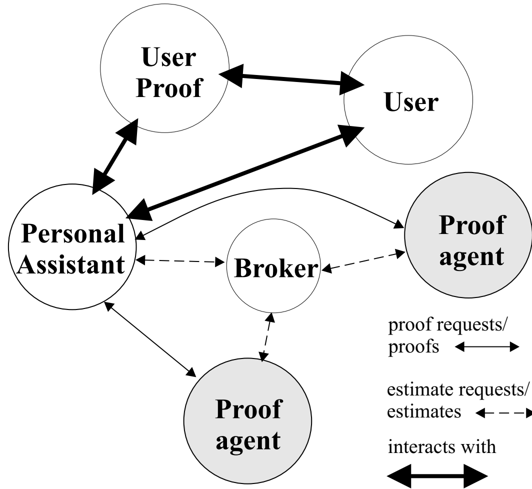

Figure 1: Overview of agent-based architecture

encapsulate a particular reasoning style, search strategy or theorem-proving domain. Apart from providing proofs via a *server interface*, a proof agent must also be able to give an assessment (an *estimate*) of both its chance of successfully providing a proof of a particular goal and of the resources it will require in the process. A proof agent may improve both its performance and the accuracy of its estimates over time (a learning component). A proof agent may also engage the services of other proof agents via a *client interface* for subgoals that it cannot deal with itself.

Finally, the broker agent is essentially a sophisticated service directory — it is responsible for putting proof agents in touch with one another. Proof agents register with the broker initially, possibly with an indication of their general reasoning expertise. When a proof request is received from a proof agent, the broker asks (possibly a subset of) the registered agents for estimates for the problem. The broker may modify the estimates based on historical accuracy before passing them on to the requesting proof agent. The requesting proof agent may subsequently wish to engage in a bidding process via the broker with one or more of the agents that provided estimates. Once a proof agent decides on which estimate (if any) to accept, the agent talks directly to the server interface of the chosen agent. In the language of the *contract net protocol* (Smith 1980), the client agent awards a *contract* to the serving agent.

3 Proof Agents

Fig. 2 provides an overview of the structure of the basic reasoning block of our architecture — the *proof agent*. A proof agent will always consist of a proof engine (the agent's reasoning capability), an estimation facility for determining the proof engine's suitability for a given problem, and a server interface for interacting with clients wishing to make use of the agent's reasoning capability. A proof agent may optionally have a client interface allowing the agent to take advantage of the reasoning capabilities of other proof agents for goals its own proof engine can only partially prove. Following Clark and Robinson (2002), these components will typically be implemented using separate threads. Indeed, because an agent may be constructing several proofs concurrently, the proof engine will itself usually be multithreaded, with one thread for each proof attempt.

Because proof agents do not autonomously choose which subgoals to attempt proofs of, a proof agent

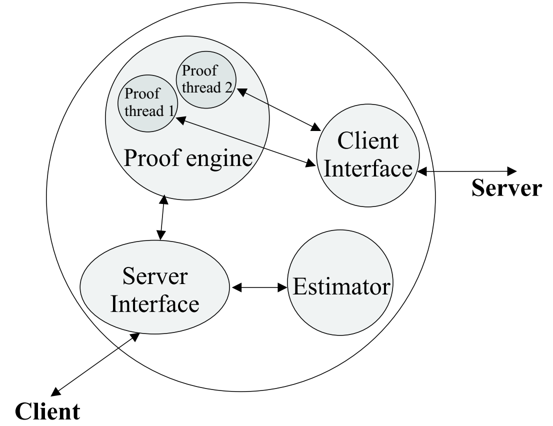

Figure 2: Overview of a proof agent

must have a server interface if it is to be of any use. A server interface supplies two types of information to other agents: proofs and estimates. The provision of estimates is referred to as *bidding*. The estimates provided by a proof agent consist of two parts — a *time* estimate and a *probability* estimate. Given some goal g to be proved, we interpret an estimate $e(p, t)$ as saying that there is a probability p that the agent will successfully find a proof for g within time t. More precisely, t is a bound on the agent's computational resource requirements. Sophisticated proof agents may provide multiple estimates for a given goal, e.g. an agent may vary the amount of search it performs, thus affecting both time and probability estimates — the longer it spends, the greater the probability that it will find a proof. Agents may base their estimates on past performance, the syntactic structure of the problem, or some other calculation.

The client interface of a proof agent is responsible for engaging the services of other proof agents when it cannot prove a subgoal itself. We provide a generic client interface *agent* that encapsulates the required functionality, simplifying the implementation of proof agents. The client interface agent becomes a subagent of a proof agent, executing independently of the proof agent's proof engine. It provides both the interface required to delegate proof jobs, and the functionality to manage the entire process for potentially multiple concurrent proof requests from multiple proof jobs.

The client interface agent uses the following formula to decide which serving agent(s) it wants to attempt a given proof:

> Given a set of estimates E, and a required time t_r (75% of the upper time bound provided by the proof engine), we choose the subset C of E $\{e(p_1, t_1), ..., e(p_n, t_n)\}$ where $\sum_{i=1}^{n} t_i < t_r$ and for which the value of $1 - \prod_{i=1}^{n}(1 - e_i)$ is maximised.

C represents the combination of estimates that collectively provide the best chance of finding a proof within the required time. Having chosen the set of successful bidders, the client interface schedules the jobs to run sequentially — proof agents are stopped once they reach their estimated time bound. Note that for the purposes of this paper we assume all agents are located on a single computational resource. Also note the 75% time bound — this allows the proof agents to overrun slightly when constructing the proof and still allow the client interface agent to meet its required bound.

As an example, one of the proof agents that we have constructed is a proof reuse agent that attempts

to construct a proof of a target goal based on a similar (already-constructed) source proof (Hunter et al. 2004). The agent's probability estimate is based on how closely the source and target goals match. It provides a series of estimates based on how much search it performs attempting to follow the source proof. For subgoals arising in the target proof that have no counterpart in the source proof, or if following the source proof is no longer possible, the reuse agent uses its client interface.

Finally, we have so far assumed proof agents provide a complete proof of a goal — we refer to these as *total* agents. Another possibility is *partial* agents that transform an open goal into one or more subgoals. We incorporate partial agents into our framework via *confidence estimates* that are provided by partial agents in place of the standard estimates. A confidence estimate is the confidence a partial agent has that it can "make progress" on a problem. It is up to the client to decide which (if any) partial agents it wants to use — a simple option is for the client to choose the partial agent with the best confidence estimate, allow it transform the goal, and then proceed as normal. Note that a partial agent would rarely make use of a client interface itself since it is expected that a partial agent produces an incomplete proof.

4 Broker

The *broker* agent acts as a sophisticated service directory. Concretely, for a particular proof request, the broker interacts with known serving agents to acquire estimates that are then modified for accuracy before being passed back to the client. For a given agent, probability estimates are modified for the last n proofs attempted by the agent using an *accuracy* measure acc_m and an *effacing* measure eff_m, which we define as follows:

$$acc_m = \frac{1}{n}\sum_{i=1}^{n} f_i \qquad \mathit{eff}_m = \frac{1}{n}\sum_{i=1}^{n} g_i$$

where for the ith proof (where p_i is the original probability estimate for that proof),

$$f_i = \begin{cases} p_i - 0.5 & \textbf{if} \text{ proof succeeded} \\ 0.5 - p_i & \textbf{if} \text{ proof failed} \end{cases}$$

and

$$g_i = \begin{cases} 1.0 - p_i & \textbf{if} \text{ proof succeeded} \\ -p_i & \textbf{if} \text{ proof failed} \end{cases}$$

The result of the accuracy measure is a number between -0.5 and +0.5 — a negative value indicates the agent would be better off guessing. The effacing measure is an indication of an agent's over/underestimation of its abilities — the sign of the measure indicates over or underestimation and the magnitude indicates the amount. Assuming an agent has completed a sufficient number of proofs for acc_m and eff_m to be significant, the broker combines the measures to produce the broker's modified probability estimate p_b as follows:

$$p_b = (acc_m + 0.5)(\mathbf{min}(1, p + \mathit{eff}_m)) + (0.5 - acc_m)h$$

where for a particular agent, p is the probability estimate provided by the agent and h is the broker's own measure of how likely the agent is to succeed at the current problem — we simply take h to be the historical number of successful proofs divided by the total number of proof attempts made by the agent. For agents with 100% accuracy, i.e., $acc_m = 0.5$, p_b depends entirely on the estimate provided by the agent (adjusted for historical over/underestimation), while for agents with $acc_m = -0.5$, the adjusted estimate depends solely on the broker's own judgment of the agent's chances of finding a proof.

The broker modifies the time estimates of an agent using the $tacc_m$ measure:

$$tacc_m = \frac{1}{n}\sum_{i=1}^{n} f_i$$

where for the ith proof,

$$f_i = \begin{cases} ta_i - te_i & \textbf{if} \quad ta_i > te_i \\ 0 & \text{otherwise} \end{cases}$$

where te_i is the estimated time and ta_i is the time actually taken. For an agent with some $tacc_m$ that submits a time estimate t, the modified time estimate is then just $t + tacc_m$. Note that $tacc_m$ is always positive — it is a measure of the average time an agent exceeds its time estimate.

5 Personal Assistant

The *personal assistant* agent can be viewed as part of a "user" proof agent: the proof the agent is working on is the user proof, the user is the proof engine, and the personal assistant provides the agent's server and semi-autonomous client interfaces.

The personal assistant's client interface drives the background agent-based theorem-proving effort. Agents can be invoked either by the user, or autonomously by the personal assistant itself, to work on problems from both the original source proof, as well as proofs that result from server interface activity.

For the case of autonomous invocation, the personal assistant initially asks each agent for bids for each unproved goal in the user proof(s), giving each received bid a rating according to a function of the form:

$$\mathit{f_PAClientRate}(p, a, 1/t, 1/f)$$

where p and t are the probability and time estimates, respectively, of the bid, a is a term relating to the age of the node for which the bid was made (the time since the node was created) and f is the number of failed proof attempts of the node. Note that we write $1/t$ and $1/f$ to indicate inverse relationships. The greater the age of the node the better, since this decreases the chance of the user starting a proof of the node themselves, thus making the agent's work redundant, whilst we equate proof difficulty with the number of failed proof attempts — the more difficult it is to find a proof of a node, the more likely it will require user intervention. The larger the value of $\mathit{f_PAClientRate}$, the more likely a bid will be accepted. Note that the statement of the function merely gives a general indication of the factors that we have identified as being important — the exact weighting given to each depends on the particular implementation of the architecture. One possibility is for the personal assistant to adjust the weightings based on experience.

Where the responsibility for actually modifying the user proof lies, depends on the theorem prover's user interface and on the variable instantiations of the agent's proof. As a rule, the personal assistant should only modify the user proof if the (agent-calculated)

proof contains no extra variable instantiations, and if modifying the proof does not disrupt any proof efforts of the user; otherwise, the personal assistant should simply provide the user with the ability to view and apply completed agent proofs.

The personal assistant's server interface receives estimate and proof requests just like any other agent, however, since the personal assistant cannot directly "invoke" its proof engine, it must attempt to persuade the user to fulfill proof requests. Why is the personal assistant interested in involving the user in the first place? For proof requests that come back to the personal assistant that are simpler than the subgoals from the user proof from which they originally arose, it makes sense for the user to work on these rather than the original subgoals. So, with respect to its server interface, the personal assistant's *desire*, in the agent sense, is to involve the user as much as possible in fulfilling proof requests without the user's interest in helping out waning over time. The user losing interest equates to a perception that their time is being wasted, possibly a result of the personal assistant over-exaggerating the importance of proof requests, or of excessive numbers of either unprovable subgoals, or subgoals that are part of a larger unprovable goal. Hence, the personal assistant needs to ensure the amount of time the user spends unproductively, i.e., time other than that spent actually constructing proofs to fill useful proof requests, is less than the time saved by the (partial) proofs provided by other proof agents.

As a baseline, the personal assistant must provide the user with enough information so the user can determine the benefits or otherwise of filling a proof request. Apart from a statement of the proof goal itself, the user must also be able to access the entire proof effort up until that point: this includes the proof tree rooted at a node in the original user proof together with information about which agents were responsible for which parts, failed proof attempts and bid information, and estimates of the amount of proof remaining. In particular, the user will often be interested in the other open proof goals — there is no point constructing a proof for one goal if there are other unprovable proof obligations.

Given the typically large number of competing incoming proof requests that might be active at any one time, more important than just presenting information to the user is for the personal assistant to *prioritise* the proof requests, thus saving the user the task of having to navigate through the myriad of requests that come the personal assistant's way. For example, the user's attention should be drawn to a proof request arising from the largely completed proof of a major chunk of the user's main proof ahead of a proof request from an agent that has made virtually no progress and that gave a low probability bid in the first place. For this purpose we use the following function to rate proof requests.

$$f_PAServerRate(R, P, I, L, U)$$

R is the likelihood the unfinished parts of a proof (not including the part of the proof associated with the current proof request) can be completed automatically by the current agents. P is a measure of the progress made by the agents on the original proof request from which the current job is derived. I is a term relating to the importance of the proof from which the request arose. We measure importance simply by the expression $1/p_{best}$, where p_{best} is the best estimate the personal assistant received for the root node in the proof request chain (the node from the user proof). L is the historical likelihood of the user completing a request by the agent involved, i.e., the completed proof requests divided by the total proof requests. Finally, U relates to whether the user or the personal assistant was responsible for originally initiating the proof job — if the personal assistant was responsible, $U = 1$, otherwise U takes on a value greater than 1. This reflects the fact that the user is better equipped to determine the suitability of agents for particular tasks than an automated personal assistant that relies on estimates from proof agents, i.e., there is a greater chance of the user wanting to fill the particular proof request.

As a final consideration for the server interface, we note that the "user" agent is highly asynchronous, i.e., it is likely the proof attempt from which a proof request was made has moved on at the very least, if not abandoned entirely, by the time the user gets around to filling the request. This is especially the case when a user only looks at the proof attempt history (including proof requests) of a subgoal when they get around to proving that subgoal themselves (which we have found to be a useful way to use such a system in practice). We address this by supplying the entire proof request history, rooted at a node in the user proof, to the personal assistant for each proof request. In this way the work that led to the proof request is not lost and the user's efforts in filling in the proof are not wasted.

6 A Prototype Implementation

We have developed a prototype implementation of our architecture in the Ergo theorem prover (Utting et al. 2002) using the QuProlog language (Clark et al. 2001). QuProlog provides support for multi-threaded applications via the Inter-Agent Communications Model (ICM) (McCabe 2000). In the distributed logic programming model provided by QuProlog, threads communicate via message passing. Each thread represents a separate logical deduction — importantly, variables are *not* shared between threads. The implication is that proofs calculated in background threads are separate from the foreground/main user proof.

This separation of proof states fits neatly with our view of reasoning capabilities being encapsulated within a proof agent. However, it does create an issue in relation to conflicting variable instantiations between proof threads/agents. If two agents working on seperate subgoals try to instantiate a common variable with conflicting terms there will be the potential for wasted computation, i.e., if both agents successfully produce proofs, one of the proofs must be discarded. The only way to address this wastage is for one agent to influence the operation of another, for example by ensuring that the agents agree an instantiation is feasible or by actually sharing instantiations. Although this seems sensible, in practice the costs associated with one agent disrupting the work of another, especially if one of the agents happens to be the user, are significant. We thus take the view that if integrating a background proof into the foreground fails, this is because the background proof is of a problem that is no longer relevant to the foreground proof. In other words, if a user has accepted a proof, and by association the corresponding variable instantiations, from an agent, any subsequent proof with a conflicting variable instantiation will be discarded.

To incorporate a proof completed by a proof agent that has been accepted by the user, we rerun/reconstruct the proof in the main user proof. Rerunning proofs simplifies the implementation — any inconsistent variable instantiations between the new proof and the existing proof will arise as part of the rerun, thus causing the process to fail. Further, this

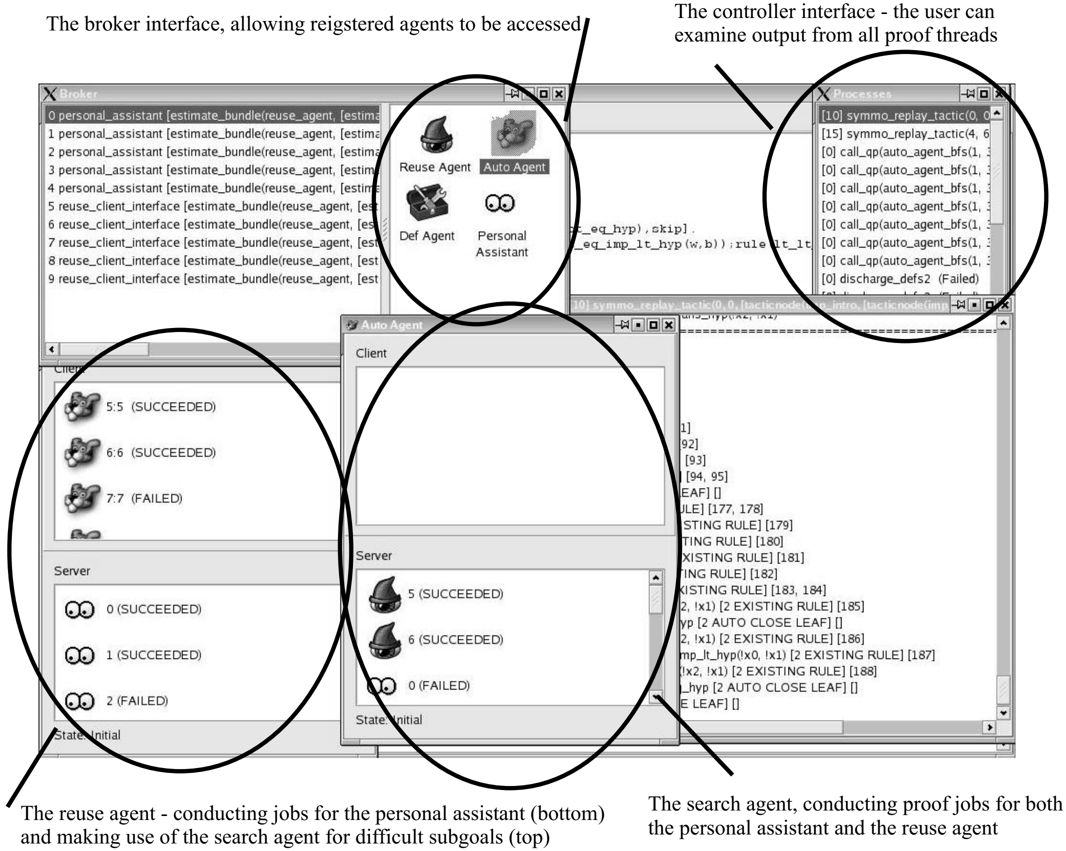

Figure 3: Our prototype user interface

implementation choice means the integrity of the Ergo core, which consequently does not have to be modified, is maintained. Note that since we are only rerunning the proof and not the process that produced the proof, the time taken to rerun proofs is typically negligible compared to the time taken to find the proof in the first place. To simplify the implementation of proof agents, we provide a *Controller* thread that provides methods for creating and destroying proof threads, incorporating the resulting proof into the main user proof, and so on.

The implementation of our core agent architecture consists of approximately 1500 lines of QuProlog code; individual agents range from around 100 lines of code to over 2000 lines (for the reuse agent described previously). Our prototype personal assistant consists of a client interface that rates bids based only on the probability and time estimates (p and t) and a server interface that simply presents an unordered list of proof requests to the user. We have developed a graphical user interface to support the use of our agent architecture; Fig. 3 shows a sample screenshot. Agents are identified by a unique icon which is used in both client/server panes as well as in the broker interface; this kind of visual aid helps the user to monitor the state of the background proof effort.

7 Case Study

As an initial step towards validating the usefulness of our agent architecture, we have used our prototype implementation to verify the correctness of a program that solves Dijkstra's *Dutch National Flag Problem* (Dijkstra 1976). The program partitions coloured array elements in accordance with the Dutch national flag (red followed by white followed by blue). Our verification takes place using a Hoare-style logic (Hoare 1989) — the program, together with pre/post conditions and loop invariant and variant (used to show the loop terminates), is shown in Fig. 4.

$$
\begin{array}{l}
[\![\quad \mathbf{con}\ N : \mathit{int}\{N \geq 0\}; \\
\quad \mathbf{var}\ h : \mathbf{array}[0..N)\ \mathbf{of}\ [\mathit{red}, \mathit{white}, \mathit{blue}] \\
\quad \mathbf{var}\ r, w, b : \mathit{int}; \\
\quad r, w, b := 0, 0, N; \\
\quad \mathbf{do}\ w \neq b \rightarrow \\
\qquad \{\mathit{Invariant} = (\forall\, i : 0 \leq i < r : h.i = \mathit{red}) \wedge \\
\qquad (\forall\, i : r \leq i < w : h.i = \mathit{white}) \wedge (\forall\, i : b \leq i < N : \\
\qquad h.i = \mathit{blue}) \wedge 0 \leq r \leq w \leq b \leq n,\ \mathit{Variant} = b - w\} \\
\qquad \mathbf{if}\ h.w = \mathit{red} \rightarrow \mathit{swap}.h.w.r;\ r, w := r + 1, w + 1 \\
\qquad [\,]\ \ h.w = \mathit{white} \rightarrow w := w + 1 \\
\qquad [\,]\ \ h.w = \mathit{blue} \rightarrow \mathit{swap}.h.w.(b - 1);\ b := b - 1 \\
\qquad \mathbf{fi} \\
\quad \mathbf{od} \\
\quad \{(\forall\, i : 0 \leq i < r : h.i = \mathit{red}) \wedge (\forall\, i : r \leq i < w : h.i = \mathit{white}) \wedge \\
\quad (\forall\, i : w \leq i < N : h.i = \mathit{blue})\} \\
]\!]
\end{array}
$$

Figure 4: A program written in the guarded command language for solving the *Dutch National Flag problem*

Fig. 5 presents an overview of the proof constructed using our architecture to verify the program's correctness. Nodes in the proof tree represent proof steps of similar size — space precludes the entire proof tree from being shown. Filled nodes represent steps contributed by the autonomous background proving efforts, while hollow nodes are user interaction steps. The source of the proof obligations are marked. Four agents were used in the construction of the proof: a *type* agent for constructing type proofs, a *def* agent for showing expressions are well-defined, a *search* agent that uses a bounded exhaustive search to construct proofs, and the *reuse* agent mentioned previously. Constructing the proof without agent support required 203 user interactions — using the prototype implementation of our architecture, this figure

was reduced to 62 user interactions for the particular scenario shown, a saving of around 70%.

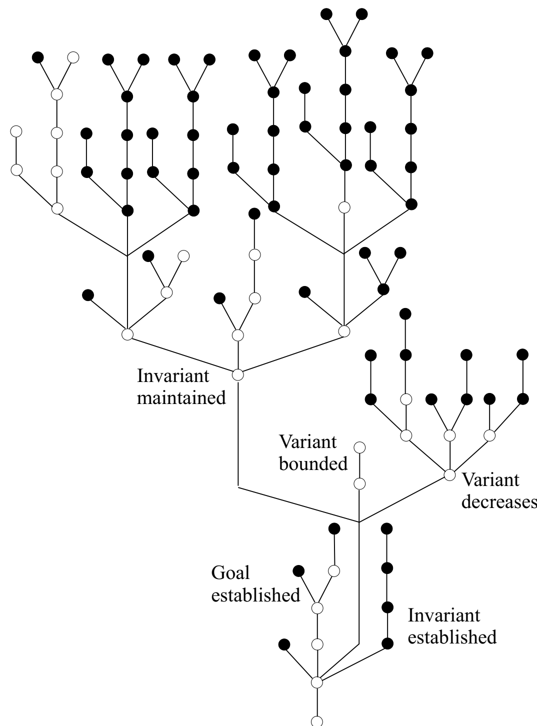

Figure 5: An overview of the proof of correctness of the DNF program

8 Conclusion

In this paper we have described an agent-based architecture for distributed interactive theorem proving. At the centre of our proposal is the idea to encapsulate a reasoning component within a proof agent. Proof agents provide both time and probability estimates for a problem, as well as the proofs themselves. Proof agents can cooperate to solve problems; the proof process is directed autonomously by a personal assistant agent that works on behalf of the user.

Our proposal shares some similarities with the MathWeb architecture (Franke et al. 1999, Zimmer 2003) for automated reasoning. Whereas we are interested in improving the viability of interactive theorem proving, the MathWeb work is more focused on standards, e.g., using KQML as the communication language for integrating existing theorem-proving systems as well as more diverse mathematical systems such as model checkers and computer algebra systems. Importantly, the way we use problem-specific estimates to drive the proving process contrasts with the more general coarse-grained capability assessment employed in the MathWeb system.

In the future we plan to complete a larger case study in a different realm of software verification. As part of this process, we will need to develop new agents, which will help to determine the viability of our estimation approach. We plan to further develop our prototype implementation with particular focus on the user interface, the quality of which we believe has a significant impact on the effectiveness of our architecture. Finally, we intend to extend our prototype implementation, specifically our personal assistant and client interface agents, to cater for the availability of multiple computational resources and to investigate the possibility of a multi-user system with multiple personal assistants.

References

Clark, K., Robinson, P. J. & Hagen, R. A. (2001), 'Multi-threading and message communication in Qu-Prolog', *Theory and Practice of Logic Programming* **1**(3), 283–301.

Clark, K. & Robinson, P. J. (2002), 'Agents as Multi-threaded Logical Objects' , Vol., 2407 of *Lecture Notes in Computer Science*, Springer, 33–65.

Dijkstra, E. (1976), *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey.

Franke, A., Hess, S. M., Jung, C. G., Kohlhase, M. & Sorge, V. (1999), 'Agent-oriented integration of distributed mathematical services', *Journal of Universal Computer Science* **5**(3), 156–187.

Hickey, J. (1999), Fault-tolerant distributed theorem proving, *in* 'Proceedings of CADE-99', 227–231.

Hoare, C. A. R. (1989), An axiomatic basis for computer programming, *in* 'C. A. R. Hoare and C. B. Jones (Ed.), Essays in Computing Science, Prentice Hall'.

Hunter, C., Robinson, P. & Strooper, P. (2004), 'Symbolic proof reuse for software verification', *in* 'Proceedings of AMAST-04', 211–225.

Jennings, N. R., Sycara, K. & Wooldridge, M. (1998), 'A roadmap of agent research and development', *Journal of Autonomous Agents and Multi-Agent Systems* **1**(1), 7–38.

McCabe, F. G (2000), 'The inter-agent communication model (ICM)', Fujitsu Laboratories of America Inc, 2000.

Smith, R. G. (1980), 'The contract net protocol: High-level communication and control in a distributed problem solver', *IEEE Transactions on Computers* **29**(12), 1104–1113.

Utting, M., Robinson, P. & Nickson, R. (2002), 'Ergo 6: a generic proof engine that uses Prolog proof technology', *LMS Journal of Computation and Mathematics* **5**, 194–219.

Vandevoorde, M. T. & Kapur, D. (1996), Distributed larch prover (DLP): An experiment in parallelizing a rewrite-rule based prover, *in* 'Proceedings of RTA-96', 420–423.

Zimmer, J. (2003), 'Proceedings of the Workshop on Agents and Automated Reasoning, 18th International Joint Conference on Artificial Intelligence'.

Zambonelli, F., Jennings, N. R. & Wooldridge, M. (2003), 'Developing multiagent systems: the Gaia Methodology', *ACM Transactions on Software Engineering and Methodology* **12**(3), 317–370.